3400

PROGRAMMING LANGUAGES AND THEIR COMPILERS

Preliminary Notes

Second Revised Version, April 1970

John Cocke and J. T. Schwartz

Courant Institute of Mathematical Sciences
New York University

The Courant Institute publishes a number of sets of lecture notes. A list of titles currently available will be sent upon request.

Courant Institute of Mathematical Sciences
251 Mercer Street, New York, New York 10012

Copyright
Courant Institute of Mathematical Sciences
1970

Table of Contents

Preface..... 1

Chapter

1. Overview..... 6
2. The principal subprocess..... 27
3. The lexical scan.....114
4. Data-directed parsing methods.....138
5. Rigorous results concerning the principal syntactic analysis methods.....274
6. Optimization methods for algebraic languages...306
7. Special purpose languages: LISP and SNOBOL....524
8. The self-compiling compiler.....642

Bibliography.....661

Appendix

I. A bibliography of formal language theory.....693
II. Industrial compiler practice.....729
III. Comparative figures for various compilers.....764

PREFACE

Our aim in the present volume is to describe the inner working of a variety of programming languages, especially from the point of view of the compilers which translate these languages from their original "source" form into executable machine code. While this aim will of course make it necessary for us to describe in some detail the external form of each of the languages which we shall study, no more detail will be given than is strictly necessary in order to make it possible for the reader to gain a clear view of the machine code forms into which the language will be translated and of the problems that a compiler for the language must handle. However, internal description of the languages studied will be carried rather far. Thus the attentive reader of the present work should gain a rather good idea of the methods which can be employed to write a compiler for a given language. On the other hand, he cannot expect to find in this book the detailed account of the source conventions for any language which he would need to use the language.

The present work also aims to treat the specific issues involved in the construction of particular languages only rather broadly, and does not discuss those issues of language design which, while they may contribute in significant ways to the convenience or elegance of a particular language, lie at a level of relative detail. The reader will therefore find that the languages which we discuss at length constitute a collection of elements maximally distinct from each other in form, purpose and spirit. Thus, for example, among the major algebraic languages, only one, FORTRAN, is discussed in any detail. From the standpoint of the present volume, the differences which separate FORTRAN from ALGOL and from other similar languages are sufficiently small as to make a discussion of more than one of these algebraic languagesssuperfluous. For the same reason, we do not discuss MAD, JOVIAL or other languages of this kind in

any detail. On the other hand, we do discuss SNOBOL, LISP, SIMSCRIPT, etc.

Our sequence of chapters is as follows. A first chapter sketches the compilation process in general and discusses the main issues arising in language design, the general logical function of a source language, and similar matters. Chapter 2 contains a systematic discussion of top-down, syntax directed compilation. The issues arising in top-down compilation are described using various metalanguages, derived from the so-called Backus normal form, which serve for the formal description of the flow of compilation. Various subsidiary issues arising in the design of syntactic metalanguages are discussed. Chapter 2 ends with a formal discussion of the Dartmouth algebraic language BASIC which, because of the simplicity of its syntax and semantics, provides a good laboratory example for the detailed development of various significant points.

In the third, relatively brief, chapter, the lexical scan process, which generally forms the front section of a compiler system, is discussed. Since the lexical scan process is quite simple, Chapter 3 is rather short. At its end we discuss procedures for the incorporation of simple macro handling features into a lexical scan program.

Chapter 4 completes our discussion of the syntax analysis of mechanical languages by describing other schemes than the top-down scheme available to the compiler writer. These include various bounded context syntactic analysis methods which, when applicable, are normally faster and more efficient than top-down procedures. Techniques permitting the combination of bounded context analysis with top-down recursive analysis are discussed. The various bottom-up syntactic methods are then compared with the top-down method.

The first four chapters have a highly pragmatic character; theoretical considerations enter them only tangentially. Whereas this pragmatic flavor is that generally intended for the present work, Chapter 5 aims to prevent excessive one-sidedness in this regard by developing a few formal theoretical results concerning syntactic analysis which, in the author's belief, contribute

substantially to a clarification of the pragmatic issues arising in syntax analysis. These results generally concern recursive unsolvability, and serve to indicate the relative power of some of the principal syntactic analysis methods by revealing the extent to which it is or is not possible to replace one syntactic analysis method by another. Various more fragmentary results concerning the relative efficiency of various particular syntactic analyzers are also given in Chapter 5.

The semi-final stage of compilation is the optimization of compiler-generated sketch code. The final step of compilation is the generation of machine code in directly executable form. Chapter 6 treats the issues which arise in these two final compilation stages. This chapter contains an account of various systematic optimization methods that have been devised for use with algebraic language compilers, especially FORTRAN compilers. The issues which arise in optimization are rather multifarious; our lengthy discussion of these issues is of necessity pragmatic rather than theoretic.

Chapter 7 discusses two special-purpose languages, chosen to be as different as possible from the algebraic languages toward which the preceding chapters are directed. We discuss SNOBOL, which is a string manipulation language having an interesting and unusual statement form. We also discuss LISP, a language devised for application to symbol manipulation.

The compiler algorithms described in the first nine chapters of this book are all formalizable in source language terms. Chapter 8 discusses the problems which arise when an attempt is made to use this fact systematically to develop a complete compiler system written in its own language and hence "boot-strapable" with relative ease from one machine to another; a number of existing self-compiling compiler schemes are also discussed and compared.

From time to time in the present work, formal algorithms for various of the processes studied are given. Whereas these

algorithms have been sight-checked by the author, none of them are "certified"; these algorithms are available for use at user's risk. The author will of course be glad to obtain corrections to and certifications of these algorithms, as well as specifications of the algorithms in machine-available programming languages.

It should be noted that this volume consists of a highly preliminary set of notes. As such, it does not cover all of the topics in compiler design that must be discussed. Furthermore, the discussions included are necessarily incomplete. An attempt to mitigate both of these faults will be made in subsequent editions.

It is strongly hoped by the authors that circulation of the present notes in their highly unfinished form will encourage knowledgeable readers to comment on deficiencies and possible improvements in the notes. Detailed technical comments are especially hoped for. In certain cases an effort has been made to survey the literature, with the results contained in the sections of "Notes and Comments" with which this second version of the notes are supplied. Readers aware of significant papers not commented upon, or differing seriously with the emphases given in the comments made are also asked to communicate with the authors.

We wish to thank a number of friends for their active assistance in bringing our work to its present level. Patricia Cundall of the IBM Corporation supplied us with many corrections and helped very actively in the development of the "Notes and Comments" section appended to Chapter 4. Sheldon Best of Decision Systems Incorporated also contributed to this section. Chris Earnest of Computers Sciences Corporation contributed very significantly to the work on optimization described in Chapter 6. Valuable advice on this section was also contributed by Fran Allen and Paula Newman of the IBM Corporation. To all of them, and to all those others who supplied us with advice and corrections we extend our thanks.

CHAPTER 1. OVERVIEW

1. Introduction. The Linguistic Approach to Programming.

To program is to specify, in all necessary detail, the steps required for the computer realization of some desired function. But, if the function is elaborate, the pattern of steps required may be highly complex. Thus the "programming problem" is the problem of expressing complex sequences of instructions and of verifying their correctness. Now, an information structure is complex to the extent that its most concise representation is long; this length (measured in number of characters or bits required) is the information content of the information structure in the sense of Kolmogorov. Since a pattern of information, brought into conjunction with a computer, may be used to generate a longer pattern of information, the minimal representation of any pattern P is in fact the shortest program which can be run on a computer to produce P.

The above very general considerations have direct application to the programming problem. The difficulty of completing, correcting, and testing a program is at least proportional to the length of the program. This difficulty even may be proportional to the square or to a higher power of the length of the program if the program is highly "coherent". In this context we may define the average coherence of a program, informally but in a manner sufficient for our purposes, as the number of other instructions to which the "average" instruction in the program stands in direct logical relation. If the coherence of the program, taken in this sense, is low, i.e. if each instruction has a relatively well defined small logical environment within the program, the difficulty of completing the program may simply be proportional to its length. If, on the other hand, the coherence of the program is high, so that the logical environment of each instruction consists of numerous other instructions scattered through the program, then the

difficulty of completing the program may be proportional to the square or to a higher power of its length.

The above reflections indicate the great importance, in programming a given function, of reducing the length of the required program and of restricting the logical environment of the instructions constituting it as much as is feasible.

Minimal expression implies suitable language; and thus the linguistic method appears as a principal tool for the efficient description of complex function. However, a more primitive step must precede the choice of a language. This step, which itself contributes to the desired minimization, is the expression of the required function in terms of simpler subfunction patterns which are as few in number and as stereotyped as possible. In this sense, effective programming depends first of all on the effective choice of standardized subroutines. Stereotypy brings with it a number of highly significant advantages. In the first place, using only a few basic elements repeatedly, one becomes familiar with them, accurate in their use, and in this way proceeds more rapidly and correctly in writing any desired program. But beyond this, and more significantly, stereotypy has the crucial advantage of permitting second level mechanization. That is, since the stereotyped elements in terms of which a program may at first be expressed are repetitive in form, they may themselves be generated from a more condensed representation by another program. Proceeding in this way one takes a large step toward the desired goal of expression in minimal length, since

(a) a function to be programmed is written in terms of stereotyped subunits,

(b) everything which is truly stereotyped can be generated mechanically,

(c) using a mechanical stereotype generator, one needs only to express the unstereotyped portion of a desired function explicitly in order to program it.

We may remark in connection with the above that adherence to the discipline involved in stereotypy is highly significant. To violate this discipline is to introduce a large mass of additional information into the expression of a function beyond what is minimally required for its definition.
Such additional information may of course have a useful role to play. Often, for instance, information of this sort is introduced into a program for purposes of program optimization. Consider, to illustrate this issue, a given function as expressed on the one hand in a problem oriented or source language, and, on the other hand, directly in assembly or machine language. The information content of the first program is considerably smaller than the information content of the second program. From the assembly standpoint the source program consists of a sequence of "sketch" indications of the operations to be performed. The assembly language program, in addition to representing a sequence of operations, also contains a great many specifications concerning the particular machine registers or register types to be used in performing these operations; the particular order in which the necessary sequence of operations is to be stored within the machine; the particular manner in which data is to be stored within the machine; the detailed pattern in which intermediate information is to be stored and reloaded, etc.

All of this additional material accumulates to a very considerable mass with which the assembly language programmer is necessarily concerned and which the source language programmer need not even consider. The source language programmer relies on the translator or compiler (which, from his source indications, is to produce actual machine language) to handle all these problems in a stereotyped way. That is, registers are chosen in a stereotyped pattern by a compiler, temporary loads and stores likewise; storage is laid out and instructions are chosen and ordered in some standard fashion, etc. It is certainly true that the compiler's stereotyped process may in certain cases yield a program which is longer (perhaps) and less efficient

(perhaps) than a specially designed hand coded program providing the same function. (It is to be noted, however, that a high quality code optimizer is in many cases capable of producing code that is superior and perhaps even considerably superior to hand code.) Nevertheless, for the programming of elaborate functions, stereotypy of approach is an essential requirement, since a sufficiently complex program becomes completely untransparent and impossible to complete and test in a reasonable lenght of time.

Another consideration illustrating the manner in which an ill-adapted language can actually penalize the programmer may be added to the above. Often a language (like a machine assembly language) which permits the specification of information beyond that strictly necessary to describe a particular function not only permits but requires this information. E.g., an assembly language normally not only permits detailed specification of the machine registers to be used to carry out a sequence of operations but requires that this information be specified in all cases. While the programmer using such a language gains efficiency in certain parts of the code through his ability to specify such information, he must pay the price of specifying it everywhere, that is, even in those less critical parts of code in which he would be content with a less efficient code which could be represented in a much sketchier way. Very often, especially in long codes, this price will far outweigh any advantage obtained.

It is also worth noting that the use of an ill-chosen language may also cause a substantial increase in the average coherence of the program. Thus, for instance, machine registers are normally explicit variables in an assembly language, necessarily common to all parts of an assembly-written code. This circumstance increases the strength of logical connection between all parts of the code, introducing a whole class of register content errors which do not exist at all in a source

language in which the machine registers are hidden. By handling the loading and unloading of registers in a completely stereotyped way, by providing methods for the logical isolation of subsections of a long program, and by reducing the amount of information which must be specified in the description of a given function, a well chosen programming language manages to avoid many of these difficulties.

The linguistic method in programming begins with the process of stereotypy described above but carries it further. Stereotyped information, as has been indicated, can be mechanically generated from a relatively minimal set of indications of its form. A good language assists this minimization as follows. Certain aspects of the pattern of information (or program expressing a given function) to be generated are rigorously determined by other parts of the information and can be calculated from them. The minimal expression of such a pattern therefore consists of that smallest subpattern from which all the rest can be deduced. A sophisticated language processor examines the string of information submitted to it, finds indicative subpatterns therein, and, on detecting these, supplies the omitted, contextually implied information which is required. In this way, very considerable economies of expression become possible, and it becomes feasible to program complex functions, otherwise very difficult of expression, rapidly and effectively.

The linguistic approach to the expression of complex functions or of groups of complex functions belonging to a given field and having a certain over-all similarity of form may then be described in the following way:

(a) By direct experimental hand-analysis of the functions required, arrive at a well chosen and maximally stereotyped set of elements suitable for the expression of these functions.

(b) Analyze repetitive patterns in the use of the stereotyped

elements needed and develop a convenient and relatively minimal set of indications from which the full structure of element invocations can be derived.

(c) Design a language processor or compiler capable of expanding these minimal indications into the full pattern of information describing the required function.

(d) Steps (a), (b), and (c) suitably carried out, will result in the design of an effectively adapted problem oriented language. The required function may then be expressed effectively in this language.

Additional advantages accrue from the use of an appropriately designed source language and from the mechanization of its translation into machine code. The level of indirection introduced in such translation results in a considerable reduction in the role played in any program by the detailed features of the particular machine on which the program is to run. By virtue of this fact the source-language programming process attains a degree of machine independence which, while it may still be quite imperfect, is nevertheless very high when compared to the extreme machine dependence characterizing assembly language code. By substituting one translator for another, it becomes possible in principle to carry a complex function over to a new machine without changing much in the source-language symbolic description of the function. To the extent that the general process of programming complex function attains machine independence, it gains in continuity, and is freed from the intolerable burden of constant reprogramming that would otherwise follow from rapid changes in machine technology and the repeated re-design of machines which is a consequence of these advances. Since the body of accumulated source language code can be very considerably larger than the length of the compiler required to translate all this code for a given machine, a considerable total advantage may result from adherence to the use of a source language in programming.

A given language will, as has been indicated above, be adapted more or less closely to the range of applications which played an explicit or implicit role in its formulation. A well-adapted language will permit the convenient invocation of the stereotyped elementary functional processes which have been found to be of most general use in the intended applications area. At the simplest level these processes may consist simply of stereotyped schemes for storage allocation, address calculation, register use, subprocess linkage and input-output; these are the principal features of a basic algebraic language like FORTRAN. Languages specially intended for areas requiring other basic operations may contain more sophisticated elements. Thus, list processing languages intended for the processing of data structures which develop dynamically in unpredictable ways will normally include one or another mechanism for the dynamic allocation and reallocation of storage. Such languages may include pushdown stacks, list space areas with automatic garbage collection, and other dynamically varying storage types as basic elements. Languages especially intended for string processing may provide methods for the convenient and concise specification of various processes of pattern matching and subpattern location. A language developing in the context of a particular application will normally develop those specific dictions which are most concise for the particular applications most often repeated. This principle, which has determined the growth of languages from a very ancient period, may be called the Masai principle: since cows are common and highly significant objects on Masai culture, any adjective not referring to an explicit noun is understood in Masai to refer to the implicit noun cow.

In the present monograph we shall apply the general linguistic method described above to the language translation process itself. That is, we shall elaborate a language which is convenient for the specification of language translation processes. Having

developed such a system, we shall apply it to describe the structure of a number of specific translators, including translators from algebraic source languages to machine language, from list oriented source languages to machine language, and also including various more special translators which translate from specialized condensed source languages into more general expanded source languages intended for further compilation. We shall also consider the relation of the syntactic methods used in translation of mechanical languages to those which may be employed for the syntactic analysis of natural languages like English.

2. General Structure of the Compilation Process

The process of compilation, that is, the translation and expansion of one description of a given process into another more detailed description, may be conveniently divided into five substages. Material to be translated is normally presented to a computer as an information string within which certain key characters and key words are embedded; these keys separate other character patterns which denote logical constants or variables of one or another type.

The first, rather simple, stage of the translation process, the so-called lexical pass, consists in the scanning of this input string, the detection in it of separator marks, and the recognition of elementary word identifiers. The lexical scan results in the reduction of the input string to a string of atomic word designators, representing essentially the same information, but representing it in a more condensed form which can be used more efficiently by the complex analytic steps which are to follow.

The second logical stage of the translation process is syntactic analysis. During this subprocess, the lexically

reduced input string is scanned to determine the manner in which the paragraphs and sentences of the source string are to be decomposed into their constituent clauses and phrases. The result of this phase of the translation is a considerably transformed, syntactically analyzed form of the source string in which the implicit grammatical structure has been made completely explicit by transformation of the order of the source string and/or by the addition of any necessary explicit marks, indicators or pointers. If, for instance, one imagined such a process being applied to English language source text, the result would be an output in which the subject word of the sentence was explicitly flagged as "subject", the verb word of the sentence was explicitly flagged as "verb", any adjectives modifying the subject were supplied with explicit pointers referencing the subject, the scope of clauses was indicated by explicit insertion of parentheses, etc. Syntactic information can in principle be represented in a variety of equivalent ways. It can, as indicated above, be represented by addition to the source string of parentheses delimiting the various subclauses of the sentences and of marks indicating the type of each subclause. Equivalently we can represent the information developed by a syntactic analyzer using a graph or tree in which the words constituting the original sentence appear as twigs and in which every node represents some specific clause type. Such representations will be familiar to many readers from elementary courses in English grammar; they are, of course, logically equivalent to a parenthesized marking of the source sentence in the style described above.

On completion of the explicit marking of the constituent subclauses of all sentences and of the types of these subclauses, the syntactic analysis phase of the translation process comes to an end. The third phase of the translation process is the explicit generation, from the fully marked tree structures produced by syntactic analysis, of target code. In principle

this process involves an ordered traversal of the analysis tree, and the application at each node traversed of an appropriate strictly local rule concerning the code to be generated when a node of that type is encountered, followed by an iterative stepping to the next node to be treated. The point to be noted here is that each of these generation steps is strictly local to an individual node; the precise point of the input string→tree transformation performed in phase two of compilation by the syntactic analyzer is the development of all the pointers necessary to reduce the information implicitly and globally indicated in the original source string to an explicit and strictly local form. The result of the third or code generation phase of translation is an output string constituting the target code version of the original source program. This target code may be either an assembly language code, a macro code suitable for relatively straightforward expansion into assembly language code, or, alternately, may be code written in the input language of yet another translator.

The translation process, carried through to the end of the above third stage, can in principle be completed by one simple additional step: assembly, i.e. the detailed laying-out of instructions and data and the explicit calculation of numerical addresses. However a fourth major process is often interpolated after code generation and before final assembly. This fourth phase of translation, which forms a conspicuous part of the more sophisticated translators, is an optimization phase in which the algorithm expressed by the compiled text is transformed via a series of equivalent algorithms in order to eliminate redundant calculations, to condense multiple steps into single steps, to replace slow processes by faster processes, to move instructions from frequently traversed to less frequently traversed code paths where possible, etc. Only rather elaborate optimizers attempt to perform all of the reductions described above; a more primitive optimizer may merely suppress unnecessary

loads and stores and eliminate redundant instructions on a local basis. A good optimizer is on the other hand a very extensive mechanized representation of the coding tricks found in assembly-language coding experience to be valuable. Coding tricks which are relatively stereotyped can in this way be supplied mechanically, and the quality of code improved to the level of programmer-produced assembly code or beyond.

The fifth, and final, stage of the translation process is the assembly phase. In this phase explicit calculations of the size and layout of all data and instruction blocks are performed. The formal identifiers used to represent these blocks during the earlier translation stages are explicitly converted to machine addresses. A detailed choice of registers may also be made at this stage. In a sophisticated translation system, a final class of optimizations depending closely on the machine hardware may also be made. Final rearrangement of the order of machine instructions and final specification of the detailed local patterns of register use will be carried out. The output from the assembly phase can be executable machine code which a simple loader program can read into a computer for immediate execution.

Figure 1 diagrams the compilation process which has been outlined above.

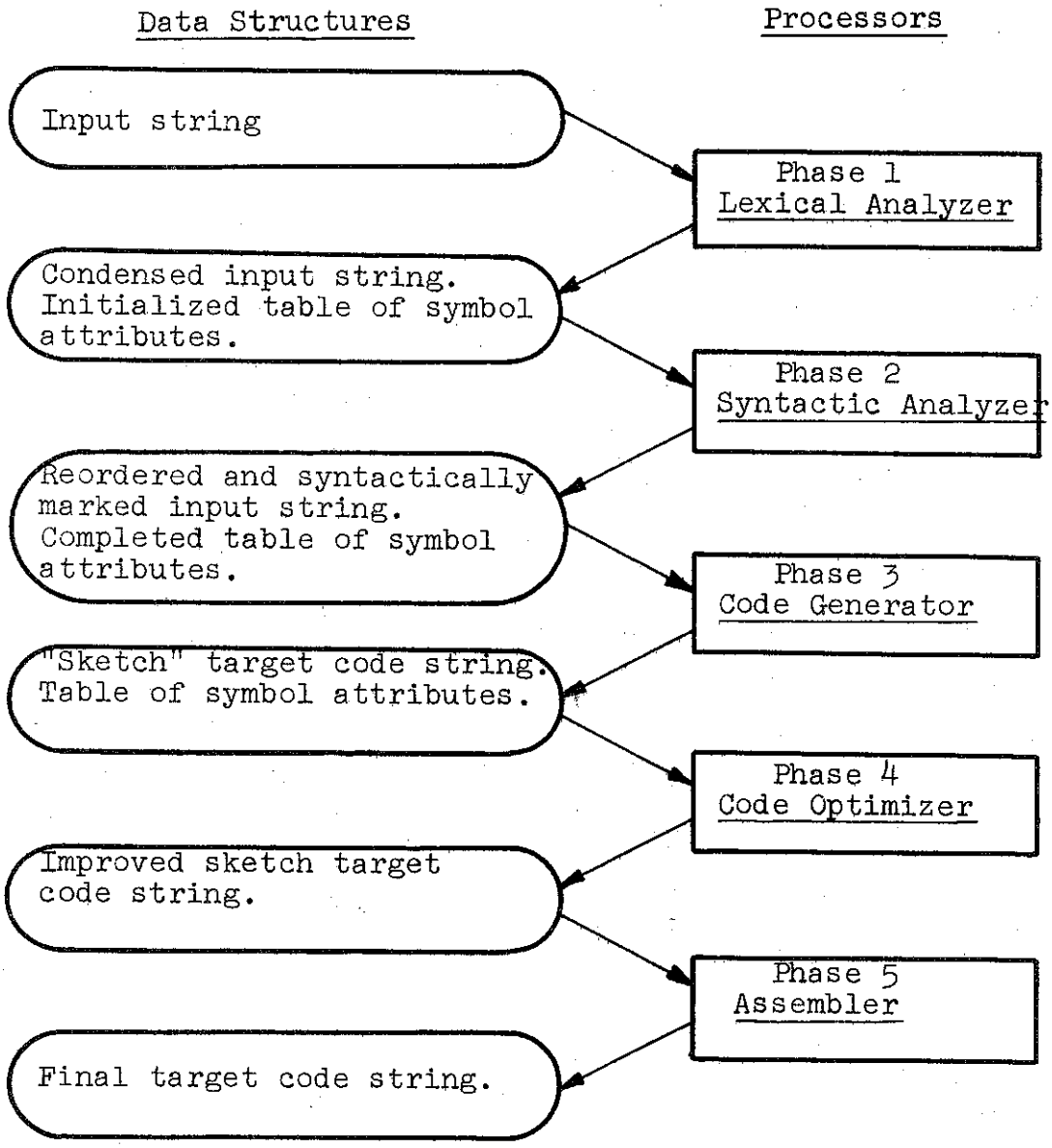


Figure 1. Schematic Flow of Translation Process.

The above outline of the compilation process is somewhat over-schematic in the distinctions that it makes between the successive stages of the translation process. Often various of the phases which we have depicted as separate are combined into a single integrated subprocess of a translator. This is sometimes done for purposes of efficiency and sometimes for purposes of convenience.

In certain cases the specific conventions of the language to be translated require the union of several translation phases. Thus, for example, the fact that standard FORTRAN does not use blanks as separators, so that the string CALLAB(I,J) = K is a valid assignment statement, while the lexically related string CALLAB(I,J) is a valid call statement in which different inter-word breaks are to be made, requires that the lexical and the syntactic scans in FORTRAN be combined.

It is also very common to combine the syntactic and the code generation pass, which we have depicted above as separate processes, into a single process. This is particularly appropriate in that the syntactic analysis process produces the information required for code generation in quite convenient form; it is therefore more efficient and no more difficult to accomplish code generation at once rather than to store syntax trees and generate code at a later point in time. Indeed, in the compilers described in this monograph, we will normally combine syntactic analysis and code generation, developing the metalinguistic conventions which we shall use in accordance with this intent. Therefore, rather than speaking of a syntactic analysis process succeeded by a code generation process, we shall come to speak of a single main subprocess accomplishing both steps.

It is not even uncommon for a compiler's code generation procedures and the optimization process which it uses to be combined into a single iterated algorithm. Such combination often results from the judgment

that the optimization process can make effective use of certain information developed by the syntactic analyzer which it would be inefficient or clumsy to store. Thus, for example, a FORTRAN DO-loop is slightly harder to recognize after code generation than during syntactic analysis, and it is sometimes found convenient to bring the syntactic analyzer and the optimizer into a sufficiently close relationship to enable the former to signal the latter at the beginning and end of loops.

Sketch code generation and final assembly are often combined, and optimization may be combined with both of them. The combination of code generation and assembly is a not infrequent response to efficiency considerations. The preparation of sketch code output intended for subsequent assembly requires a reasonably elaborate organization of subfields of information into machine words or into BCD patterns. At the beginning of the assembly process these words or patterns must be picked apart once more. By combining code generation with assembly, one is able to avoid two intermediate processes, thereby gaining in translation speed, perhaps even substantially.

The translation process as we have outlined it is sometimes elided in various useful ways. Instead of generating machine code at all, it is possible to generate sketch code in appropriate form and to end the translation process proper with this sketch code. This code may then be executed interpretively rather than directly. Such an alternative may for instance be appealing for codes that are to be executed infrequently but whose size is a consideration. In such circumstances, the sketch code form can be specially chosen to permit highly dense storage, and the resulting total program, consisting of sketch code plus interpreter, may be considerably smaller in size than fully expanded machine code. It is also possible to adapt the translator not to produce any code other than the source code at all, but to combine the syntax analysis process directly with the interpretation process. This option simply requires replacement

of the code generation subroutines of a syntax analysis-code generation processor by corresponding routines for the direct execution of the instructions which otherwise would be generated. Direct interpretation of this kind, as well as indirect interpretation in the style described just above, are sometimes particularly convenient in situations where code is under debugging and only very short periods of execution are expected. In such a situation the fact that the original source form of the code has not been totally digested means that incremental and even dynamic-on line changes to the code can be accomplished very readily. Moreover, an interpretation process which keeps the symbolic source at hand can provide a variety of useful debugging services, such as traces, traps, etc., in particularly convenient form, i.e. provide printed diagnostics which refer directly to the symbols of the original input. Of course, a suitably designed machine code compiler could also provide the same services.

The form of the sketch code to be generated by the main translation subprocess is worth separate consideration. In certain cases this sketch code is quite close to the final machine code, and only differs from final machine code in that addresses (and perhaps registers also) are designated symbolically rather than numerically, the numerical determination of these quantities being left for the final assembly step. In other cases, the especially when a fairly elaborate optimization phase is to follow, the generated sketch code consists not only of schematic instructions but includes additional information concerning the destination of branches, the dependence or independence of referenced data fields, the nature of invoked subroutines, and so forth. All of this additional information is intended to facilitate the optimization process by providing it with necessary information on the structure of the program to be optimized. The code produced by the optimizer may have a form directly adapted to the conventions of the machine

language for some particular machine. On the other hand, this code may consist of a sequence of macros which is relatively machine independent, and which, on submission to an appropriate macro-expander program can be transformed into executable code for any one of a variety of different machines.

The various principal phases of the translation process have particular flavors which it is worth commenting on. The lexical analysis pass is a rather simple one and is normally accomplished by a simulated finite state process. The syntactic analysis pass is highly recursive and typically makes use either of implicit or explicit push-down stacks. The code generation pass is normally straightforward and directly computational; the computations of which it consists are often combined directly with syntactic analysis. The code optimization pass, if elaborately done, is rather complex, amounting as it does to a mechanization of a miscellaneous assemblage of programming tricks. It involves two principal families of subprocesses. The first family includes those processes which analyze program flow and attempt the global elimination of redundant instructions and the global motion of instructions from frequent to less frequent code paths. The second main family of optimization processes consists of procedures which are somewhat more local in nature and amount essentially to the local combination of multiple instructions into single instructions where a machine order code set permits this.

The final assembly process is straightforward and essentially arithmetic. It uses two sets of tables, one describing the order codes of the machine for which code is being assembled and a second which is an identifier table. The identifier table is used in a straightforward way to calculate the final address of each identifier in the sketch code string produced by the code generator and to replace all occurrences of logical identifiers by corresponding machine addresses.

The design of a compiler not only involves questions concerning the algorithms by which translations are effected but also requires decisions as to how the data on which these algorithms operate is to be held. That is, one must define the data structures to be used by the compiler.

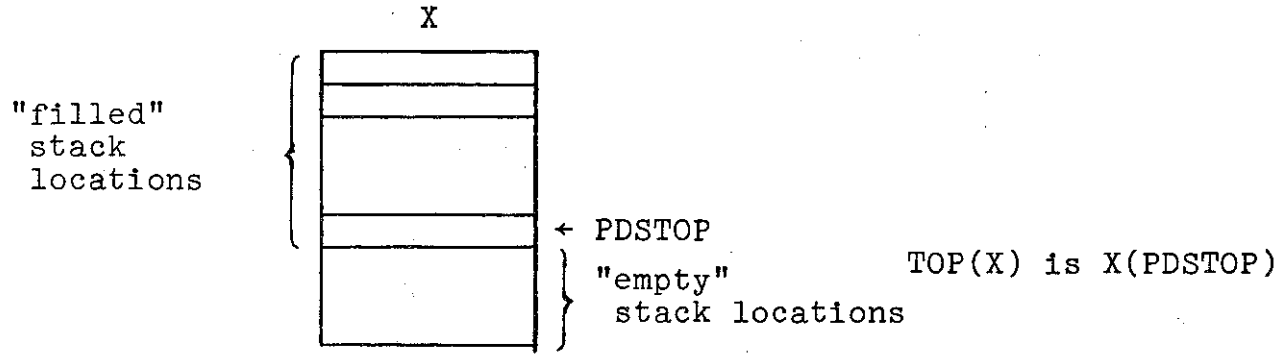
One data structure used by almost all compilers is the symbol table alluded to above. This table, whose construction begins during the lexical scan phase, is used throughout the compilation process for the global accumulation of information. As a general rule, an entry is made in the symbol table for each name found in the input string and for each auxiliary name generated by the compiler itself during the compilation process. The table entry associated with each of these names consists of a group of fields used to record the attributes of the name as these are progressively determined in the course of compilation. For example, the type of a variable (e.g., character string or arithmetic; sometimes broken down more finely, e.g., fixed or float, character string of length n, etc.) is often recorded. One may record other attributes as well. For example, one may record whether or not the name is used as a subroutine argument or whether or not it identifies a global (COMMON) variable. In some compiler designs it also turns out to be convenient to admit additional symbol table entries corresponding to elementary operations (e.g., A+B). This technique, which we shall describe more fully in a later chapter, allows one to treat variables and elementary operations in a uniform way and has certain other technical advantages.

Although almost every compiler uses a symbol table, the way in which this table is structured varies considerably. For example, since the size of the symbol table required for the compilation of one program may be quite different from that required for a different program, one must decide how to allocate storage for the symbol table. Should one allocate a fixed amount of storage at the beginning of the compilation, or should one "acquire" storage from a common storage pool as entries are made in the table?

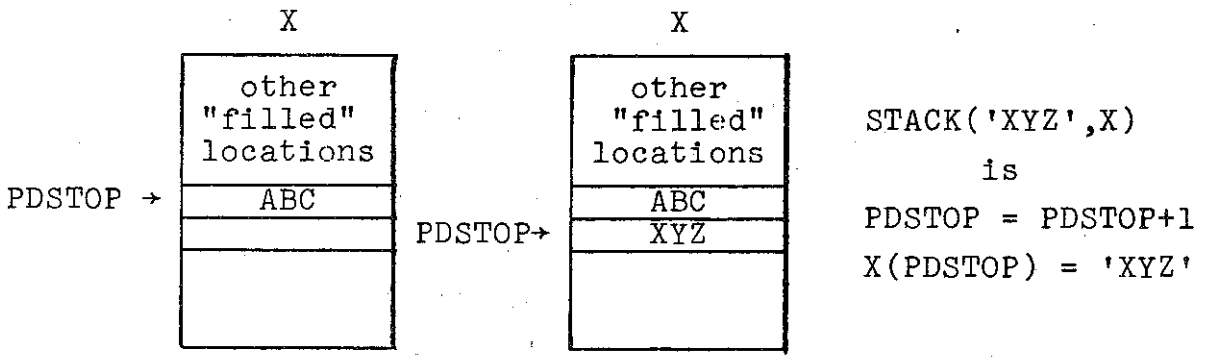
Moreover, the information fragments associated with one name may be quite different from those associated with another name. For example, if a name is used as a subscripted variable, one must record both the dimensionality and the bounds associated with that name; on the other hand, if the name represents a simple identifier this information is not needed. This raises the question of whether the entries in the table should be uniform of whether the size and information content of an entry should vary. The structure of the language to be translated will also affect decisions regarding the symbol table structure. For example, many languages permit a single name to be used to represent different variables at different points in the same program. This fact must be suitably reflected in the symbol table.

Generally speaking, questions of this sort have no unique best answers. The form of symbol tables adopted will depend partly on other design criteria which the compiler is to meet and partly on the environment in which it must operate (characterized, e.g., by memory size). For example, a compiler that attempts to produce especially good code may require different information structures than a compiler for which fast translation time is the most important goal. For reasons of this sort, one cannot specify a unique general structure for a symbol table, although common techniques can (and will) be elucidated.

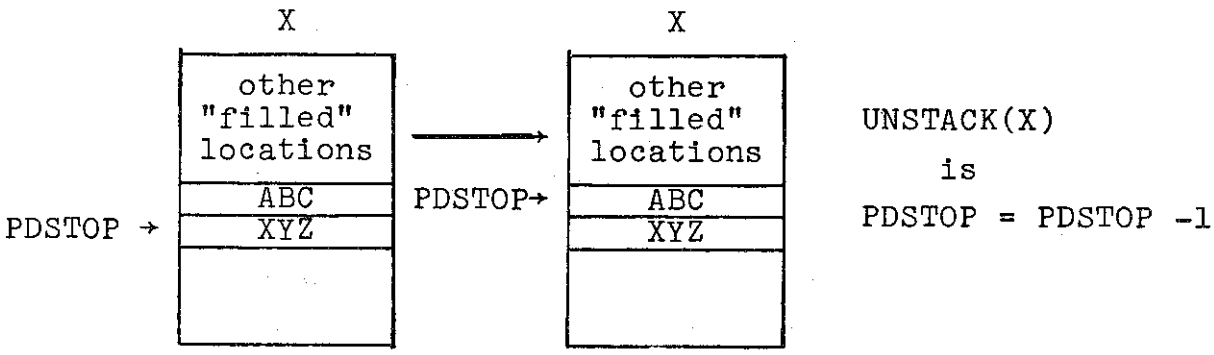
The push-down stack is a type of data structure that is frequently used by compilers. Such a stack is characterized by the fact that an entry is made only at the "top" of the stack. Characteristically, in using a push-down stack, this top element is accessed more frequently than other elements of the stack. When the top element of a stack is removed, the element immediately beneath it becomes accessible. Stacks may be implemented using arrays, as shown in Figure 1.



a. Accessing the TOP of stack X.



b. Stacking an element on stack X.



c. Unstacking an element from stack X.

Figure 1. Typical use of a pushdown stack.

For example, two principal data structures used by a compiler-compiler described later in this monograph are pushdown stacks. These are the recursion control stack and the argument stack. The recursion control stack is used for sequencing the recursive recognition procedures and contains machine addresses (or their logical equivalents), indicating the point from which a recognition routine was called. In the process of analyzing a syntactic fragment, a recognition routine may require the services of another such routine. When this occurs, the current machine locations is saved on the recursion control stack and a transfer to the required subroutine is made. When this subroutine is finished, it retrieves the address A to which it is to return from the control stack, removes the top entry (thus making the next lower entry available) and transfers to the address A. Thus the top of the control stack always indicates the address to which the current routine is to return when its task is accomplished.

The argument stack is used for the transmission of information from one fragmentary subprocess of syntactic anslysis to another and also for the transmission of information between the syntactic analysis processes and interspersed code generation routines. Its use will be described in more detail later in the present work.

It is seen from the above that the information which must be supplied to define the syntactic analysis pass consists of two principal subportions: the syntax specification describing the phrase structure of the source language to be translated, and the symbol table manipulating routines which are invoked at appropriate points during the analysis. A source language to be translated may therefore be complex either because the syntax of the language is complex in and of itself, or because the set of attributes being accumulated in the symbol table and the routines for manipulating these attributes are complex. For an algebraic language like FORTRAN, possessing a fairly straightforward syntax, but admitting a rather large family of variable types and corresponding declarations, the symbol table manipulating routines will normally amount to a mass of code at least equal in length to the syntax specifications needed.

CHAPTER 2. THE PRINCIPAL SUBPROCESS.

In this section we shall describe methods used in analyzing sentences of a programming language and ultimately for generating code for these analyzed sentences. Our aim will be to develop a language in which both of these programming tasks can be performed comfortably. We will approach this goal through a series of steps, beginning with a well-known language that is purely descriptive; i.e., that describes, in a concise and elegant way, all the legal sentences of the language. We shall show how a description written in this language relates, in a natural way, to an algorithm for recognizing these sentences in an input string. By a gradual (and, hopefully, well-motivated) process of augmentation and modification we shall turn this language into one closer to that ultimately required -- a language in which it is possible to express both sentence analysis and code generation in a natural way.

1. The Backus metalanguage.

For the effective description of language structure we require a specialized language-describing language, or a metalanguage. (On the other hand, since a reasonably designed metalanguage can describe itself, no meta-metalanguage will be required.) We shall in the course of this section and the next have occasion to consider a variety of metalanguages, each with its own particular flavor. We shall begin our discussion with the elegant metalanguage originated by John Backus of the IBM Corporation, and called the Backus metalanguage or the Backus normal form in his honor. We shall at first approach this metalanguage somewhat informally, considering it to consist of a set of rules for the iterative construction of the family of all grammatical sentences of a specified language. Each rule or definition of the Backus metalanguage consists of a left hand portion, the definiens, of an equal sign separating the left and the right hand portion of the rule, and of a right hand portion: the definer. The definiens always consists of the name of the syntactic type being defined enclosed conventionally in pointed brackets as exemplified by:

(1) <story> .

The form allowed for a definer is somewhat more complicated. Most simply, a definer might consist of a sequentially concatenated string of syntactic type names, each syntactic type name being enclosed in pointed brackets. Thus, for example, a definer might have the following form.

(2) <beginning> <middle> <end> .

The definition with the first displayed formula as definiens and the second displayed formula as definer would then be

(3) <story> = <beginning> <middle> <end> .

Regarded as a rule for the production of the syntactic element "story" this states that a story may be constructed in any manner by the concatenation of a "beginning", a "middle", and an "end". Of course, "beginning", "middle", "end" are themselves syntactic types requiring further definition. The definition of "beginning" might for instance be

(4) <beginning> = <title> <first subheading>
<introductory paragraph> .

Similarly, "middle" and "end" would require definitions, as would the syntactic types constituting the definer of the definiens "beginning", namely "title", "first subheading", and "introductory paragraph".

It is evident that the procedure as so far outlined can only lead from definitions to new definitions without end. In order to enable the constructions specified by the Backus metalanguage to lead not only from syntactic types to other syntactic types but also from syntactic types to final sentences, one includes atomic symbol strings in the Backus metalanguage. These strings occur on the right hand or definer side of a definition and, by convention, may be concatenated into a definer at any point. They are distinguished from the syntactic types occurring in a definer simply by not being included in pointed brackets. Thus for instance we might include in our collection of definitions some such definition as the following:

(5) <title> = HAMLET

for the syntactic type "title". This hypothetical rule states that the only possible replacement for "title" is the literal word HAMLET. Of course, a syntactic type which can only represent a single literal word can as well be omitted. That is, the logical effect of (4) and (5) could equivalently be

conveyed by the following alternative definition of the syntactic type "beginning".

(6) <beginning> = HAMLET <first subheading>
<introductory paragraph> .

A more realistic instance of the above sort of construction is given by the following definition.

(7) <ifstatement> = IF <expression> <relationop> <expression>
THEN <line number> ,

taken from an actual set of language definitions which we shall consider in some detail later, namely the metalinguistic definitions for the algebraic language known as BASIC. Definition (7) states that an <ifstatement> in the BASIC language consists of the literal word IF, followed by any string that can be substituted directly or indirectly for the syntactic type <expression>, followed by any string that may be similarly substituted for the syntactic type <relationop>, followed by yet another string which is substitutable for <expression>, followed by the literal word THEN, followed finally by any string which may be substituted for the syntactic type <line number>.

The description which we have given of the Backus metalanguage is still incomplete in one essential regard. The metalanguage permits definers which rather than describing only a single prescribed form for a definiens, allow any one of a number of options. The options allowed are conventionally indicated within the sequence of symbols constituting the definer by subsequences separated by vertical bars. Thus, for example, if the syntactic type "beginning" were defined by the equation

(8) <beginning> = HAMLET <first subheading> <introductory paragraph> | MACBETH <introductory paragraph> ,

then a "beginning" could be constructed either by following the

literal word "HAMLET" by some string representing a "first subheading" and some other string constituting an "introductory paragraph", or alternatively by following the literal word MACBETH with any string constituting an "introductory paragraph". The number of alternatives which may be included in a Backus normal form definer is unlimited; as indicated above, successive alternatives are to be separated by vertical bars.

Given a set of metalinguistic definitions of the form described above, one may choose a distinguished syntactic type, called the basic or root definiens, or root type. A set of definitions, together with a designated root type, is called a context free grammar or a (simple) phrase structure grammar; the set of literal strings that can be generated from the root type is called a phrase structure language. The set of definitions describing such a language is sometimes called the Backus Normal Form or BNF description of the language. Such a set of definitions formally determines a collection of generated sentences in the following way. Let Σ_1 be the one-element set consisting of the basic definiens. If Σ is any set of strings, let $\tau(\Sigma)$ consist of the set of strings in Σ together with every string which may be formed from the strings in Σ by replacing a syntactic type by one of its alternative definitions. Clearly, $\tau(\Sigma) \supseteq \Sigma$. Inductively write

$$\Sigma_2 = \tau(\Sigma_1) \; ; \quad \Sigma_3 = \tau(\Sigma_2) \; ; \quad \text{etc.}$$

Let Σ_∞ be the union of all the sets Σ_n. Let Ω be the collection of all strings in Σ which contain no remaining syntactic types but only literal words. The family Ω of sequences of literal words generated in this way constitutes the family of sentences of the language defined by the original finite set of Backus definitions. Note that the set Ω will, in general, be infinite. This indeed is part of the power of the Backus notation; it is a method of describing an infinity of sentences in a finite way.

The illustrative definitions above may be included in a complete, though miniature, grammar as follows:

<story> = <beginning> <middle> <end>
<beginning> = HAMLET <first subheading> <introductory paragraph> | MACBETH <introductory paragraph>
<introductory paragraph> = ONCE UPON A TIME | ONCE LONG AGO
<first subheading> = CHAPTER 1 | PREFACE
<middle> = BOY MET GIRL | GIRL MET BOY
<end> = AND LIVED HAPPILY EVER AFTER

The reader may review and test his understanding of the Backus metalanguage by generating various <story>'s belonging to the phrase structure grammar described above. It is to be noted that even though all of the generated strings are entitled either HAMLET or MACBETH their structures are primitive and quite unshakespearian. A more interesting language would of course require a less rudimentary grammar for its definition.

While the Backus metalanguage, as described above, already constitutes a useful device for the description of languages, it is convenient, especially for the description of mechanical languages, to add two more features to it. The first addition is motivated by the following consideration. Often, in mechanical languages, repetitive lists are used, as, for example, in the FORTRAN declaration statement

(9) INTEGER A,B,C,D,E,F

It is, in fact, possible to define repetitive lists of the form occurring in (9) directly in terms of the Backus metalanguage. The required metalinguistic construction is merely

(10) <list> = <term> | <term> <cterm>
<cterm> = , <list>

Overfrequent use of such a recursive form is sometimes clumsy. To provide a less clumsy metalinguistic expression applicable

to the definition of repetitive lists we can introduce the convention that if the name of a syntactic type included in pointed brackets is suffixed with an asterisk then repetition of the syntactic type any number of times, from zero to infinity, is intended. With this convention we can rewrite the syntactic definition (10) as

(11)
$$\langle list \rangle = \langle term \rangle \ \langle cterm* \rangle$$
$$\langle cterm \rangle = \ ,\langle term \rangle$$

The definitions (11) may be read informally as follows: a list consists of a term followed by an arbitrary number, perhaps zero, of "cterms"; a "cterm" consists of an occurrence of the literal "comma", followed by a "term".

A second addition to the Backus metalanguage has the following motivation. In some cases, it is more efficient, as well as more convenient, to treat a few of the simpler elements of a language lexically, rather than syntactically. The details of the lexical scan process required to accomplish this are given in Chapter 3 below; a theoretical discussion indicating the point at which the boundary between the lexical and the syntactic may most appropriately be drawn will be found in Chapter 4. When such a lexical scan has been applied, some of the character substrings in the source text are conglomerated into "words" which it is appropriate to treat in syntactic analysis as indivisible atoms. Each of these lexical atoms will be designated by the lexical scanner as having some particular type. E.g., in standard FORTRAN the lexical types: integer, name, octal constant, Hollerith constant, etc. would appear. At certain points in the syntactic description of a sentence we would then wish to indicate that an atom of a given type (rather than some fixed literal) is required. For this purpose we shall merely indicate the type

of a required atom with a prefixed asterisk as follows:

(12) <*name>

Adopting this convention, we are able to write syntactic equations like

(13)
<subroutine statement> = SUBRØUTINE <*name>
(<*name> <namec*>)
<namec> = , <*name>

The syntactic definitions (13) are to be read as follows: a "subroutine statement" consists of the literal word SUBROUTINE, followed by an atomic name, followed by a literal left parenthesis, followed by another atomic name, followed by an arbitrary number of repetitions of the syntactic type "namec", followed by a literal right parenthesis. The syntactic type "namec" consists of a literal comma followed by an atomic name.

We conclude the present section by noting that the Backus metalanguage, like any other reasonable metalanguage, may be used to define itself. In doing so however we must avoid confusion between the literal symbols occurring in the language being defined and the markers used as metalinguistic separators in the definition of this language. The only symbols having a special meaning in the Backus metalanguage, as we have described it, are the left-hand pointed bracket, the right-hand pointed bracket, the vertical bar, the asterisk, and the equal sign. All ambiguity can be avoided merely by substituting other symbols for these symbols. The formulae below then describe the Backus metalanguage in terms of itself. In writing them we have substituted round brackets for pointed brackets, and a slash mark for a vertical bar. As remarked above, this enables us to regard the conventional BNF separators as literals rather than as syntactic separators. Since the asterisk and the equal sign always occur in fixed locations relative to the other separators, they can cause no confusion and we need not use another symbol for either of them.

(wholedescription) = (defingroup*)
(defingroup) = (definition)
(definition) = <(*name)> = (definers)
(definers) = (definer) / (definer) (xtradefs*)
(xtradefs) = | (definer)
(definer) = (parts*)
(parts) = (*name) / <(*name)> / <*(*name)>/<(*name)*>

The reader should examine this set of definitions closely and note that in its brief compass it specifies precisely the Backus metalanguage described informally in the preceding paragraphs.

The sequence of generative actions which lead from the root type of a grammar to a sentence valid in the grammar may be represented in a convenient and illuminating graphical form by the use of diagrammed parse trees. Such a tree is a collection of nodes, represented by points, and of branches, represented by lines connecting (higher) nodes to (lower) nodes. Each application during the generation of a given sentence of a Backus definition to replace an intermediate symbol α by a string of symbols $\beta_1 \ldots \beta_n$ is represented in the parse tree by a node which is marked with the symbol α and from which there depend n branches to nodes marked with the symbol $\beta_1, \ldots, \beta_n$ respectively. Thus, for example, the parse tree of a particular sentence belonging to the miniature grammar described just above is shown in Figure 1.

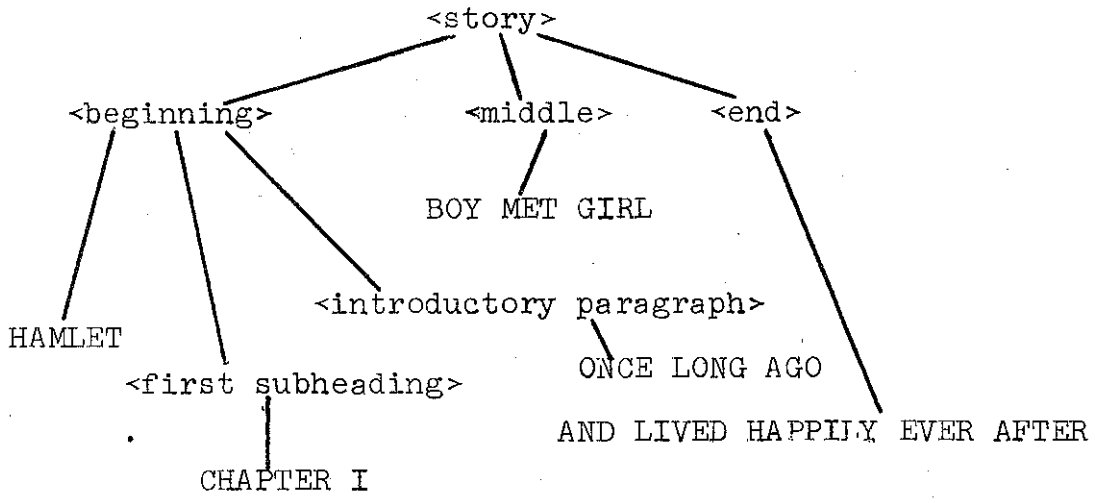


Figure 1. An illustrative parse tree.

A language is called ambiguous if there exists a sentence of it which has more than one parse tree, unambiguous otherwise; almost all of the programming languages with which we shall deal are unambiguous. The general aim of syntactic analysis is the explicit construction of the parse tree of input sentences; this forms a necessary part of compilation since the sequence of machine operations corresponding to a given source-language statement is represented explicitly in the structure of the sentence's parse tree, but only implicitly in the sentence itself. The parse tree of a sentence may be represented explicitly, and related explicitly to the grammar defining it, as follows. Let a Backus grammar with definitions

$$(14) \qquad a = b_1 b_2 \ldots b_n \mid c_1 c_2 \ldots c_m \mid \ldots$$

be given. Modify each definition in the following way:

i) Introduce, for each intermediate symbol a of the grammar, an otherwise unused terminal symbol, A.

ii) Modify each definer of the intermediate symbol a by prefixing to it a left parenthesis and affixing to it the symbol A. followed by a right parenthesis.

Applying the above definition to the definition (14), for example, we obtain the modified definition

$$(15) \qquad a = (\beta_1 \beta_2 \ldots \beta_n \; A \,) \mid (c_1 c_2 \ldots c_m \; A \,) \mid \ldots$$

The grammar (15) may well be called the explicitly tagged version of the grammar (14). A parse tree of a sentence according to a grammar (14), which exhibits the use of some particular sequence of Backus definitions, corresponds precisely to the unique sentence produced according to the modified grammar (15) by the use of a precisely matching sequence of definitions. The parenthesis and node-type marks present in the latter sentence show the node structure

of a parse tree in an explicit way. Thus, for example, we might by the use of the "explicit tagging" technique indicated above represent the parse tree shown in Figure 1 by the explicitly marked sentence

(16) (('HAMLET'('CHAPTER 1' firstsubheading)
('ONCE LONG AGO' introductory paragraph)beginning)
('BOY MET GIRL' middle)
('AND LIVED HAPPILY EVER AFTER' end)story)

From this point of view, we may regard any parser as aiming to supply the parenthesizing and node-type marks present in a sentence generated according to the explicitly tagged version of the grammar (14), but present only implicitly in the corresponding sentence generated according to the grammar (14) itself. This makes it plain that a parser is useful because it enables us to write programs in a relatively brief and convenient form in which a large number of auxiliary marks are left implicit, and to recover these marks subsequently for explicit use in the generation of machine instructions.

2. Anticipatory Survey of the Parsing Algorithms to be Discussed in What Follows.

It is the intent of the present section to prepare the reader for the discussion of parsing algorithms which is to follow (principally in the remainder of the present chapter and in Chapter IV) by surveying the parsing methods which are to be studied in what follows. As we have seen, parsing is the process of making explicit the implicit structure of a sentence. The methods which can be used to parse may be classified according to various of their salient properties. One basic distinction is that separating the top-down or goal-directed methods from the bottom-up or data-directed methods.

In a top-down method (methods of this class will be more systematically and carefully described in later sections of the present chapter), the basic strategy is to start from the root symbol <a> and attempt to generate a sentence that matches the input sentence. Since, as we noted above, a grammar in general describes an infinity of sentences, we must include in this generation process checks on the plausibility of the partially formed string matching the input string. We accomplish this approximately as follows.

The root symbol <a> is defined by a set of alternatives

$$\langle a \rangle \;=\; \ldots \mid b_1 b_2 \ldots b_n \mid \ldots \;\; .$$

We begin by choosing one of these alternatives, say $b_1 b_2 \ldots b_n$, and forming the string

$$b_1 b_2 \ldots b_n \; .$$

Each b_i represents either a terminal symbol or a syntactic type. If b_1 is a terminal symbol, it is matched against the first symbol on the input string. If there is a match, an input string pointer is incremented and attention turns to b_2; otherwise another alternative is chosen for <a> and the process restarted. If b_1 represents a syntactic type, it is replaced in the string by one of its alternatives, say $e_1 \ldots e_r$, resulting in a string

$$e_1 \ldots e_r b_2 \ldots b_n \; .$$

The expansion and comparison process is then applied to e_1, etc.

In general one will be dealing with a string

$$c_1 c_2 \cdots c_k c_{k+1} \cdots c_f$$

in which $c_1, c_2, \ldots, c_k$ are terminal symbols that have been successfully matched to the first k symbols of the input. The elements $c_{k+1}, \ldots, c_f$ remain to be examined. If c_{k+1} is a syntactic type, we replace it by one of its alternatives. If c_{k+1} is a terminal symbol, it is checked against the (k+1)st input symbol; if it matches, attention is turned to c_{k+2}. If not, it is clear that the altnerative of which c_{k+1} is a part was an incorrect choice. Therefore, that whole alternative (and possibly additional alternatives in which it was contained) must be replaced with another and the input pointer backed up accordingly. This procedure continues until either a string is produced that exactly matches the input or until the possible alternatives for generation are exhausted, in which case the input is gramatically incorrect.

In a bottom-up method we survey the input string w and attempt (using any one of a number of alternative algorithms, cf. Chapter IV) to find an intermediate symbol <f> of the grammar and a substring w' of w which forms an allowed definition for this intermediate symbol. Whenever this is possible, we replace the substring w' of w by the single intermediate symbol <f>, thereby obtaining a condensed intermediate string w_1 to which the same basic step may be iteratively reapplied. Proceeding in this way we attempt to condense the given string into a single character representing the root-type of Γ. Whenever the condensation process is blocked, we must find an alternative sequence of condensations to try.

A second basic distinction among parsing methods separates the advancing schemes from the backup-oriented schemes. Advancing parsers aim to attain efficiency by avoiding all conditional decisions which may arise during a parse and which can lead after several apparently successful steps to an ultimate failure. A well-designed advancing parser will analyze an input sentence consisting of n lexical atoms in a number of steps proportional to n, the best asymptotic behavior which can be expected in view of the evident necessity to scan every symbol of an input string in parsing it. Such schemes pay for their efficiency through their inability to handle the most general grammar. Backup-oriented parsers, accepting the necessity to reverse prior parse-guesses at a later stage, can handle more general grammars, but may be less efficient in their treatment of grammars permitting an advancing parse.

We may usefully distinguish between parsing methods capable of handling only a sub-class of the class of all Backus grammars, and parsing methods capable of handling the most general Backus grammar. It should not, of course, be expected that every backup-oriented parsing scheme can handle every Backus grammar; we note in particular that the very simple top-down backup-oriented schemes presented as an introduction to parsing techniques in the next sections of the present chapter, while they are capable of handling a fairly large class of grammars, will nevertheless fail completely to handle even very simple Backus grammars if these grammars contain any one of a number of forbidden features. (We will first meet parsing algorithms capable of treating arbitrary context-free grammars in Chapter IV). A principal effort in the present chapter will be the discussion of an easy-to-use class of top-down parsers, basically advancing but capable of backup when necessary, which can handle an extensive but not completely general class of grammars.

Ease of use is of course an important feature according to which parsers may be classified. Various other aspects of a parser combine to determine its ease of use. A parsing algorithm capable of treating arbitrary grammars will often be easier to use than one which can only handle a special class of grammars. A parser capable of treating only a special class of grammars may be particularly clumsy to use if the class of grammars which can be treated is defined by conditions which are not easy to state directly in terms of those structures, visible in the terminal strings of the grammar, for which the would-be compiler writer has an intuitive familiarity. Frequently such conditions are imposed on a grammar in order that it may be transformed into some set of tables which the recognizer finds convenient to use. The precedence oriented recognizers discussed in Chapter IV employ techniques of this kind. This leads to another parameter determining ease of use, namely the extent to which the metalanguage in which the parse is expressed must be transformed to obtain the code and/or tables which constitute the running parser. Extensive metalinguistic pre-processing, especially if only a sub-class of grammars can finally be accepted, will normally imply a system somewhat harder to use than one involving a metalanguage closer in structure to the parser code which it represents. The extent of pre-transformation will also govern the degree to which a metalanguage can incorporate features which, strictly speaking, represent extensions of the basic Backus concept rather than elements possibly belonging to this concept. One of the advantages of the class of top-down schemes described in the later sections of the present chapter is that, involving relatively simple metalinguistic processing, they are easy to use and adapt easily to generalization.

Parsers may appropriately be classified according to the speed with which they are capable of analyzing sentences of a given length. Advancing schemes will analyze sentences of length n in a number of steps asymptotically proportional

to n multiplied by an efficiency factor characterizing the particular scheme and determined by the adequacy of the methods which it employs to avoid false starts and to proceed directly to the performance of required steps. The best algorithms from this point of view are undoubtedly the deterministic bottom-up schemes described in Chapter IV. We note in this connection that action of a strictly advancing bottom-up parser may, in many cases, be understood in terms of the notion of a context-dependent grammar, inverse to a given Backus grammar Γ. A context-dependent grammar or more preferably, a set of context-dependent re-writing rules is, for our purposes, a set of definitions of the form (17),

$$(17) \qquad \alpha_1 \ldots \alpha_m \beta_1 \cdots \beta_n \gamma_1 \cdots \gamma_k = \alpha_1 \cdots \alpha_m \delta \gamma_1 \cdots \gamma_k \cdot$$

and where

$$(18) \qquad \delta = \beta_1 \cdots \beta_n$$

is an alternative of some definition belonging to Γ. In writing (17), we mean to imply that the string $\beta_1 \cdots \beta_n$ when it occurs in a context in which the string $\alpha_1 \cdots \alpha_m$ appears to its left and the string $\gamma_1 \cdots \gamma_k$ appears to its right, may be replaced by the character δ. Intuitively speaking, such an inverse grammar describes a family of syntactic condensations, which if the original grammar is to be parsable by use of the inverse grammar, must be capable of condensing arbitrary sentences written according to Γ back into the root symbol of Γ. Each application of a context-dependent rule (17) inverts some particular application of a definition of Γ, and may be considered as corresponding to a given node in the parse tree of the sentence to which the rule is applied. The grammars which may be parsed by bottom-up deterministic methods are those

for which inverse grammars of the above sort exist. Parsing algorithms taking advantage of the existence of inverse grammars fall into two main classes: those for which the grammar inverse to Γ is produced from Γ itself by application of a syntactic pre-transformation, and those which require the compiler-writer to supply the inverse grammar explicitly. In Chapter IV, we shall consider deterministic parsing algorithms of both types.

Parsing algorithms may also be rated by the speed with which they can analyze sentences written according to grammars not permitting parse by an advancing scheme, or, alternatively, sentences written according to grammars with a high degree of local ambiguity. Among known parsers, the best in this regard are the "nodal span" parsers described in Chapter IV, which analyze arbitrary sentences of length n written according to unambiguous grammars in a time proportional to n^2, and analyze sentences of length n written according to arbitrary grammars in a time proportional to n^3. These speeds are very much superior to the speeds with which other parsers will analyze sentences of like length. In particular, it is to be noted that many back-up oriented parsers may require a time proportional to k^n to parse sentences of length n written according to an ambiguous grammar.

The amount of storage required by a parser is an aspect of its efficiency as important as the speed of the parser. A parser may require large amounts of storage for one of two different reasons. On the one hand, the parser itself may either consist of a large amount of code or may require rather large pre-compiled decision tables as auxiliary information during its run. On the other hand, a parser may require extensive storage space for the storage of information developing dynamically during the parse of a given sentence. Advancing parsers are generally free of the latter problem, since they normally require an amount of space for the storage of dynamically-developed information which is not more than

proportional to the length of the sentence being parsed and which in many cases will be considerably smaller than the space required to contain the full input sentence. It is to be noted, however, that certain of the advancing schemes described in Chapter IV, especially those which use "look-ahead" very extensively, may require excessively large pre-compiled tables of look-ahead information. However, for certain languages which can be parsed by an advancing scheme requiring only a very few characters of contextual information, the tables required by the best bottom-up advancing parsers are not large, so that a bottom-up parse scheme which is highly efficient both in regard to time and in regard to space can be developed. The nodal scan parsers described in Chapter IV require, even in treating the most general grammars, an amount of intermediate storage proportional only to the square of the length of the sentence being parsed. For extensive classes of grammars these same parsers require an amount of intermediate storage which is only linearly proportional to the length of the sentence being parsed. It is to be noted that, by passing from a fully compiled parser to an interpretive version of the same parser, thereby replacing fully expanded code by some specially contrived dense tabular representation of the same code, we can decrease the amount of code space required by the parser at the cost of a certain loss in running speed.

We may describe the parse algorithms to be presented in Chapters II and IV in the light of classifications described above as follows. We begin by developing a very simple top-down back-up oriented scheme. This scheme is neither efficient nor wholly general and is in fact rather inadequate; it is presented merely as an introduction to the general principles involved in top-down parsers. Then we go on to describe a basically advancing top-down scheme and its associated metalanguage. This scheme is reasonably adequate,

is easy to use, and, since the generation of a running parser involves relatively simple pre-transformations of the meta-language in which the parsers of this class are described, is easy to generalize. We shall use the "extended Backus" metalanguage corresponding to this class of parser as a standard means for the description of programming languages in subsequent chapters of the present book.

In Chapter IV we begin to consider bottom-up schemes. We start with a description of the very attractive "nodal span" scheme due to Cocke, Younger, and Early. This algorithm handles arbitrary Backus grammars, gives best-known asymptotic parse speeds for general grammars whether ambiguous or unambiguous, and is relatively moderate in its storage requirements both for code and table space and for space needed for the storage of dynamically developed information. Next we give an account of bottom-up parsers defined by the explicit specification of a grammar inverse to the Backus grammar defining the language to be parsed. Such schemes are not hard to use, and are capable of producing parsing programs quite satisfactory from the point of view of efficiency and size.

The final sections of Chapter IV are devoted to a description of advancing bottom-up parsing schemes which use bounded context methods. These schemes probably define the fastest known parsers for the class of languages which they handle. In some cases, however, the attractiveness of these parsers is marred by the large auxiliary tables which they require. It may also be noted that, since parsers of this kind can treat only a restricted class of grammars, the analysis of a given language by these methods may involve a preliminary stage of grammatical debugging during which, by modification of the grammar for the language which we wish to parse, we force the grammar to conform to the restrictions imposed by the parsing method.

We note in conclusion of the present introductory section that, as is often the case with theoretical models of procedures having a pragmatic origin, the parsing algorithms to be developed in the subsequent sections of this chapter and in Chapter IV go considerably beyond the methods directly needed for the analysis of the commonest programming languages. Almost all languages that will be met in practice can be treated either by a bottom-up precedence method, i.e., by the simplest of the methods to be discussed in Chapter IV, or by some slight variant of such a method. This parsing method is also fast; moreover, it requires little space. Thus even the simplest of the parsing algorithms to be described in what follows gives a good practical answer to the question "how to parse."

3. Sentence Analysis in a Phrase Structure Language.

The Backus metalanguage, as we have now defined it, is quite a powerful tool for the description of languages. Somewhat further on in our discussion we shall want to supplement and to modify this metalinguistic tool in various ways. However, before doing so, it is appropriate to take a first look at the principal problem in the use of a metalanguage in the construction of translators: the interpretation of metalanguages for language analysis rather than their use to describe sentence synthesis.

Consider a typical BNF definition, as e.g.

(1) <ifstatement> = IF<expression><relationop><expression>
THEN<line number> .

This statement not only describes the construction of an <ifstatement>, but, conversely, indicates the steps of analysis required to discover the constituent subportions of a source language statement, assuming this source language statement to be an <ifstatement>. These are as follows:

1. The literal word IF must be found.
   If this word is not present, then our attempt to analyze the given source language statement as an <ifstatement> fails.
2. Next we must find a collection of atomic constituents which together constitute an <expression>. If not all the constituents required are present, our attempt to analyze the given source language statement as an <ifstatement> fails.

3. Next we must find the constituents required for a <relationop>.
4. Next must follow a set of constituents which together form an <expression>.
5. Next we must find the literal word THEN.
6. Next we must find all the necessary constituents to form a <line number>.

If all of the six steps listed above are successfully completed, then our analytic construction of an <ifstatement> comes to a successful end. If any of the six steps fails, then our source statement may be a grammatical statement of some other sort, but it is certainly not an <ifstatement>.

The six steps outlined above can be represented as a sequence of invocations of just two procedures, RECOGNIZE and FIND SUBPART. These procedures will work on a source string in left to right serial order, moving a 'next word' pointer along as syntactic analysis progresses. The RECOGNIZE procedure requires only a single parameter, namely the literal word which the procedure is required to RECOGNIZE. When invoked, the RECOGNIZE procedure has only to signal whether the specified literal word is or is not present as the next atomic word to be scanned in the source string. The success-failure signal which the RECOGNIZE routine provides may conveniently be transmitted through some machine register or memory cell available to the remaining parts of an over-all syntactic analysis program. The FIND SUBPART procedure also requires a single parameter, namely an identifier defining the particular subpart which is to be found. FIND SUBPART therefore acts as a parametrized subroutine transfer, routing the syntactic analysis process to the beginning of the sequence of steps required for the analysis of whatever syntactic type is momentarily required as a constituent in some larger structure.

In terms of these two procedures the syntactic reconstruction of an <ifstatement> as described by the metalinguistic definition (1) above is as follows.

(2) RECOGNIZE (IF)
FIND SUBPART(EXPRESSION)
FIND SUBPART(RELATIONOP)
FIND SUBPART(EXPRESSION)
RECOGNIZE(THEN)
FIND SUBPART(LINE NUMBER) .

The sequence of steps (2) may be considered to represent a section of syntactic analysis in skeleton form. However, it is incomplete in a few major regards.

a) Since the algorithmic steps (2) are to be part of a larger algorithm which will have to invoke (2) as a subportion, a pair of labels, which we might agree to supply in some such conventional form as

BEGIN IFSTATEMENT

and

END

are required. The first of these labels is of course to be placed at the beginning of (2), and the second is to be placed at the end of (2).

b) In writing the algorithmic steps (2) we mean to imply that if any of the successive recognition or subpart finding substeps fails of successful completion then the whole sequence of steps is considered to fail.

To complete the specification of the analysis procedure we must therefore define the action to be taken if any of the sequence of steps (2) fails. One of two responses will be appropriate in the case of failure. If the <ifstatement> is a subpart of some more inclusive syntactic type in which the <ifstatement> is a constituent of one alternative but which allows other alternatives as well, we merely wish, on failing to find a complete <ifstatement>, to pass to the next alternative and to attempt analysis in accordance with this alternative. In this case, we must also push the pointer defining the next source word to be scanned back to the position it had at the beginning of our attempt to construct

an <ifstatement>. If, on the other hand, no syntactic alternative exists, then the failure of an attempted analysis indicates that the source sentence under examination is ungrammatical. We then wish to emit some diagnostic remark useful to the originator of the source language statement and to pass on at once to analysis of the next sentence in the source string.

When the FIND SUBPART procedure is invoked in a semantic analysis we must therefore arrange to store

(i) The address to which return is to be made on successful construction of the required subpart.

(ii) The alternative address to which we wish to proceed in the event of failure.

(iii) The current value of the word scan pointer.

A reasonable method for storing this information as it is developed dynamically, and which has the essential property of being fully recursive, is as follows: Establish a push-down stack on which the information items (i), (ii), (iii) may be stacked. Establish a POPUP routine which works in conjunction with the FIND SUBPART and RECOGNIZE procedures. This POPUP routine should transfer to one of the two stacked addresses, (i) or (ii), in the event of successful return from an invocation of FIND SUBPART, and to the other in the event of an unsuccessful return. On unsuccessful return, the POPUP routine should also restore the word scan pointer to its stacked value. At the time of return, POPUP should also remove the three stacked items which it has just used from the top of the pushdown stack.

A stack of the sort that we have described forms the basic mechanism of recursive control in a BNF syntactic analyzer. If the analysis process is to produce syntax trees as output we require another data structure and some few additional procedures. In this case the POPUP routine will also bear the responsibility for building the syntax tree.

A very simple tree-building method which can be used is as follows. On each successful return from a FIND SUBPART call a syntactic tree subsection is

to be built. When such a tree section is built, a pointer referencing it is left on the top of a so-called argument pushdown stack. The elements to be linked into the tree subsection to be built at any point are the top few elements referenced by the argument pushdown stack. That is, on successful return from a FIND SUBPART invocation (1) we conglomerate the tree sections referenced by the top few elements of the argument pushdown stack into a single larger subtree, (2) remove the reference to the conglomerated elements from the top of the pushdown stack, and (3) place a reference to the new conglomerate on top of the pushdown stack. The number of elements at the top of a pushdown stack which are to be conglomerated at any point is precisely equal to the number of syntactic subelements contained in the syntactic type which has just been successfully found. If syntax trees are the only desired output from the syntactic analysis process this single recursive tree construction procedure can be used throughout the operation of the syntactic analysis process. On unsuccessful return from a FIND SUBPART invocation, all those tree fragments which correspond to successfully found syntactic subelements of the failed syntactic type must be erased and references to them removed from the top of the argument pushdown stack.

The following example will illustrate the combined action of the syntax analysis mechanism and its auxiliary tree-building mechanism.

Consider the simple language defined by the following small set of descriptors.

(3) <ifstatement> = IF<expression><relatop> <expression> THEN <*line number>

<expression> = <*name> + <expression> | <*name> - <expression> | <*name>

<relatop> = > | < | =

The basic type in this little phrase structure grammar is, of course, <ifstatement>. Suppose that the source statement to be analyzed is

(4) IF A - B > C THEN 100 .

When syntax analysis begins the argument stack is empty, the word scan pointer is set to the first word of the source string, and no syntax trees or subtrees have been constructed. The initial state is represented diagrammatically in Figure 1.

(argument stack) (tree sections)

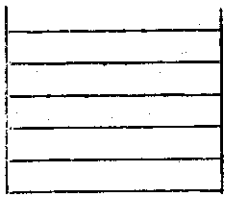
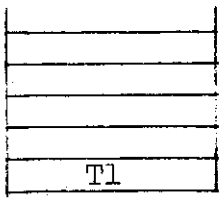


Figure 1. IF A - B > C THEN 100

The RECOGNIZE procedure is now called to recognize the word IF, does so successfully, and advances the word scan pointer by one word. Figure 2 shows the situation after the operation has taken place. We indicate the current position of the word scan pointer by writing all words preceding it in small letters; all other words are written in capitals

(argument stack) (tree sections)



T1: IF

Figure 2. if A - B > C THEN 100 .

FIND SUBPART is now invoked and required to find an <expression>. It begins with the first alternative form of the syntactic type <expression>, which requires that the lexical element <*name> be found. This is successfully found, and made into a miniature

tree section consisting of just one twig. A reference to this tree section is placed on the top of the argument pushdown stack. After this operation, the analytic situation is as shown in Figure 3.



Figure 3. if a - B > C THEN 100

Next, RECOGNIZE is called to find the sign +. It fails to do so. The first alternative definer <*name> + <expression> consequently fails, and the syntactic analysis process reverts to its status as of the beginning of the attempt to parse the source statement using this definer. After this reversion, the situation is as indicated in Figure 4.

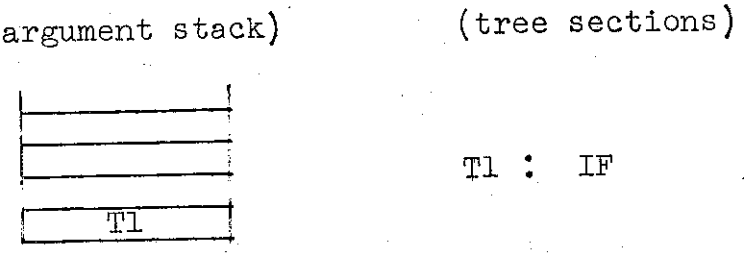


Figure 4. if A - B > C THEN 100 .

Now the second alternative version of the syntactic type <expression> is tried. This requires that the lexical type <*name> be found. It is found and made into a subtree, leading to the situation show in Figure 5.

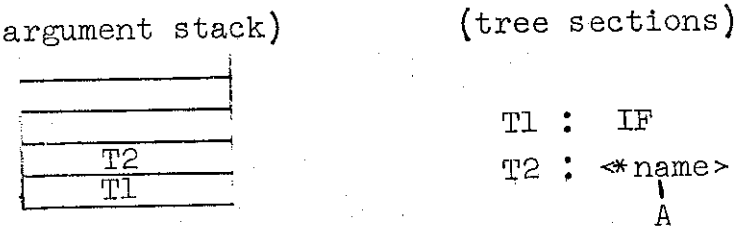


Figure 5. if a - B > C THEN 100 .

Next, RECOGNIZE is required to find the sign -. It does so, a reference to a mini-tree T3 is placed on the stack, the word scan pointer is advanced, and FIND SUBPART is again called to find an <expression>. After trying two false alternatives it will succeed, whereupon the situation shown in Figure 6 will be attained.

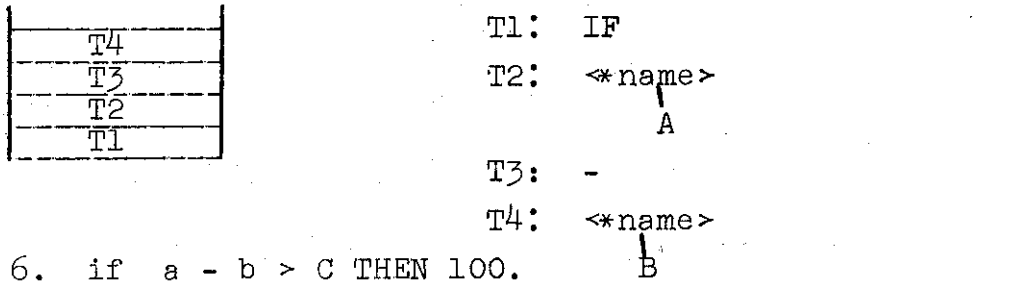


Figure 6. if a - b > C THEN 100.

Since the call to FIND SUBPART to find an <expression> is complete, the components of this expression (only T4 in this case) are removed from the argument stack and replaced by the tree for the expression; this yields the situation shown in Figure 7.

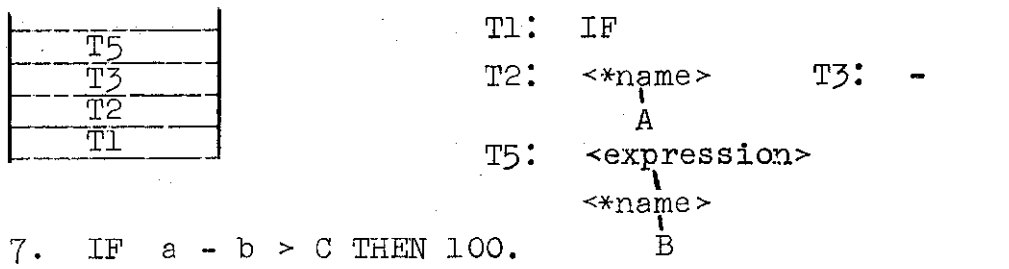


Figure 7. IF a - b > C THEN 100.

But this completes the call to FIND SUBPART for the larger <expression>; consolidation of the nodes T2, T3, T5 representing this expression leads to the situation shown in Figure 8.

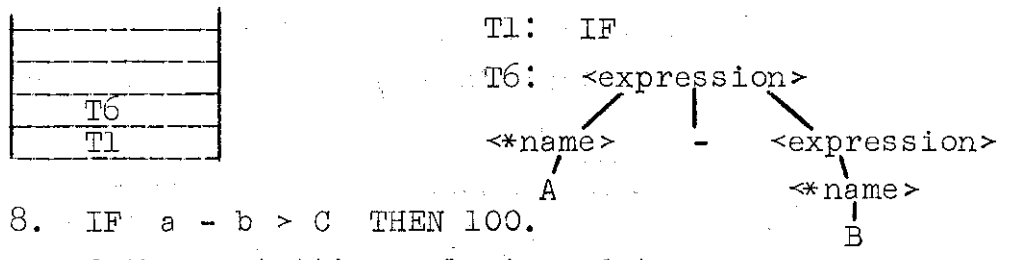


Figure 8. IF a - b > C THEN 100.

Subsequent stages of the syntactic analysis and tree-building process are shown in Figures 9 and 10.

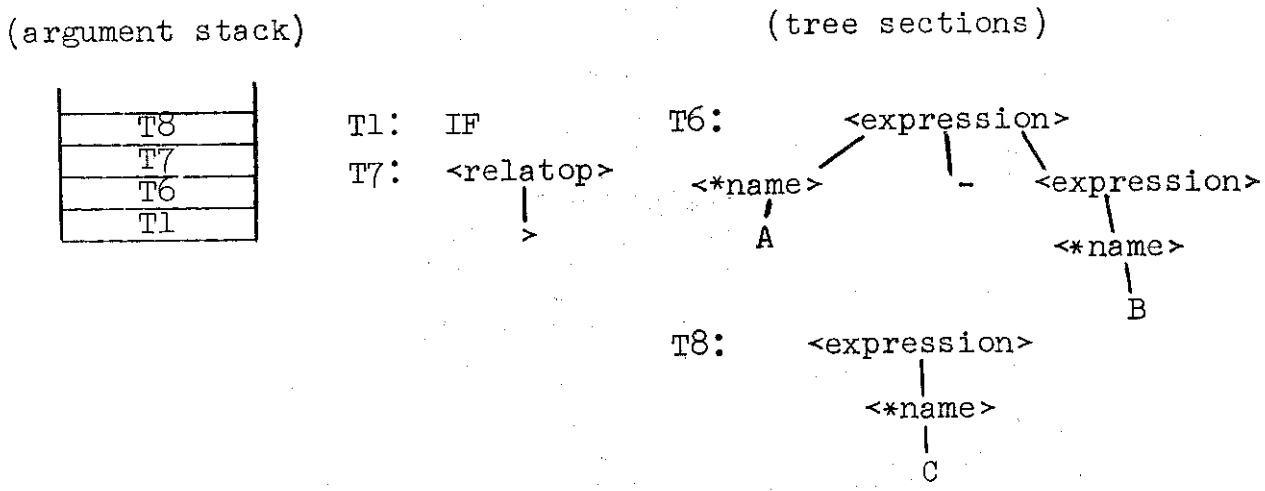


Figure 9. if a - b > c THEN 100

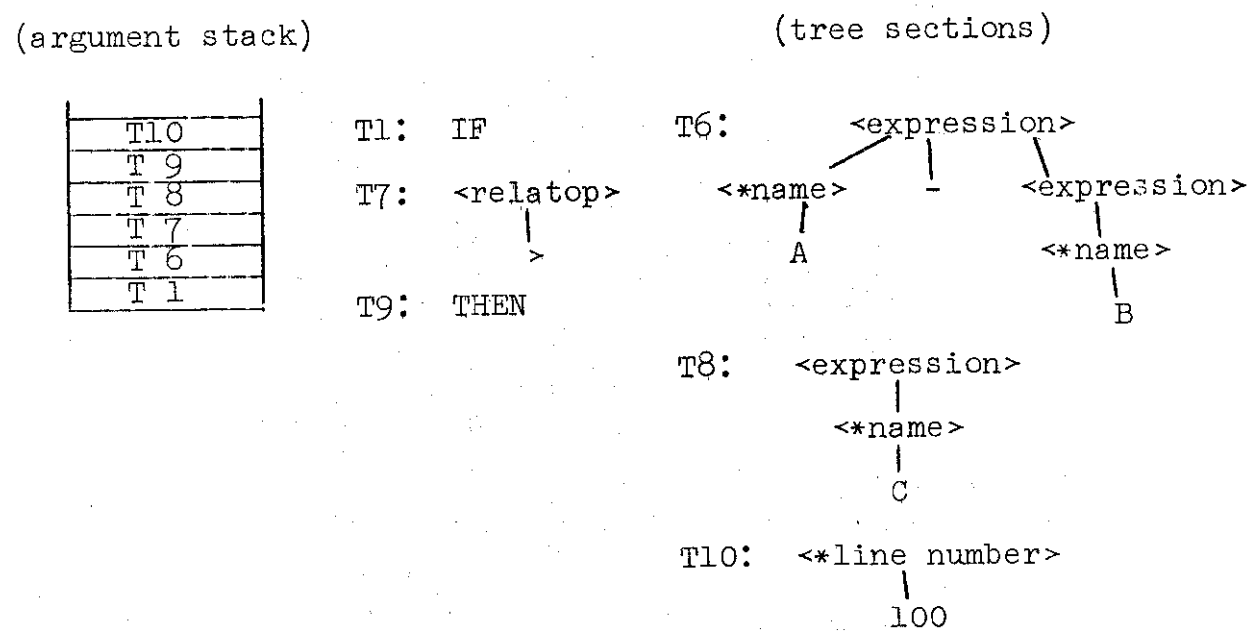


Figure 10. if a - b > c then 100

At this point, the call to FIND SUBPART for <ifstatement> has been successfully completed. Therefore the tree sections T1, T6, T7, T8, T9, T10 representing its components are consolidated into the single tree shown in Figure 11.

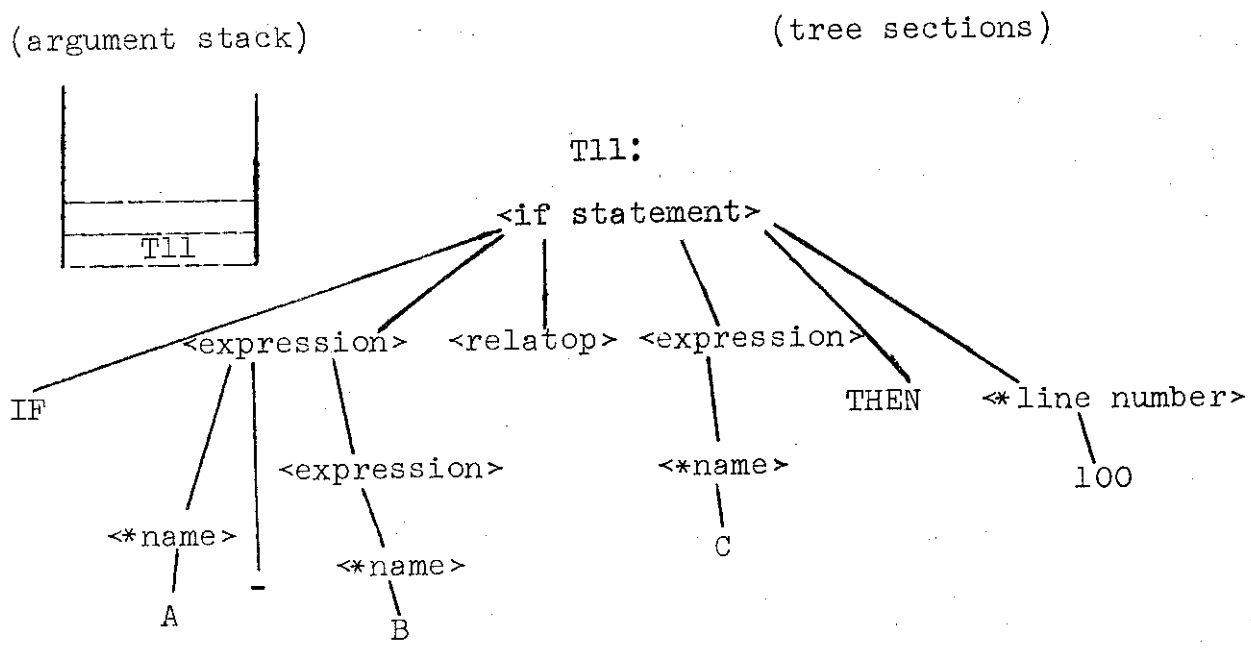
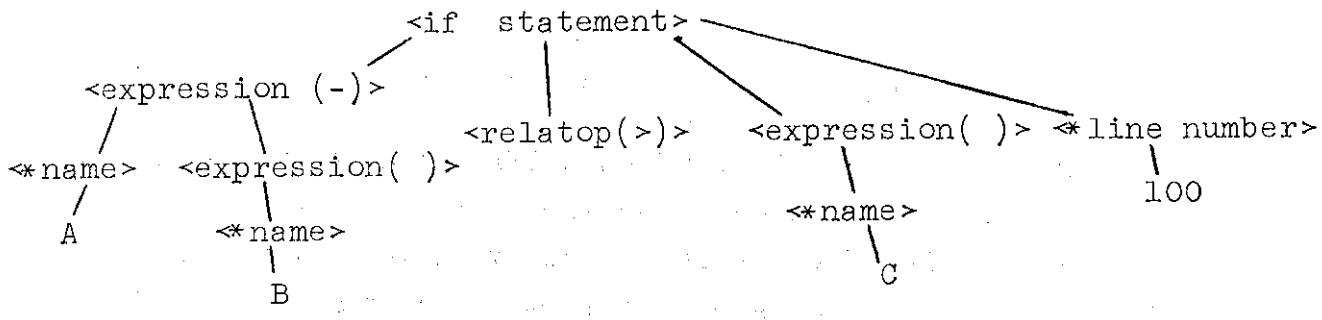


Figure 11. if a - b > c then 100

Observe that the tree section of Figure 11 is a completely explicit representation of the syntactic structure of the source statement (4).

It is not strictly necessary to represent the terminal items discovered by calls to RECOGNIZE (e.g., IF, THEN) as separate elements of the parse tree. Rather, one can indicate the presence of these items by a parameter of the syntactic type in which they occur. For example, one could rewrite the tree of Figure 11 as:



In fact, we shall use this condensed form of the tree from here on in.

A convenient and simple mechanism for building the tree sections alluded to in the preceding paragraphs is as follows. A tree section stack is established. Each item placed into this stack is either a node descriptor or a branch indicator. A node descriptor describes the type of a particular node, and in particular, specifies the number of branches which the node supports. A branch indicator will either indicate an atomic symbol, or, alternatively, correspond to some branch emerging from a node, and is merely a pointer to the node descriptor belonging to the node which this branch reaches. The branch indicators therefore merely contain those tree section stack or symbol table addresses at which node items on the one hand or atomic symbol items on the other are to be found. Branch descriptors are therefore identical with argument stack entries. To build a new node during the parse process, we have only to transfer a number of (branch descriptor) items from the argument stack to the tree section stack, cap off these items with an appropriate node descriptor, and place the address of the new node descriptor at the top of the argument stack. Note that, if we use this procedure, we can erase nodes merely by decrementing the "top of the stack" pointer of the tree section stack.

The analysis process which we have just described has previously been summarized in the algorithm (2) above. The reader will recognize, however, that (2) gives only an incomplete representation of the syntactic analysis process. Note first of all that this algorithm contains no indication of the instruction to which transfer is to be made on the failure of any of the "RECOGNIZE" or "FIND SUBPART" invocations of which the algorithm consists. In order to supply this missing feature we must, of course, allow a label to be attached any line of a syntactic analysis algorithm of the type represented in (2) above. Moreover, we must provide for the use of an additional basic syntax analysis subroutine whose purpose it will be to establish a label to which transfer

is to be made on a failure return from any "FIND SUBPART" or "RECOGNIZE" invocation. The necessary additional basic routine may be called "STACK", and can conveniently be coded as a subroutine with a single argument, this argument being the label to which transfer is to be made by "RECOGNIZE" or by "FIND SUBPART" on a failure return. It is convenient, in coding this routine, to have it stack not only this "FAILURE" address, but also to enter a pointer to the current top location of the tree section stack used in the syntactic analysis process. Of course, both of these quantities are to be placed on the top of the syntactic analysis control stack. By saving an indication of the current state of the tree section stack, we make erasure of partially constructed subtrees on the occurrence of a failure return midway through a syntactic analysis convenient. Indeed, as indicated in the preceding paragraphs, the syntactic analysis tree is progressively be built up on the tree section stack during the syntactic analysis process, and, using this technique, erasure may be accomplished simply by returning the tree section stack pointer to its earlier value.

A new "failure label" will be established by a call to the STACK subroutine at the beginning of the section of recognition code corresponding to each Backus normal form alternative. This label will, in each case, reference the section of syntactic analysis code corresponding to the next Backus alternative in sequence to be tried on the failure of the Backus alternative represented by the current algorithmic block, if any such alternative exists. On the other hand, if no Backus alternative exists, the failure label established by the STACK subroutine must (explicitly or implicitly) reference a standard BACK label, upon transfer to which the POPUP routine already described (cf. the paragraph following i), ii), and iii) above) is to be called, the success-failure flag being set to the "failure" state.

Using the conventions introduced just above, it is easy for us to write the detailed form of the algorithmic expansion of the "IF" statement syntax described by the Backus formulae (3). Such an account is given in Table I. Each of the labels in Table I corresponds to a Backus alternative. The separate subroutine invocations which follow this label correspond to the various syntactic items of which the complete Backus definer of <ifstatement> is composed. The routines invoked in Table I are RECOGNIZE and FIND SUBPART, whose significance has already been explained, the routine STACK which has the significance described immediately above, and two routines POPYES and POPNO, which are invocations of the POPUP routine in its "successful return" and its "unsuccessful return" alternatives.

In Table I we have also made use of a subroutine FIND LEXICAL, whose significance may be explained as follows. In writing the syntactic description in (3), we have explicitly indicated that certain syntactic types (in particular, the syntactic type "name") are to be treated lexically rather than syntactically. This has been indicated in terms of the "prefixed asterisk" convention established and explained in the paragraph of section 1 of the present chapter immediately preceding formula (12). The occurrence of such an atomic lexical type in a Backus descriptor implies that an elementary routine permitting essentially immediate recognition of a lexical atom, rather than a complex, possibly recursive, procedure for the recognition of a composite syntactic type, is to be used. It is this elementary lexical-atom finding routine that has been called FIND LEXICAL in the algorithm of Table I. The FIND LEXICAL procedure is assumed to consult a table of lexical atoms, established by a preceding lexical scan of the raw input string, which scan, in the manner already explained, and using the techniques to be described in more detail in Chapter 3, reduces the raw input string to a condensed string of lexical atom pointers on the one hand, and to a set of symbol table entries describing the various lexical atoms on the other.

Table I. Complete Algorithmic Expansion of <ifstatement> syntax.

```
ifstatement:    stack(back)
                recognize(if)
                find subpart(expression)
                find subpart(relatop)
                find subpart(expression)
                recognize(then)
                find lexical (line number)
                popyes
expression:     stack(expressm)
                find lexical(name)
                recognize(plus)
                find subpart (expression)
                popyes
expressm:       stack(expressnam)
                find lexical (name)
                recognize(minus)
                find subpart (expression)
                popyes
expressnam:     stack(back)
                find lexical (name)
                popyes
relatop:        stack(lessrel)
                recognize(gtsign)
                popyes
lessrel:        stack(eqrel)
                recognize(ltsign)
                popyes
eqrel:          stack(back)
                recognize(eqsign)
                popyes
back:           popno
```

The FIND LEXICAL procedure will normally be able to determine whether its success or its failure return is to be taken by direct examination of certain pre-established "attribute" bits in a symbol table, and consequently has, in this regard, a flavor more like that of the elementary RECOGNIZE routine than like that of the more complex FIND SUBPART routine. FIND LEXICAL differs from RECOGNIZE however, in that instead of merely returning a success indicator in case of success, it enters a pointer to the successfully found lexical atom into the argument stack. This pointer will, in the manner explained above, subsequently be incorporated into the complete syntax tree which we take to be the desired output of the syntactic analysis procedure.

Note that, in the algorithm shown in Table I, we have assumed that <ifstatement> is a syntactic subtype forming part of a larger Backus grammar, so that, when and if our input string fails to parse as an "ifstatement" we merely make a normal failure return to the higher level grammatical from which FIND SUBPART(IFSTATEMENT) has been invoked.

Note also the fundamental fact that the syntactic analysis algorithm shown in Table I can be obtained by straightforward mechanical expansion from the Backus normal form description (shown in formula (3) above) of <ifstatement>. Of course, in a mechanical expansion, meaningless rather than mnemonic labels would be attached to the code corresponding to the two alternative versions of the syntactic type "expression"; in Table I, we have attached the mnemonic labels "expressm" and "expressnam" to these alternative versions of the syntactic type <expression>. Except, however, to the human reader of such an algorithm, this point is without significance. The detailed methods by which Backus normal form descriptions of the type (3) are to be expanded into algorithms of the type shown in Table I will be discussed in more detail below, when we introduce conventions enabling us to describe code generation in direct connection with syntactic analysis.

It is worth noting as an important aside that routines of the sort shown in Table I may either be "executed" or "interpreted", i.e., may either be expanded into a true sequence of machine level subroutine calls, or, alternatively, may be expanded into tables of some other form descriptive of the pattern of calls to be executed, the transfers to the corresponding subroutines then being made by a special "interpreter" code. The first method, i.e., direct execution, has the advantage of high speed; on the other hand, the second may often require less space for the representation of the same procedures, but involve some loss of speed. The particular style, execution or interpretation, which is most advantageously employed in any particular case depends on practical considerations, i.e., on the extent to which compile time on the one hand and space limitations on the other represent critical bottlenecks for a given machine and with a given total work load.

If the Backus normal form descriptors with which we are concerned are to allow the repetition feature described in the paragraphs preceding formula (11) of Chapter 1, we must have the ability to use a corresponding routine in our syntactic analysis algorithms. For the sake of definiteness and convenience, we may call this routine ITERATE SUBPART. The routine in question, which has a single argument denoting a particular composite syntactic type, will work in the following way. On being invoked, it will issue the corresponding FIND SUBPART call repeatedly and indefinitely until a first unsuccessful return from FIND SUBPART occurs. Until this "first unsuccessful return from below" occurs, ITERATE SUBPART merely continues to call FIND SUBPART, always with the same argument, i.e., the argument with which ITERATE SUBPART itself has been called. When the first unsuccessful return from below occurs, ITERATE SUBPART makes a successful return to the level from which it has been called. Note therefore that ITERATE SUBPART can never return unsuccessfully.

A tree building procedure, which can be employed by ITERATE SUBPART just before its terminating return is made, is as follows. All the stacked instances of the syntactic type found by the repeated calls to FIND SUBPART are linked together into a tree of the structure shown in the following figure. If this number is zero, a nominal nulltree is created. The stacked instances are then removed from the top of the argument stack, and replaced by a single entry at the top of the argument stack which references the newly created subtree.

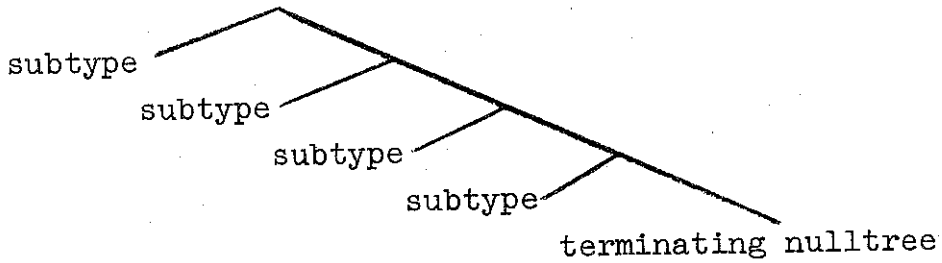


Figure 11. Tree structure which may be generated on return from ITERATE SUBPART(subtype) call.

Alternatively and somewhat more efficiently, a special type of node containing an explicit "branch count" field may be introduced, and the appropriate number of branches hung directly on this single node.

An example of a syntactic analysis algorithm involving an invocation of the ITERATE SUBPART routine is obtained by considering the grammar of the FORTRAN subroutine statement, represented by the following Backus normal form descriptors.

<subroutine statement>
(4) = SUBROUTINE<*name> (<*name> <namec*>) | SUBROUTINE<*name>
<namec> = , <*name>

The corresponding recognition procedure is shown in Table II.

Table II. Syntactic Analysis Algorithm for SUBROUTINE Statement in FORTRAN.

```
subrout:    stack(noargs)
            recognize(subroutine)
            find lexical(name)
            recognize(leftparen)
            find lexical(name)
            iterate subpart(namec)
            recognize(rightparen)
            popyes
noargs:     stack(back)
            recognize(subroutine)
            find lexical (name)
            popyes
namec:      stack(back)
            recognize(comma)
            find lexical (name)
            popyes
back:       popno
```

It is worth noting, from a more general point of view, that subroutine invocations of the FIND LEXICAL type merely exemplify something more general, namely, calls which lead from one style of syntactic analysis to another. In particular FIND LEXICAL takes us from recursive syntactic analysis to finite state lexical analysis of the type described in Chapter 3. From the point of view of general principle however, it is not essential that precisely this transition be made. For example, while a general grammar may require the use of the fully recursive mechanisms described in the present chapter and paragraphs, the subanalysis corresponding to some particular syntactic type and of all the subtypes that can occur in the course of its analysis, may correspond to a subgrammar which is sufficiently restricted so that some simpler and more efficient grammatical analysis

method becomes applicable. In particular, it is possible that a grammatical subtype within a fully recursive Backus grammar can be analyzed by the more efficient "bounded context" methods described in Chapter 4. In this case, one obtains a more efficient overall grammatical analysis if, on beginning the analysis of the syntactic type in question, one passes, via a subroutine call, out of the general recursive framework which we have described in the preceding paragraphs, and into a modified syntactic analysis algorithm of a more efficient kind, possibly written in a different metalanguage. In the simplest case, this more efficient procedure may simply be a lexical scan procedure and, at its very simplest, may be carried out as a lexical pre-scan before syntactic analysis even begins. On the other hand, calls to alternative syntactic analysis procedures of quite arbitrary style may be interpolated into syntactic analysis algorithms of the type shown in Tables I and II, provided only that the interfaces between the different syntactic analysis routines are established by common convention and are respected by all the syntactic analysis subroutines invoked.

4. Code generation in the syntax analysis context.

Up to the present point in our discussion, we have evaded all the issues pertaining to code generation, merely assuming that our syntactic analysis process generated an explicit syntax tree usable as input to a code generation process. However, it is more convenient in many situations to combine syntactic analysis with code generation, since, as we have already observed, the individual pointers required by code generation routines are precisely those which are developed in the course of syntactic analysis before explicit conglomeration of these pointers into a syntactic tree.

In order to do this, we require a method by which calls to a variety of generator routines may be specified in direct connection with syntactic descriptions of Backus form. This is conveniently done by modifying the Backus definers described at the beginning of section 1 of the present chapter, so that they become what might be called "generating definers" i.e., symbolic strings describing both a series of syntactic elements and the particular generative action which is to be taken on successful detection of all the elements of such a syntactic string. In line with this general intent, we define a generating definer to consist of a Backus normal form definer of any of the preceding types, followed by two vertical bars, followed by the name of a generator routine, or, briefly, generator. The generator which occurs in a generating definer is merely the name of a subroutine which is to be invoked immediately on the successful completion of a parse according to the Backus normal form definer which immediately precedes the generator. Using this convention we can write the combined syntax and semantics of the "IF" statement, shown in (3) above, as follows.

(5) <ifstatement> = IF <expression> <relatop> <expression>
THEN <*line number> . genif

<expression> = <*name> + <expression> . gensum |
<*name> - <expression> .gendif | <*name>

<relatop> = > .gengt | < . genlt | = . geneq

Each of the syntactic formulae shown in (5) consists of one or several separate Backus alternatives; every alternative may call for the use of a distinct generator. The separate alternative definers for a given syntactic type occur to the right of an equals sign and are separated by single vertical bars.

The expression into an algorithm of the generating Backus syntax description (5) is shown in the following Table III, which, for its relation to our previous Backus algorithms without generation, should be compared to Table I.

Table III. Algorithmic expansion of <ifstatement> syntax, code generation included.

```
ifstatement:    stack(back)
                recognize(if)
                find subpart(expression)
                find subpart(relatop)
                find subpart(expression)
                recognize(then)
                find lexical (line number)
                call genif
                popyes
expression:     stack(expressm)
                find lexical (name)
                recognize(plus)
                find subpart(expression)
                call gensum
                popyes
expressm:       stack(expressnam)
                find lexical(name)
                recognize(minus)
                find subpart(expression)
                call gendif
                popyes
expressnam:     stack(back)
                find lexical(name)
                popyes
```

[continued]

```
relatop:    stack(lessrel)
            recognize(gtsign)
            call gengt
            popyes
lessrel:    stack(eqrel)
            recognize(ltsign)
            call genlt
            popyes
eqrel:      stack(back)
            recognize(eqsign)
            call geneq
            popyes
back:       popno
```

The action of the generator routines which occur in the above example may be explained as follows. The arithmetic sum generator, GENSUM, always finds two variable names at the top of the argument stack when it is called. Let these two variable names be designated as A and B. Then GENSUM creates a new variable name, which we shall call C, removes the variable names A and B from the top of the argument stack, places the variable name C on the argument stack, and generates a single line of code, having the significance C = A + B, into an output code stack. In much the same way, the subtraction generator GENDIF, removes the names A and B from the top of the stack, places a new variable name C at the top of the argument stack, and generates a single line of code, C = A - B, into the code stack. Using this procedure, the code corresponding to an arithmetic expression of arbitrary complexity is recursively built up, i.e., an arbitrarily complex arithmetic expression is compiled into a sequence of elementary, perhaps machine language, instructions. The other main generator routine occurring in the syntax formulae (5) is the generator GENIF that is invoked when an if statement has been completely parsed. It is the responsibility of this generator to construct

and emit the particular conditional transfer instruction, which, in low level or machine level terms, corresponds to a FORTRAN IF statement. The action of the GENIF routine is as follows. It finds three items on the top of the argument stack. The first two of these are the names of expressions to be compared; the third is a reference to a code line to which transfer is to be made, conditionally on the result of this comparison. The particular comparison to be made will be a comparison according to either the greater than, the less than, or the equality relation. A flag indicating which relational operation is desired in each particular case must be transmitted to the GENIF routine by that one of the three auxiliary generator routines GENGT, GENLT and GENEQ which is invoked when, in the course of syntactic analysis, one of the three signs >, <, = is found. This flag can be transmitted in any one of a number of ways. The most consistent manner of transmitting such a flag, though not necessarily the most efficient, is simply to place the flag on the top of the argument stack, where GENIF would then find it. If this style of coding is employed, then GENIF will find not three but four items on the argument stack, of which one is a flag indicating the particular relational operation to be compiled. A plausible alternative treatment of the same point is as follows.
A common core location, known both to the GENGT, GENLT, and GENEQ routines, and to the GENIF routine, can be established. Each of the three routines, GENGT, GENLT and GENEQ, when it is called, can simply set this common location to some conventional value known to GENIF. By examining this core location, GENIF can then determine the particular comparison to be compiled in each particular case. This technique, while conforming less well to a very systematic approach aiming to use as few data structures as possible during the compilation process is, of course, somewhat more efficient than the technique described previously, in that it avoids some of the intermediate stack manipulations required by a systematic approach. At any rate, whatever

particular device is used for the transmission of the comparison-signifying flag to the GENIF routine, GENIF will find at the top of the argument stack two values to be compared, and the number of a line of code to which transfer is to be made depending on the outcome of this comparison. Call the two variables in question A and B. The GENIF routine will then generate two instructions into the output string. The first of these may normally be a subtraction which forms the difference of the two variables with which the comparison to be performed is concerned. Suppose we call this difference C; then the second instruction emitted will be one of three possible conditional transfer instructions. All of these conditional transfer instructions will transfer to the code referenced by the label which constitutes the final argument of the GENIF routine; a "transfer on positive" will be used if the conditional transfer is to be made on the > relation, a "transfer on negative" will be used if the conditional transfer is to be made on comparison by the relationship <, and, finally, a "transfer on zero" will be emitted if the relational operator occurring in the original if statement is that of equality.

Note that each generator routine, as a normal matter, receives as its arguments a rather small set of parameters representing the condensed result of calls to prior generators which are invoked recursively in the course of analyzing composite syntactic subelements via the subroutine FIND SUBPART. This circumstance corresponds, of course, to the fact that in generating a syntax tree, only one single node is created on the successful completion of any syntactical alternative, all necessary subtrees representing contained syntactic subelements having already been generated recursively via intermediate calls to FIND SUBPART. We may summarize this basic fact in a phrase by stating that <u>lower syntactic levels are normally predigested</u>.

The Backus metalanguage, extended to include calls on generator routines in the way indicated above, may appropriately be called the generating Backus metalanguage. As observed at the end of the first section of the present chapter, the Backus metalanguage may be used to define its own syntax. Similarly, the generating Backus metalanguage may be used to define itself, both as to syntax and as to semantics. In doing so, we must, as noted at the end of the first section of the present chapter, avoid confusion between literal symbols occurring in the language being defined and the markers used as metalinguistic separators in the definition of this language. As previously, we attain this end merely by substituting other symbols for the few symbols which occur as separators in the generating Backus metalanguage; that is, we substitute round brackets for pointed brackets and a / for a |. With these conventions, the self-definition of the generating Backus metalanguage appears as follows.

```
(wholedescription) = (defingroup*) END . GENBACK
(defingroup)       = (definition)
(definition)       = (nampart) = (definers)
(nampart)          = <(*name)>  . LABEL
(definers)         = (definer) (alternate*)
(alternate)        = (bar) (definer)
(bar)              = |  . NEWLABEL
(definer)          = (part) (part*) . (*name)  . CALL /
                     (part) (part*)
(part)             = (*name) . RECOGNIZE / <(*name)>  .SUBPART
                     / <*(*NAME)> . LEXICAL
                     / <(*NAME)*> . ITERATE
```

The eight generator routines which occur in the above syntactic formulae are LABEL, NEWLABEL, CALL, RECOGNIZE, SUBPART, LEXICAL, ITERATE and GENBACK. The reader will find it easiest to understand the significance of these various

generators if he compares the syntactic description (3) of the <ifstatement> with the corresponding syntactic recognition algorithm given in Table III. Each of the eight generator routines which we have listed has tne responsibility of constructing a part of a syntactic analysis algorithm like that shown in Table III from the corresponding element in the more condensed representation of the same algorithm as contained in syntactic formulae of the type of (3) above (or (4) above, etc.).

The generator routines RECOGNIZE, SUBPART, LEXICAL, ITERATE are quite simple. Each of these four generator routines finds a single parameter at the very top of the argument stack; this parameter is merely the name of some lexical or syntactic type. Each of these generator routines merely emits a call to a corresponding subroutine into an output code buffer. These calls have, of course, the respective forms RECOGNIZE(name), FIND SUBPART(name), FIND LEXICAL(name), ITERATE SUBPART(name). Upon emitting its output, each of the above generator routines also removes its argument from the top of the argument stack, thereby reducing the argument stack slightly in accumulated size. Thus, each of the four generator routines RECOGNIZE, SUBPART, LEXICAL, ITERATE simply translates an element of a Backus syntactic string like those of (3) into the corresponding call seen in Table III. The generator routine CALL is equally simple; it receives, as argument, a single name, which it finds at the top of the argument stack. The generator routine CALL merely transfers this name from the top of the argument stack into the output stack. In this way, it introduces function invocations, i.e., generator routine calls, into the output string. The two generator routines LABEL and NEWLABEL have a somewhat more complicated structure, owing to the fact that during the expansion into algorithmic form of a syntactic formula having multiple alternatives, and with the understanding that only a single forward pass over the syntactic formula is to be made in order to secure its complete expansion, it is never known at the beginning of the expansion of a

particular alternative whether another alternative will follow, or whether the alternative in question is the last alternative generating definer on the right hand side of a complete Backus definition. We overcome this small difficulty as follows. A counter is kept; this counter is used to generate an indefinitely long sequence of labels of some such form as L0000, L0001, L0002, ... Each of the generator routines LABEL and NEWLABEL is assumed to generate an instruction of the following form:

label 1: stack(label 2)

In this labelled call, "label 1" will be found by the generator routine LABEL as one of its arguments, while, on the other hand, the generator routine NEWLABEL will calculate label 1 from the current value of the label generating counter in the manner just explained. Moreover, both of the routines LABEL and NEWLABEL will increment the current value of the label generating counter, and will construct the label "label 2" in the output string shown above using this new value of the label counter. Thoughtful examination of the code shown in Table I will reveal the fact that, by following this procedure, the correct output code will be generated whenever the Backus definer momentarily being expanded is followed by an alternative. However, in those cases in which the Backus definer being expanded is not followed by an alternative, but constitutes the last definer on the right hand side of a complete Backus definition, the code generated is incorrect, in that it should have the form

label: stack(back)

instead of the form displayed above. In order to obtain the correct output, we subsequently reach back in the generated output string to make the necessary correction; i.e., we change whatever label has been generated in forming the symbolic output STACK(LABEL2) to the alternative form BACK. In order

to make this process of recorrection efficient, we proceed as follows. A core location containing the address in the output buffer of the generated label (label 2 in our preceding discussion) is saved. Call this core location LASTSTACK. When the generator routine LABEL is called, i.e., when a new definition appears in the syntactic source text, we know that the last preceding alternative is not followed by any other alternative belonging to the same definition, and hence, as explained above, the last previously generated call STACK(label 2) generated is in error and must be corrected. Thus, the syntactic subroutine LABEL will, whenever it is called, correct the output buffer location referenced by LASTSTACK, changing the label appearing in the referenced location to the label BACK.

Our final generator routine GENBACK generates the labelled line

```
back:  popno          ,
```

thereby completing the syntax expansion process. GENBACK must execute any necessary cleanup operations, including the final emptying of the generated output buffer.

The reader may test his understanding of the expansion process just outlined by hand-expanding the above self-definition of the generating Backus metalanguage into the corresponding syntax expansion routine. Such hand-expansion is, of course, the basic "bootstrap" step necessary in initial implementation of the syntax expander.

The first, non-generating form of the Backus metalanguage, as presented earlier in the present chapter, may be regarded as a degenerate form of the generating Backus metalanguage, in which every definition ends with an implied call to a standard POPYES tree section generator. From this point of view, the essential difference between the generating and the non-generating Backus metalanguage is that the generating Backus metalanguage allows arbitrary generator routines to be called at the end of each section of parse, rather than forcing a call to a single standard generator in every case.

5. An Expanded Backus Metalanguage.

The generating Backus metalanguage described at the end of the preceding section, while containing everything necessary in principle for the expression of syntax and semantics, is somewhat too slight to be truly serviceable in the formal description of any very substantial programming language. In the present section, we shall introduce an expanded and very much more adequate version of the Backus metalanguage, sufficiently rich in features as to be comfortable during extensive use. We may motivate the features of this extended metalanguage as follows. In the first place, the Backus metalanguage as presented above is based on a uniform "recovery" convention according to which we always back up to the end of the last previously complete successful section of parse on failure to find any required element of any Backus definer. Besides leading to unnecessarily inefficient parses, this procedure has the decisive flaw of not allowing the convenient generation of error diagnostics. In order to improve the power of the metalanguage in both senses, it is appropriate to supplement the Backus definer strings in terms of which syntactic descriptors are composed by adjoining to them strings of transfer labels, each explicitly indicating the point to which transfer is to be made when the corresponding syntactic element of the definer cannot successfully be constructed. While this complicates the form of a definer, and imposes an additional burden of decision on the syntax writer, it more than pays for itself in flexibility and power gained. Introducing this additional string of transfer labels, we gain the freedom to invoke generator routines at any point in a syntactic scan, provided always that our text takes proper account of the prior occurrence or non-occurrence of a generator invocation. Generator subroutine calls are most useful and powerful if we allow them to have explicit arguments; of course, each generator also has available, as a set of

implicit arguments, the quantities which occur at the top of the syntactic argument stack described in the preceding section. Thus, in defining our extended metalanguage, we allow ourselves to intersperse into a definer string not merely simple generator routine calls but generator routine calls with associated lists of parameters. It is also convenient to allow syntactic strings to end not only with an implied call to a POPYES routine, but also with an explicit transfer to any syntactic label. Using such a feature we may, for example, efficiently combine the description of two syntactic types having a common terminal part by writing each as an initial section, both terminating in an unconditional transfer to a Backus definer representing their common terminal part. By the same token, it is convenient to allow conditional transfers to an arbitrary syntactic label to be inserted into a definer string, transfers conditional, that is, on the state of the syntactic success-failure flag. By doing so, we allow generator routines which set or reset this flag to be used as conditional tests of quite an arbitrary sort.

In the extended metalanguage, we maintain the "iteration" feature included both in the Backus and the generating Backus metalanguages of the preceding section, and, as a matter of convenience, extend it to allow the specification not only of an arbitrary number of iterations, but also to allow specification either of a minimum number of iterations, or alternatively of both a minimum and a maximum number of iterations. To improve the readability of our syntactic text, we provide, as a very useful feature, the ability to intersperse comments into syntactic text. Finally, in order to specify the occurrence in a syntax string of literal symbols even when these symbols are identical with symbolic marks of the metalanguage, we provide the standard "quote" option. According to this convention, any literal string which does not contain any special symbol of the metalanguage, represents itself; while any literal string beginning and ending with quote marks

but not containing any embedded quote marks, represents the literal quantity contained between the two quote marks. In such a scheme, of course, the quote mark itself requires special treatment, which may be provided in any one of a number of standard ways.

We also find it convenient, for the sake of readability, to make certain modifications in the stylistic conventions employed in the metalanguages of the preceding section. More specifically, we continue to let each definition begin with

(1) <type> = ,

but, when several definers occur as part of a single definition, we separate them by successive equal signs, rather than by single vertical bars; this convention is just as persuasive and readable as our previous convention. Each definer of the extended Backus metalanguage consists of two parts, a direct part and an alternate part. The direct and the alternate parts of a definer are separated by a slash mark. The direct part of a definer corresponds rather closely in form to the definers occurring in the two previous metalanguages, but includes generator calls, etc., freely interspersed with the syntactic elements constituting the definer. The allowed elements of the direct part of a definer are as follows. In the first place, syntactic type designators representing calls on the main recursive FIND SUBPART routine have the same form as previously, i.e., are represented by the occurrence of the name of the corresponding syntactic type, enclosed in pointed brackets. We also allow calls on the FIND LEXICAL routine, which we write, as usual, in the form

(2) <*lextype> ;

similarly, our ITERATE SUBPART calls continue to be written as

(3) <type * > .

Calls on the generalized ITERATE SUBPART routine in a form specifying both a minimum and a maximum number of iterations are written as follows

(4) <type (min,max)> ;

in case we wish to specify only a minimum but not any maximum number of iterations we write

(5) <type(min,*)> .

Comments are written in the following form

(6) <-this is the form of a comment> .

The text of the comment includes everything from the first pointed bracket and its following dash to the next following pointed right bracket. Literal elements, representing calls on the syntactic RECOGNIZE routine, are represented either by the direct occurrence of the corresponding literal in a definer string, or by the occurrence of a quoted literal in the style explained above.

The three last classes of elements which may be inserted into the direct portion of a definer are subroutine calls, unconditional go to indicators, and conditional go to indicators. A generator subroutine call has the following form

(7) .name (param 1, param 2, ..., param k) ,

and may occur anywhere in the direct portion of a definer string. Any number of parameters, separated by commas, are allowed. The unconditional and the conditional transfer indicators have

two allowed alternate forms. The simple form of an unconditional transfer is

(8) ..label

The simple form of a conditional transfer is

(9) ...label ,

where the transfer is taken if the success-failure flag is set to failure. It is also convenient for certain purposes to provide both the unconditional and the conditional transfer in an expanded form in which they are supplied with a single parameter. The expanded form of the unconditional transfer is

(10) ..label(param) ,

the expanded form of the conditional transfer is

(11) ...label(param)

When a transfer, conditional or unconditional, and involving a parameter, is actually taken, we stack the current value of the parameter at the top of the argument stack; this value is then available to generator routines subsequently called. This provides, among other things, a convenient mechanism for signalling a generator routine as to the point from which entry into it was made, a possibility which is useful, for example, in the generation of error messages.

Our list of the items permissible as elements of the direct portion of a definer in the extended Backus metalanguage is now complete. The alternate portion of a definer, which, as we have already noted, is separated from the direct portion by the occurrence of a slash mark, consists of a sequence of explicit or implicit conditional transfers, each of which may be supplied with a parameter if desired. An explicit conditional transfer occurring as part of the alternate section of a definer has the following form:

(12) .label

in case no parameter is provided, and has the form

(13) .label(param)

in case a parameter is provided.

We also allow two useful forms of implicit transfer labels. The form

(14) .b

indicates a conditional transfer to the system BACK label; the occurrence of a completely blank label, i.e., the form

(15) •

indicates a transfer to the next alternative definer of the same definition if any such exists, or, if none such exists, indicates a transfer to the system BACK label. All the conditional transfers in the alternative part of an extended Backus definer are intended to be conditional upon the state of the syntactic success-failure flag, always upon return from the syntactic subroutine corresponding to a particular syntactic element in the direct part of the same definer. Thus, for example, the definer

<type 1> <*lex2> LIT3 / .label 1. label 2. label 3

causes recursive construction of the syntactic type "type 1" to be attempted; if this fails, transfer to the point "label 1" is made. Next, the lexical type "lex 2" is sought, via an invocation of the FIND LEXICAL subroutine; if this element is not found, transfer to "label 2" is made. Finally, the literal element "LIT3" is sought, via a call to the RECOGNIZE routine; if this element is missing, transfer to the point "label 3" is made. On the other hand, if all three syntactic elements are successfully found, the system popup routine is called in its success form POPYES.

Every syntactic element of the direct part of an extended Backus definer which could possibly lead to a failure return

is required to be represented in the alternate part of a Backus definer by one and only one corresponding transfer label. Thus, the total number of transfer labels, implicit or explicit, forming the alternate portion of a Backus definer must correspond to the total number of elements of the direct part of the same definer representing calls to FIND SUBPART to FIND LEXICAL, to the RECOGNIZE routine, or to the ITERATE SUBPART routine when this routine is called either with a specified minimum or with a specified minimum and maximum number of allowed iterations (but not when the ITERATE SUBPART routine is called without the specification either of a minimum or of a minimum and maximum number of iterations).

With the above explanation of the syntax and semantics of the extended Backus metalanguage itself, the reader should be in a position to follow the formal self-description of this language given in Table III. We use the dash (-) to indicate the end of all of the alternatives of a definition. We end the set of definitions with the word END.

Table III. Self-description of the extended Backus metalanguage.

```
<grammar> =  <statement> <statements*>/ .er(2) -
<statements>='-' <statement> / .er(1) .er(2) -
<statement> = '<' <*name> '>' .check...er(6).clear(0)
              .genlab<spart(1,*)>/..er(3).er(4).er(1)
            = END .check...er(6).clear(0).exit/.er(5) -
<spart>     = '=' .clear(1)<direct*>'/'<alternate*>.check...er(6)
              /..er(1) -
<direct>    = '.' '.' .set(uncond) '.' .set(cond)..go/
              .maylb..go
            = <-.><*name> '(' <params> ')' .gencall(p)/
              .er(7)..er(8).er(9)
            = <-.name> .gencall(np)/ -
<go>        = <*name> '(' <*name> ')' .gengo(p)/
              .er(11)..er(8).er(9)
            = <-name> .gengo(np).allri/ -
```

[continued]

```
<maylb>      = '<' '-' <*name> '>' .genfind.count/
               .er(3).er(4)
             = <-<> <*name> '*' '>' .geniter/.er(3)..er(4)
             = <-<name> '(' <*name> ',' <*name> ')' '>'
               .count.genrgo(rept,fin)/.mays.er(13).er(13)..er(9)
               .er(4)
             = <-<name(name,> '*' ')' '>' .genrgo(rept,inf)
               .count/.er(13).er(9).er(4) -
<mays>       = <-<name> '>' .genrgo(norept,fin)/.er(4) -
<magq>       = <*apos> <notapos*> <*apos> .concat.count.genrec
               /..er(12)
             = <*name> .count.genrec/. -
<alternate>  = '.' .set(cond) 'b' .stack(sysbakk).genins(np)/.b.-
             = '.' .patch/. -
             = <*name> '(' <*name> ')' .genins(p)
               /.er(11)..er(8).er(9)
             = <-.name> .allri.genins(np)/ -
<er>         = .ermes<notdash*> / -
<params>     = <*name> <namec*> /. -
<namec>      = ',' <*name> /..er(8) -
<notrbkt>    = '>' .false.backl/.
               <*eof>..er(12)/.
             = <*any>.off/ .er(10) -
<notapos>    = <*apos> .false. back 1 / .
             = <*eof> .. er(12)/ .
             = <*any> / .er(10) -
<notdash>    = '-' .false / .
             = <*eof> .stack(12).ermes.exit/.
             = <*any> / .er(10) -
```

The semantics of the extended Backus metalanguage will be precisely defined once we have described the action of each of the generator toutines that appears in Table III. Let us note, to begin with, that the generator routine ERMES, which

occurs at the label ER in Table III, is simply an error message transmitting routine. This routine always finds a numerical parameter at the top of the argument stack, and merely sends out the error message corresponding to this numerical argument. Table IV contains a list of all the error messages which may appear.

Table IV. Error messages for the extended Backus syntax expander.

1. illformed terminal portion of syntax statement
2. illformed initial portion of syntax statement
3. illformed name of syntactic type
4. missing right bracket
5. missing left bracket in type name
6. number of alternatives does not match number of elements
7. illformed function name
8. illformed function parameters
9. missing right parenthesis
10. system error
11. illformed transfer label
12. unmatched apostrophe or right bracket causes read to end of input
13. illformed repetition bracket

At this point we may also note a convenient and general auxiliary procedure which can be followed in connection with syntactic error diagnostics. One may merely print, on the output listing generated by a syntactic recognizer routine, a line of asterisks underneath any line representing the section of syntactic source text in which an error has occurred. This line of asterisks should terminate at the point in the input text corresponding to the last input symbol successfully scanned by the parser. In this way, the syntax-programmer will be made aware of the approximate point in his syntax text

at which the parser begins to experience difficulty; it will normally be quite close to this symbol that a correction is required.

Certain of the generator routines occurring in Table III have very simple actions. The generator routine FALSE merely sets the syntactic success-failure flag to its failure condition; the generator routine ALLRI sets the same flag to the success condition. The generator subroutine SET merely sets a special "transfer-type" flag location to indicate either that a conditional transfer or that an unconditional transfer is subsequently to be generated. This flag is used by the GENGO generator routine to determine the type of transfer which it is to generate.

The generator routine CHECK checks for a match between the number of elements constituting the alternate part of an extended Backus definer and the number of direct parts which require corresponding alternate parts. This correspondence check is performed as follows. The syntactic ITERATE SUBPART generator always leaves, on the top of the syntactic argument stack, a count of the total number of iterations which it has performed. The generator subroutine CHECK therefore finds the total number of Backus definer alternates on the top of the argument stack when it is called. The total number of elements in the preceding direct part which require a corresponding alternate part are progressively enumerated by the routine COUNT (besides maintaining the current value of this quantity, COUNT also performs several auxiliary actions, described below). The routine CHECK has only to compare the integer which it finds at the top of the argument stack with the accumulated count found in a parser core location, and to set or reset the system success-failure flag, depending on whether these two quantities are identical or not.

The generator subroutine BACK1 simply backs up one character in the syntactic input string; it is called when this atom must, on the one hand, be scanned in order to make the parse of a preceding string of atoms definite, but, on the other hand, does not belong to this preceding string and must be parsed separately. The subroutine STACK simply places the argument with which it is called at the top of the argument stack; this provides us with a convenient way for inserting "implicit" quantities into the flow of our parse; a procedure useful, for example, in making an implicit label to its explicit. The generator subroutine OFF simply removes a single element from the top of the argument stack; it will be called when the argument stack is known to contain a quantity in which we are no longer interested. The generator subroutine CONCAT finds, at the top of the

argument stack, pointers to a sequence of symbol table entries, together with an integer specifying the number of these entries. It then concatenates all the referenced symbols into a single longer symbol; enters this symbol into the symbol table, and replaces the whole string of items on the argument stack by a single item referencing the newly constructed symbol. Subsequently, the whole sequence of symbols contained between apostrophes may be treated as a single symbol.

The action of the remaining syntactic generator subroutines will be better understood if we first explain the style of syntactic output code that these generators are to produce. The occurrence of an explicit label in the syntactic text, i.e., the beginning of a new syntactic definition, generates an output statement of the form

```
label:  continue
```

The implicit labels corresponding to alternate definers forming part of the same definition are treated as follows. A steadily increasing count is kept in a core location which we shall call LASTLAB. The integer values contained in this location are used to generate an indefinitely long sequence of distinct labels. The occurrence of an implicit label in the syntactic text, i.e., the beginning of a new alternative for a given definition, generates an output statement of the form

```
L000n:  continue
```

Immediately after the generation of this label, the location LASTLAB is incremented by one.

The occurrence of the syntactic type in a definer causes a call of one of the forms FIND SUBPART, FIND LEXICAL, etc., to appear in the final output. Each of these calls must be followed immediately by code for a conditional transfer, the label in this transfer being specified by the corresponding element in the alternate portion of the same Backus

definer. This remark makes it plain that, in order to generate the correct final code, we must intersperse fragments of code corresponding to parts of the direct portion of each definer with other fragments of code corresponding to elements of the alternate portion of the same definer. In order to do this, we first place all the code fragments generated by elements in the direct portion of a definer into an intermediate direct code buffer and the fragments of code generated by the elements of the alternate portion of the same definer into an alternate code buffer. When a definer is terminated, either by the occurrence of the next following definition or by the occurrence of an (=) sign representing the beginning of the next alternative of the same definition, the generator routine CLEAR will be called. This generator routine transfers the contents of the two above buffers into a final output code buffer, interspersing them alternately according to the following rule: First, code in the direct code buffer is moved into the output string, down to the point at which a code line calling for an alternate code line is encountered. At this point, a single line of code is transferred out of the alternate code buffer into the final generated code string. The same process then repeats. An element of the alternate portion of the definer always generates a conditional transfer, which may be written in one of the following forms:

ifgo (label)

is the form generated if the corresponding alternate element is a label without any parameter;

ifgop (label,param)

is the form generated if the corresponding alternate element is a label to which a parameter is attached;

ifgo (L000n)

is the form generated by an explicitly blank alternative element; here the label in question is to be constructed in the way already explained from the count maintained in the location LASTLAB; and finally

```
ifgo (sysbakk)
```

is the form to be generated by the occurrence of the implicit transfer element .b as an alternative element.

In order to give those labels LOOOn which should coincide with the system "sysbakk" label SYSBACK the right significance we proceed as follows. The CLEAR routine is given a parameter, allowing it to be called in one of two forms, the first corresponding to the occurrence of the beginning of a new definition, the second corresponding to the occurrence of the beginning of a new definer continuing an old definition. When CLEAR is called in its first form, it places the current value of the LASTLAB counter on a terminal arguments stack. The clean-up routine EXIT, called when the END statement terminating the whole body of syntactic text is encountered, accesses the terminal arguments stack, generating a whole body of statements of the form

```
LOOOm:  continue
LOOOn:  continue
LOOOk:  continue
... etc.
```

and, finally, generating the terminal code lines

```
sysbakk:  popno
```

One last explanation is required to complete our picture of the target code into which we intend to expand extended Backus metastatements. The output code generated by any definer not ending with an unconditional GO TO indicator should be terminated by a POPUP call. On the other hand, if the last element of a definer is an unconditional GO TO statement, this call is superfluous and may be omitted. In order to secure this action, we make use of a "go flag";

this flag is set to an "on" condition whenever an unconditional GO TO is generated, and is reset whenever any other type of statement is generated, and also when a label occurs in the generated code string. The CLEAR routine, when called, consults this flag and inserts a POPUP statement when and only when the flag is found to be in its 'set' condition.

Keeping these facts in mind, we may explain the action of the remaining syntactic generators appearing in Table III. The generator subroutine GENRGO generates a CALL SUBPART statement. The routine GENREC generates a RECOGNIZE statement. GENRGO also has the responsibility of generating ITERATE SUBPART calls in the special case in which either a minimum or a minimum and maximum number of repetitions are called for. The particular version of the ITERATE SUBPART call required is signalled to the GENRGO routine through its pair of parameters. The related subroutine GENITER generates ITERATE SUBPART calls in the most common case, that in which neither nor a maximum number of required iterations is specified. The generator routine GENFIND generates a FIND LEXICAL call. The generator GENCALL sets up subroutine invocations, and is just a bit more complicated than the other generators of its class in that it must be prepared to deal with an arbitrary number of parameters. Each of the above five generators, when called, resets the "go flag."

The subroutine GENGO generates a conditional or unconditional GO TO statement with or without a parameter. The presence or absence of a parameter in the code to be generated is signalled to this routine by the value of its own single parameter. The GENGO routine determines whether a conditional or unconditional transfer is to be generated by examining the transfer type flag which has previously been brought to the right value by the generator subroutine SET. If an unconditional GO TO is generated, the "go flag" is also set by GENGO to the value 1. The routine GENLAB finds a label

at the top of the argument stack and attaches this label to a CONTINUE statement. The GENLAB routine also resets the "go flag." The GENINS subroutine generates a conditional transfer statement, with or without a parameter, and inserts it into the alternate code rather than into the direct code buffer.

The two remaining syntactic generator subroutines occurring in Table III are CLEAR and PATCH. The CLEAR subroutine zeroes the running count of the number of elements in the direct portion of a definer requiring matching alternate parts, empties both the direct and the alternate code buffers into the final output code buffer, and renews these buffers for subsequent use; resets the "go flag"; generates a POPUP statement if the go flag is set; and inserts a label either into the output code buffer or into the deferred label stack, depending on the parameter value with which the CLEAR subroutine itself is called. The generator subroutine PATCH constructs a label from the count contained in the LASTLAB location, and, using this label, makes up a conditional transfer which it places in the alternate code buffer.

The reader will obtain a clear idea of the combined action of all of the above described generator subroutines if, taking the first few lines of the syntactic text contained in Table III, he hand-expands it into the corresponding parsing algorithm; this parsing algorithm is of course the extended Backus syntax expander itself. The "bootstrap" step necessary in implementing the extended Backus metalanguages on a new computer, is of course just such a hand expansion of the syntactic code of Table III, either into some language already available on the computer or directly into appropriate machine language.

6. Dartmouth BASIC -- a simple algebraic language.

In this section, we illustrate the use of the tools developed earlier in the present chapter by applying them to describe the syntax and some of the semantics of a small, simple, but quite useful algebraic language, the BASIC language, developed at Dartmouth for student use there. As is commonly the case with programming languages, BASIC exists in several slightly variant versions, some of which incorporate features not to be found in others. The particular version of BASIC that we will describe is essentially that of the third edition of the BASIC language manual.

The BASIC language includes an assignment statement of standard form, written in the following manner:

LET <variable> = <expression>

The five standard arithmetic operators may be used to form expressions, as may the members of a built-in set of a dozen or so fundamental transcendental functions (including the trigonometric and exponential functions). Moreover, the programmer is allowed to define new functions by any single algebraic statement and to use these functions in forming expressions. Each statement in a BASIC program is required to have a line number; this line number, as a standard matter, is a three digit quantity. Using these line numbers, control of program flow is attained by the use of GO TO statements, IF statements, FOR statements with associated NEXT statements, and GOSUB statements with which are associated RETURN statements. The following well-formed BASIC lines exemplify the use of all of the statements mentioned above.

```
101     LET A(J) = COS(A(J-1)
102     GO TO 109
103     IF A(J) >= A(J-1) THEN 108
104     FOR J = 1 TO 100 STEP 5
```

```
105     LET A(J) = 0
106     LET B(J) = B(J-1) + COS(C(J))
107     NEXT J
108     RETURN
109     GOSUB 108
```

The operation of the assignment statement, the GO TO statement and the IF statement are all rather self-evident. The FOR statement acts as follows. Each FOR statement involves iteration on a particular variable, specifically on the variable whose name immediately follows the key word "FOR" in the FOR statement. The scope of the iteration is the whole body of BASIC text up to the next following NEXT statement with a matching variable; note that a NEXT statement always consists of the key word NEXT followed and concluded by a simple variable name. "FOR loops" of this form may be nested within each other in the manner normal for iterative loops. The GOSUB statement in BASIC cause the execution of a block of code as subroutine, i.e. execution terminated by return to the next following statement after the particular GOSUB statement by means of which execution of the subroutine block in question was initiated. This "subroutine" feature is provided in a rather rudimentary form. In particular, all variables are "common" to a whole BASIC program, no "subroutine name isolation" is provided, nor does a subroutine call involve any special transmission of arguments. The subroutine call or GOSUB action in BASIC consists merely in placing a return address on an address stack, and transferring to the line of BASIC code referenced in the GOSUB statement. The next subsequently executed RETURN statement transfers to the address found topmost on the return address stack.

Input and output in the BASIC language are invoked by the BASIC READ and WRITE statements. The following examples will illustrate the form of these statements:

```
200     READ A, B(10), C(J+1)
201     WRITE A, "TWO VARIABLES MULTIPLIED TOGETHER = "A*B
202     WRITE "AN ERROR HAS OCCURRED"
```

Variables in BASIC may be indexed with one or two indices, i.e., may be one or two dimensional arrays. Arrays of more than minimal size must be explicitly dimensioned; the BASIC dimension statement has the rather standard form shown in the following example:

```
300     DIM A(10), B(20), C(30,40)
```

The definition of new functions in a BASIC statement is accomplished by using DEFINE statements. Each such statement defines a new function. The following example illustrates the form of the BASIC DEFINE statement.

```
400     DEF  FNA(X) = COS(X * X) + SIN (X * X)
401     DEF  FNB(Y) = X + FNA(FNA(X))
```

The name of a programmer function is required to begin with the two first letters FN. Moreover, each function must be defined by a single arithmetic expression.

Every BASIC program is terminated by an END statement, consisting of the single key word END. Immediately preceding the END statement terminating a BASIC program, any number of DATA statements may appear. Each DATA statement consists of the key word "DATA" followed by a list of signed numbers; both integer and real numbers are allowed. A DATA statement merely generates the quantities which it lists into a serial list of input data; this input data is then accessed by READ statements appearing subsequently in the BASIC program.

A formal account of the syntax of the basic language is given in Table V.

Table V. Synatx of a version of the Dartmouth BASIC Language.

```
<nxtlab>      = <*integer> .genlab..nxtstat/.er(8)-
<nxtstat>     = <asign> /.
              = <readstat>/.
              = <pntstat>/.
              = <datast>/.
              = <gost>/.
              = <ifstat>/.
              = <forstat>/.
              = <nextstat>/.
              = <dimstat>/.
              = <gosub>/.
              = <return>/.
              = <define>/.
              = <end>/.
              = <comment>/.er(15)-
<var>         = <*name> '(' <expr> ',' <expr> ')' .subscr(2)/
                .b. simpvar. er(1). .er(1) .er(2)
              = <-name(expr> ')' .subscr(1)/.er(2)-
<simpvar>     = .subscr(0).allri/-
<functrm>     = <*fname> '(' <expr> ')' .genfun/..er(3).er(1).er(2)-
<factor>      = <*number>/.
              = <var>.genfetch/.
              = <functrm>/.
              = '(' <expr> ')' /..er(1).er(2)-
<expfac>      = <factor> '↑' <factor> .arith(expon)/.er(1)..er(1)
              = <-<factor> .allri /-
<term>        = <expfac> <terms*>/.er(1)-
<terms>       = '*' <expfac>.arith(prod)/..er(1)
              = '/' <expfac>.arith(quot)/..er(1)-
<expr>        = '+' <term>.arith(unplus)<sums*>/.er(1)
              = '-' <term>.arith(unmin)<sums*>/.er(1)
              = <term> <sums*>/.er(1)-
<sums>        = '+'<term>.arith(plus)/..er(1)
              = '-'<term>.arith(minus)/..er(1)-
<asign>       = LET<var>.save '='<expr>.genasin..cdend
                /..er(4).er(5).er(1)-
```

```
<readstat>    = READ <var> .genread . genasin <rvar*>
                ..cdend/..er(6)-
<rvar>        = ',' <var> .genread. genasin/..er(6)
<pntstat>     = PRINT <pitem> <pitems*> ..cdend/..er(7)-
<pitems>      = ',' <pitem> /..er(7)-
<pitem>       = <expr> .genprint(value)/.
              = <*mesg> .genprint(mesg) <expr>.genprint(value)/.er(1).
              = .allri/.
<datast>      =  DATA <signed> <csigned*> ..cdend/..er(6)-
<csigned>     =  ',' <signed> /..er(6)-
<signed>      = '-' <*number> .gendat(minus)/..b.
              = <*number> .gendat(plus)/.-
<gost>        = GOTO..goend/.
              = GO TO .. goend/.-
<goend>       = <*integer> .gengo..cdend/.er(8)-
<relatop>     = '>' '=' .setop(ge)/.lesfirst..
              = .allri.setop(gt)/-
<lesfirst>    = '<' '>' .setop(lgt)/.eqlsgn..
              = <-<> '=' .setop(le)/.
              = .allri.setop(lt)/-
<eqlsign>     = '=' .setop(eql)/.er(9)-
<ifstat>      = IF<expr> <relatop> <expr> THEN <*integer>
                .genif..cdend/..er(1).er(7).er(1).er(10).er(8)-
<forstat>     = FOR <*name> '=' <expr> TO <expr> STEP
                <expr> .genfor..cdend/.b.er(4).er(5).er(1).er(11)
                .er(1)..er(1)
              = .allri.stack(1).genfor..cdend/-
<nextstat>    = NEXT <*name> .genext...er(12)..cdend/..er(4)
<dimstat>     = DIM <dimelt> <cdimelt*> ..cdend/..er(7)
<cdimelt>     = ',' <dimelt>/..er(7)
<dimelt>      = <*name> '(' <*integer> ',' <*integer> ')'
                .gendim(2) ...er(13)/.er(6).er(3).er(6).er(6).er(2)
              = <-name(integer>')' gendim(1)...er(13)/.er(2)-
<gosub>       = GOSUB<integer> .gencall..cdend/..er(8)
<return>      = RETURN.genret..cdend/.-
<define>      = DEF <*fname> '(' <*name> .genanam ')' '=' <expr>
                .genasin.defend/..er(4).er(3).er(4).er(2).er(5).er(1)-
```

```
<end>       = END. exit/.-
<backup>    = .backl/-
<er >       = .<ermes><notend*>/-
<cdend>     = <*edge>  ..nxtlab/.er(14)-
<notend>    = <*edge>  .false/.
            = <*eof>   .stack(16).ermes.exit/.
            = <*any>/.er(7)-
```

The parser described by Table V will, in the course of its action, generate various diagnostic messages. A complete listing of all of these diagnostic messages is given in Table VI.

Table VI. Error message list for the BASIC compiler.

1. illformed arithmetic expression
2. missing right parenthesis
3. missing left parenthesis
4. illformed variable name
5. missing equal sign
6. illformed list element
7. system error
8. illformed label
9. illformed relational operator
10. illformed if-statement
11. illformed for-statement
12. improper nesting of for-loops
13. excessively large dimensioned arrays or repeated dimensioning
14. illformed terminal portion of statement
15. illformed initial portion of statement

Table VII contains a complete listing of the generator subroutines called by the parser described in Table V. It also lists the lexical types with which the parser of Table V is concerned.

Table VII. Generator subroutines and lexical types for the BASIC compiler.

Generator subroutines:

false; allri; ermes; setop; stack;
subscr; save; defend;
genfun; genfetch; arith; genasin; genread; genprint; gendat; genif
gengo; gendim; genfor; genext; gencall; genret; genanam; genlab.

Lexical types:

<*name> ; <*fname>; <*number> ; <*edge> ;<*eof> ; <*mesg>.

Certain of the generator routines occurring in Table VII have very simple actions. The generator routine FALSE merely sets the syntactic success-failure flag to its failure condition; the generator routine ALLRI sets the same flag to the success condition. The generator routine ERMES is simply an error message transmitting routine; this routine always finds a numerical parameter at the top of the argument stack and simply sends out the error message corresponding to this numerical argument. The generator routine SETOP merely sets the contents of some flag location to a value indicating which one of an allowed set of relational operators appears in an if statement. This location is then referenced by the GENIF subroutine to determine the particular conditional transfer which it is to generate. The generator subroutine STACK merely places its own single argument at the top of the system argument stack; this gives us a way of inserting implicit arguments into the flow of our parse, in explicit form.

Some of the remaining generator subroutines have quite straightforward code-emitting actions. GENLAB finds a label referenced on the top of the argument stack and generates a labelled continued statement

LABEL: CONTINUE

in whatever assembly language form is appropriate. GENGO finds a label reference (on top of the argument stack) and generates an unconditional transfer

JUMPTO LABEL

in appropriate assembly language form. GENCALL finds a label reference at the top of the argument stack and emits two lines of assembly language code:

CALL RETSTACK
JUMPTO LABEL

In these lines of code, RETSTACK is a reserved system label at which an assembly-written subroutine is entered; this subroutine

a) determines the location from which it has been called;
b) using this address, and adding a small integer, determines the appropriate address for eventual subroutine return;
c) stacks this latter address at the top of a system return address stack, checking for stack overflow and giving a diagnostic in case of overflow;
d) makes a direct return to execute the JUMPTO LABEL instruction which is the second of the two instructions displayed above.

Note that the code sequence beginning at the system address RETSTACK is one of a number of system assembly-written sequences which must be appended by the compiler to the code which it compiles. Other such system sequences will be noted in their turn in the following paragraphs.

The generator subroutine GENRET generates an unconditional transfer of the following form:

JUMPTO RETBEG

RETBEG is a label which must be attached to the initial location of a systems sequence at which

a) the top address of the system return address stack is fetched, a diagnostic being given in case of underflow;

b) an unconditional transfer is made to the address fetched.

The remaining generators have slightly more complex logical actions, in that they are more intimately involved in reading and/or manipulation of symbol table entries, in temporary variable accounting, etc. Before describing these generators therefore, we must explain the general circumstances which determine their form. In doing so, we will have a first exposure, "in miniature" to the wider circle of issues involved in code generation, issues whose more systematic discussion is postponed to Chapter 8 below.

The main issues which must be faced are as follows.

1. Indexed quantities must be fetched and stored using a procedure slightly different from that appropriate for non-indexed quantities. This procedure requires information concerning the dimensions associated with the indexed variable which it is to treat. Inconsistent repeated dimensioning of a single variable must be avoided; a quantity referenced as unindexed in any statement must not subsequently be dimensioned.

2. In the course of computation of a nested expression like (A * B) + (C * D), temporary variables for the storage of A * B and of C * D are required. A suitable rule for the generation of such temporary quantities may be stated as follows: Maintain a temporary variables counter. Each binary arithmetic operation advances this counter by one, but decreases it by one for each argument of the operation which is a temporary variable. Instead of appearing in the symbol table along with programmer defined variable names, temporary variables may conveniently be represented by specially flagged "pointer items" which, in fact, merely contain a temporary variable number. The current maximum value reached by the temporary variables counter must be maintained during compilation, since this information is needed when a new

temporary variable is to be generated, and is also needed when storage for the full set of temporary variables is to be generated. A slight additional complication to all of this arises from the circumstance that, since a programmer-defined function may be called at any point during the evaluation of an expression, each programmer-defined function requires its own private temporary variable storage area. We satisfy this additional requirement as follows. An auxiliary counter is used during the compilation of every BASIC DEFine statement. This counter counts temporaries as they are generated during the compilation of such a programmer-defined function. On terminating the compilation of a DEFine statement, we return to the use of the standard temporary variable counter, but set the value of the auxiliary temporary variable counter to the current maximum value which this auxiliary counter has attained.

If another DEFine statement is subsequently compiled, it will therefore generate temporary variables never used before.

3. In a function definition like

```
DEF FNCUBE(X) = X *  X *  X + C *  D,
```

the argument name X which appears is a logical dummy having no relationship to the variable of the same name which may appear elsewhere in the same BASIC program. We adjust to this fact by treating the name of a function as an ordinary variable and, by using it during the evaluation of the function to represent the argument of the function. More specifically, on calling the function FNCUBE appearing in the DEFine statement above, say, in the form

```
A = FNCUBE(B)
```

we would

a) execute

```
FNCUBE = B
```

before entering the function evaluation code itself.

b) treat the above definition exactly as if it read

```
FNCUBE = FNCUBE * FNCUBE * FNCUBE + C * D
JUMPTO RETBEG
```

c) execute

```
A = FNCUBE
```

on return from the function evaluation code. Substitution of a function's name for the variable X appearing in the definition of the function is accomplished by use of a function switch, which is turned on during the compilation of a DEFine statement, and of two auxiliary locations, in one of which the symbol table pointer referencing the current function name is stored, while the second contains the symbol table pointer to the function argument name. When, in the course of compiling any algebraic expression, GENFETCH is called, it examines the function switch, and, if this is on, replaces any reference to the function argument with a reference to the corresponding name.

4. The generation of iterative FOR-next requires a solution to the problem of matching the beginning with the end of each such loop in proper fashion. The loop-beginning GENFOR routine finds, on the argument stack, a reference to the name of the loop's iteration variable, and to a set of three temporary locations, using which we then generate code of the following form:

```
          TEMP = TEMP1 - TEMP3
          VARIABLE = TEMP
LABEL1:   CONTINUE
          TEMP = VARIABLE - TEMP2
          PLUSJUMP LABEL2
          TEMP = VARIABLE + TEMP3
          VARIABLE = TEMP
```

On encountering the matching NEXT statement which terminates

the loop, we generate the end of loop code

```
        GO TO LABEL1
LABEL2: CONTINUE
```

The pair of labels, LABEL1 and LABEL2, appearing in the above-displayed code must not conflict with any programmer defined label. To attain this end, we maintain a label counter LC which is incremented each time a FOR-loop is generated, and which provides us with an indefinitely long string of generated labels; and also maintain a FOR-stack, into which the routine GENFOR, when called, places both the label counter value it has used for the generation of its labels, and the iteration variable used in the FOR-statement. The corresponding NEXT statement removes these quantities from the top of the FOR stack. Using the stacked value of the label counter GENEXT can generate appropriate loop closing code. The GENEXT subroutine will also compare the variable which it finds at the top of the argument stack with the variable referenced at the top of the FOR-stack to verify that all four loops are properly nested.

To proceed as explained above, we make use of the following data locations and fields:

1. A one bit function switch FS; a function name location FN; and a function argument name location AN.
2. A temporary variable counter TV and a temporary variable maximum counter TM. An auxiliary temporary variable counter ATV and an auxiliary temporary variable maximum counter ATM.

3. In each symbol table entry, a two bit dimension state field, in which 0 denotes a neutral state, 1 denotes a singly indexed variable, 2 denotes a doubly indexed variable, and 3 denotes a simple variable.

In each symbol table entry, a dimension pointer field which references a dimension table entry if the variable is dimensioned; in this case the dimension table entry gives the dimension information for the variable. Also, a total

accumulated array storage counter TAS.

4. A FOR-label counter LC; a FOR-stack and related stack pointer.

5. An indexing flag IX, used to distinguish indexed from non-indexed fetches and stores.

We now proceed to explain the action of the remaining generator subroutines. SUBSCR will be called with an argument which is either 0, 1 or 2, and also references the variable pointed to by the top of the argument stack. SUBSCR(2) finds two index quantities at the top of the argument stack. It sets the indexing flag, checks to verify that the variable with which it is dealing represents a two-dimensional array, and, using the dimension information for this two-dimensional array, emits code to calculate the linear location of the required array entry. A temporary variable containing this quantity replaces the two separate indices at the top of the argument stack. SUBSCR(1) merely sets the indexing flag, and checks to verify that the variable in question represents a one-dimensional array. SUBSCR(0) checks that the variable with which it is dealing is non-indexed or neutral, and designates a variable as non-indexed if it is found to be neutral. Then SUBSCR(0) drops the indexing flag; if the function flag is set it checks the variable for identity with the current function argument, and replaces it with the name in the current function name location if the variable is identical with the current function argument.

The subroutine GENFUN finds a function name and the variable pointer at the top of the argument stack. It drops the indexing flag and, using GENASIGN and GENCALL, generates the machine code sequence

```
FNAME = VAR
CALL  RETSTACK
JUMPTO  FNAME
```

GENASIN finds a pair of variable names and possibly also an index quantity at the top of the argument stack. The index quantity will be present if and only if the index switch is on.

Consulting this switch, GENASIN emits assembly language code to perform a simple store in the nonindexed case, or an indexed store if the index switch is on. If the index switch is on, GENASIN drops it. Note that, if the variable found at the top of the argument stack by GENASIN is a temporary variable, GENASIN must reduce the temporary variable counter TV by 1. GENFETCH finds either a variable name or a variable name and an index quantity at the top of the argument stack. If the index switch is off, GENFETCH acts as a null operation. If the index switch is on, GENFETCH drops it, advances the temporaries counter and generates the assembly code equivalent of the indexed fetch

TEMP = INDEXFETCH(VAR,INDEX).

The generator subroutine ARITH is the main generator for all arithmetic operations. It generates the assembly code sequences required for the various allowed binary arithmetic operations and for the unique monadic arithmetic operation, arithmetic negation; the result value is always assigned to a temporary variable and a pointer to this temporary variable is left at the top of the argument stack. Note that ARITH always advances the temporary counter by 1, but reduces this counter by 0, 1 or 2, depending on whether none, one or both the quantities on top of the argument stack when ARITH is called are temporary variables.

GENIF uses ARITH as a subroutine to form the difference of the two quantities whose comparison is to determine the outcome of the IF statement being compiled, and replaces these two quantities by a single temporary variable containing their difference. Then the GENIF subroutine generates appropriate transfer code, in one of the following forms:

```
                ZEROJUMP  LABEL
```

for a comparison by equality;

```
                NONZEROJUMP  LABEL
```

for a comparison by inequality;

```
                ZEROJUMP  LABEL
                PLUSJUMP  LABEL
```

for comparison by the greater than or equal to relation;

```
                ZEROJUMP  LABEL
                MINUSJUMP  LABEL
```

for comparison by less than or equal to;

```
                ZEROJUMP  OVER
                PLUSJUMP  LABEL
        OVER:   CONTINUE
```

for comparison by the greater than relation;

```
                ZEROJUMP  OVER
                MINUSJUMP  LABEL
        OVER:   CONTINUE
```

for comparison by the less than relationship.

The generator subroutine GENFOR finds a variable and three expressions at the top of the argument stack. Using the FOR-label counter LC it generates the loop heading code already described. Then the loop variable and the current value of the loop counter LC are placed at the top of the FOR stack, and LC is incremented by 2. GENEXT finds a single variable at the top of the argument stack. It compares this variable with the variable referenced at the top of the FOR-stack and, if these disagree, issues an "improper loop nesting" diagnostic. If, on the other hand, no loop nesting error is detected, GENEXT uses the stacked value of the loop counter to generate the loop closing code described.

The two generator routines GENANAM and DEFEND are used in connection with the BASIC DEFine statement. GENANAM finds a function and function argument name referenced at the top of the argument stack; sets the function flag; places references to the function name and argument name in the locations FN and AN respectively; and generates an assembly language label and associated storage location. GENANAM also interchanges TV with ATV and TM with ATM. DEFEND drops the function flag, sets TM = ATM, and interchanges TV with ATV and TM with ATM. It also generates the output line

```
JUMPTO  RETBEG
```

The GENDAT and READ routines work together, using a common system INPUT array. GENDAT finds an integer or real number at the top of the argument stack; it converts this quantity to its internal value, takes the negative of this value if called with "minus" as its own proper argument, and puts the calculated value into the next free space in the input buffer array. The GENREAD routine finds a variable name A referenced at the top of the argument stack, emits the lines

```
CALL  READER
A = IOTEMP
```

and stacks the pointer to the system variable IOTEMP on the top of the argument stack; the system routine READER here invoked merely transfers the first available element in the input buffer into the IOTEMP location, taking a "data exhausted" exit of the INPUT buffer is empty.

GENPRINT can be called in a "data print" or in a "message print" form. In its first form, it uses GENASIN to emit code which will transfer the value to be printed into the system IOTEMP location, and then emits a call to a system value-print routine which accomplishes the conversion and exterior transfer of this value. In its second form, GENPRINT emits code for entering, into the IOTEMP location, the two message buffer pointers defining the beginning and

end of a message to be printed, and emits a call to a system message print routine which accomplishes the external transfer of this message.

The generator subroutine GENDIM finds a variable name and one or two constants on the top of the argument stack. It checks the dimension state field of the variable to make sure that the variable is in the neutral state, and then sets this dimension state field to the appropriate value, 1 or 2. At the same time, a dimension table entry is built and a pointer to this dimension table entry attached to the variable entry in the symbol table, and the total space required for the dimensioned array calculated and added to the total array storage count TAS, a diagnostic being emitted if this total becomes excessive.

EXIT attatches assembly language code for all necessary system functions to the code compiled during the immediately preceding parse run; generates all necessary temporary storage locations, using the value of both the TM and the ATM counters; generates storage for all necessary arrays, using the entries in the dimension table, and transfers the accumulated contents of the input buffer and theprint message buffer to proper location. Finally, EXIT will print out an assembly language listing of the code that has been compiled and/or call an assembler to produce final machine code.

This is the last of the generators required for the compilation of the BASIC language. Programmed in a suitable higher level language, each of the above generators should be no more than few dozen lines long; so that the whole parser for the BASIC language, written in a combination of our syntax language and FORTRAN should amount to no more than two to three hundred cards.

7. Optimization of recursive syntax analysis.

The syntax analysis methods presented in the previous sections of the present chapter, especially in Section 5, are at their most efficient when the language to be parsed has a high degree of local ambiguity. In this case, the parsing algorithms of Section 5, in the form given, imply a great deal of "saving" and "restoring"; i.e., we must often note and save an indication of our momentary location in the input string in order, on a subsequent failure, to reattempt a parse along an alternate line, starting once more from the same point in the input string. In some cases, of course, the language which is to be parsed may be relatively unambiguous; this, for example, is the case for both the extended Backus metalanguage itself and for the BASIC language described in the preceding section. This is also the case for many other programming languages: the design of such languages is often influenced by a desire to avoid ambiguity. Even in these cases, however, there are certain common situations in which a recursive parser acts with less than optimal efficiency. In the present section, we will describe several methods by which the efficiency of a parser may be improved, irrespective of whether the language to be parsed is ambiguous or unambiguous.

We begin by considering the situation, typified by the "keyword search" which begins the syntactic analysis of every BASIC statement, in which a lexical atom must be compared with every one of a large number of fixed atoms before a decision can be made concerning the line of action to be taken subsequently. We shall describe a relatively simple device for increasing the efficiency of parsing in this common case.

The idea is as follows. Instead of making comparisons with all the members of a relatively large set of other atoms, and taking a transfer depending on the result of all these comparisons, we make use of a hash table of atoms containing the corresponding transfer addresses, and of an alternative

procedure in which we

a) hash the atom which is to be compared with our table entries;

b) locate this atom in our hash table by a fast look-up procedure;

c) note, and transfer to the corresponding address.

This technique speeds up the key word scan which, according to the grammar of Section 5, begins the syntactic analysis of every BASIC statement very effectively.

A more widely useful optimization makes use of somewhat more sophisticated techniques resembling the "bounded context" parsing techniques which we will discuss in more detail in Chapter 4. This technique, which may be called context anticipation, involves the following sequence of steps:

1. During the expansion of a grammar written in the extended Backus metalanguage into the corresponding algorithm, make up a complete list, called a composites list, of all FIND SUBPART and all ITERATE SUBPART calls, both those involving and those not involving the specification of a minimum or a minimum and maximum number of iterations. This may readily be accomplished by making slight modifications to the GENRGO and the GENITER routines used by the syntax expander of Section 4.

2. During the same syntax expansion, make up a list, called the symbol types list, of all recognized atoms and lexical types (this is merely a list of all the quantities occurring as arguments to the RECOGNIZE and FIND LEXICAL calls). This goal may be accomplished by making slight modifications to the GENREC and GENFIND routines used by the syntax expander.

3. Next, using the tables prepared (in steps 1 and 2), make up a two-dimensional outcomes table for each call appearing in the composites list. Each entry in this table corresponds to a pair S1, S2 of elements on the symbol types list. The entries of each such outcomes table are to be formed as follows:

a) Begin a simulated run of the parser, starting with the given FIND or ITERATE SUBPART call, and with a two symbol input string consisting of S1 and S2.

b) If and when any subroutine other than POPUP, POPNO, FIND SUBPART, ITERATE SUBPART, FIND LEXICAL, RECOGNIZE, ALLRI, or FALSE is called during the simulation, terminate the simulated run and enter the symbol D into the outcomes table.

c) If the input scan pointer maintained by the parser ever attains the value 3, terminate the simulated run and enter D into the outcomes table.

d) If after a set maximum number of simulated steps, (say 100 or 1000) the simulated run has not yet returned from the original FIND or ITERATE SUBPART call, terminate the run and enter the symbol D into the outcomes table.

In every remaining case, return from the initial FIND or ITERATE SUBPART calls would have occurred without any generator call other than those listed in b) above having been made, and without any character other than the two initial characters S1, S2 having been scanned.

In this case, the following material is entered into the outcomes table:

i) The state of the syntactic success-failure flag on return; the value, 1 or 2, of the scan pointer on return; and, if this value is 2, so that a character has been scanned, a flag indicating whether or not a pointer to this character has been inserted into the argument stack.

ii) The total number of calls to any of the admissible routines listed in b) above that have taken place between our initial FIND or ITERATE SUBPART call and the subsequent return from this call.

4. The next step is to reject any FIND or ITERATE SUBPART call, which according to a reasonable criterion, is not worth special optimization. This may be done as follows. The entries in the outcomes table containing a D are entries which, because of the complex actions which would be initiated

on encountering the corresponding pair S1, S2 of input symbols, have in any case to be performed; that is, for such entries, we do not attempt any special optimization. On the other hand, entries not containing a D are entries which can, if it is worth our while to do so, be specially optimized. Then, in order not to optimize unnecessarily, we take, for each complete outcomes table, the maximum of all the numbers of intermediate calls recorded in any non-D entry in the table. If this maximum exceeds a reasonable threshold value (say, 2 or 3) it signifies that the corresponding FIND or ITERATE SUBPART call could lead to a complex and lengthy series of actions, terminating however, in a simple return from the call, at most one input character having been scanned in the meanwhile. In such cases it is worth setting up a special optimization. On the other hand, if the cited maximum is less than 2 or 3, it is not worth optimizing in any special way, since the procedure followed by our unoptimized algorithm is bound to be reasonably fast and efficient. Thus, depending on the value of the cited maximum and on the threshhold value we choose, we decide either to optimize or to neglect the optimization of a given FIND or ITERATE SUBPART call.

5. In those cases in which we decide to optimize a SUBPART call, we proceed to set up a packed and condensed version of the outcomes table, proceeding as follows. We first reduce each entry in the outcomes table to an entry four bits in length. This is done as follows. A 'D' entry is represented by four bits, all zero; an entry different from D is represented by a certain pattern of four bits, the first of these bits being set to 1. The second bit of these four indicates the state of the syntactic success-failure flag on return from the corresponding SUBPART call. The third bit indicates the number, zero or one, of input atoms scanned at the time of return from the SUBPART call. The final bit indicates whether the scanned input symbol, if any is scanned, is merely RECOGNIZED or is taken

by the FIND LEXICAL subroutine and placed on the argument stack. Proceeding in this way, we reduce our initial outcomes table to a packed table of four bit entries, one entry for each possible pair of elements of the symbol types list.

6. Our next aim is to condense the packed outcomes table, arriving at a considerably smaller table containing the same information. This we do as follows. By sorting the rows and then sorting the columns of the packed outcomes table, we classify all the different symbols occurring in the symbol types list into equivalence classes, two symbols being considered equivalent if the rows in the outcomes table which correspond to these two symbols are identical and the columns in the outcomes table corresponding to these two symbols are also identical. Applying this criterion, we develop a list of equivalence classes of symbols, the list corresponding to the given SUBPART call, which will normally be very considerably shorter than the full symbol types list, and, in many cases, may contain some 4 to 8 entries. We next construct an equivalence class table which, for each symbol type in our symbol types lists, gives the number of the corresponding equivalence class. The condensed outcomes table is now a table containing as many rows and columns as there are equivalence classes and, for each pair of equivalence classes, containing the four bit entry which the full outcomes table would associate with a pair of characters belonging to this pair of equivalence classes.

7. Our procedure on entering an optimized FIND or ITERATE SUBPART call is now as follows. We pick up the two immediately following symbols in the input string; and determine the symbol type of each symbol.

Then, making fast access through the equivalence class table and the condensed outcomes table, we decide either to execute our SUBPART call in the normal way, or decide that this is unnecessary since the outcome of the call can be

predicted directly. In this latter case, we use the remaining information in the condensed outcomes table to establish the appropriate setting for the syntactic success-failure flag, to obtain a properly incremented value of the scan pointer, and, if necessary, to place one additional lexical atom pointer and/or the quantity zero (in case of an iterated subpart call) at the top of the argument stack.

Note that the indexing operations necessary to carry out the steps in 7 above can be carried out most efficiently and at highest speed if the number of rows and columns of the condensed outcomes table are both taken to be a power of two.

The information which must be gathered in order to form an initial version of each necessary outcomes table can be collected easily if suitable modifications to the generator routines occurring in the syntax expander are made. These modified generators can produce, from any given syntactic text, a preliminary version of the corresponding parse algorithm in which each of the subprocess calls invoked is replaced by a call to a routine with a combined simulation and information gathering function. At the end of a first information gathering phase, a complete set of condensed outcomes table can be prepared, one for every SUBPART call which is to be optimized. At that point all these condensed tables can be passed back to the syntax expander, the syntax expander at the same time being informed as to what particular SUBPART call each table belongs. Then, using this information, the syntax expander can produce an optimized parser with one single additional run. Note finally that by applying the general optimization methods to be discussed in Chapter 8 below to the parse program thus produced, still further, rather general improvements in efficiency may be obtained.

CHAPTER 3. THE LEXICAL SCAN

1. The basic lexical scan algorithm.

The task of the lexical scan is to divide an input character string into words which can be treated as atomic subunits during the remainder of analysis, to carry out the initial entry of these words into a symbol table, and to flag each symbol table entry with the initial lexical-syntactic type of the word. During this process the machine-internal representation for the value of any symbol word found to designate a constant may also be calculated and entered into the symbol table. The original input string is thereby replaced by a string of pointers to the newly created symbol table.

The lexical scan process prepares in a general way for the principal syntactic pass. It has the obvious advantage of reducing the size of the string to be processed during the principal pass, since many more characters will normally be used to form a mnemonic symbol-name than are required for a symbol table pointer. The lexical scan process is fast, simple, and normally strictly serial. The subsequent stages of syntactic analysis are more complex and may often involve repeated motion back and forth over portions of the input string. Hence, by condensing the input via a lexical scan the number of items of input which must be treated repeatedly is substantially reduced.

It is also worth noting that certain entirely trivial lexical operations will have to be performed very often during the analysis of normal input programs; the suppression of blanks in the parsing of FORTRAN and of repeated blanks in the parsing of many other languages are typical for this observation. By applying a well-adapted high-speed technique in these recurring trivial cases, a lexical scanner can effect considerable savings in total compilation time. Morever, since most of the character-

related operations of a compiler belong to its lexical sections, the use of a lexical prepass serves to localize a compiler's character-set dependencies.

In describing any given programming language we always have the option either of carrying syntactic description down to the single character level or of regarding certain simple syntactic types as lexical atoms. The latter approach generally speeds up the overall compilation process.

From the more general point of view of Chapters 2 and 4, we may make the following comments concerning lexical scans. Any general recursive parsing method makes use of a control stack. If the grammar according to which the parse is to be conducted is sufficiently simple so that, as a matter of fact, this control stack can assume only one of a finite number of states during a parse, then we can develop a considerably more efficient realization of the parsing process as follows. In the first place, enumerate all possible states of the control stack; these become the states of a finite state automaton. Next, note the transitions of control state induced on encountering any given input character, obtaining in this way a state transition table. Whenever a generator subroutine would be called on encountering a given input character in a given state, this fact should be indicated by attaching an appropriate special entry to the state transition table. Reformulating the parsing process in this way whenever possible has the crucial advantage of permitting realization of the parse by a very high speed programming

technique; we may either encode the states of the hypothetical finite state automaton into the instruction address of an actual computer, and encode the state transitions as indexed transfers; alternatively, we may describe the required pattern of state-transitions by a state table and use an indexed lookup of the form

STATE = STATE TABLE(STATE,CHARACTER)

to achieve the proper transition on digesting each new character in the input string.

In the present chapter, we shall describe a heuristic procedure by which the state table and high speed scan corresponding to a sufficiently simple, "lexical" grammar may be set up directly. Those would-be syntactic types which may appropriately be reduced to the status of lexical types may be determined by perusal of the syntactic definition of the language. For a syntactic type to be reducible to the lexical level, we require that its syntactic definition is simple and iterative rather than complex and recursive. In particular, any syntactic type in which a fundamentally recursive structure is embedded as a subpart is not a candidate for reduction to the status of lexical atom. Moreover, the generator routines which need to be called in connection with the construction of a lexical atom should be simple and involve rudimentary initialization but not complex manipulation of the

symbol table.

We may readily perceive the practical force of these theoretical observations by examining the following direct syntactic definition, which might be given for the principal lexical types occurring in the BASIC language of Chapter II, Section 5. (We write these definitions in the most rudimentary Backus language, since, for the moment, we are not interested in the details of the corresponding generative actions.)

```
<ordinary> = A|B|C|D|E|G|H|I|J|K|L|M|O|P|Q|R|S|T|U|V|W|Y|Z
<alphabetic>  =  <ordinary>|F|N
<digit>    = 0|1|2|3|4|5|6|7|8|9
<integer>  = <digit> <integer> | <digit>
<decimal>  = <integer> . <integer> | <integer> .
<exponent> = E<integer>|E - <integer>
<number>   = <decimal> <exponent> | <decimal> | <integer>
<nonspecial>=<alphabetic> | <digit>
<nametail> = <nonspecial> <nametail> | <nonspecial>
<name>     = <ordinary> <nametail> | N<nametail>
             |F<ordinary><nametail>|F<digit><nametail>
             |FF<nametail>
<fn>       = FN<nametail>.
<fname>    = SIN|COS|TAN|ATN|EXP|ABS|LOG|SQR|INT|RND|<fn>
```

It is plain that all these constructions, even those which have been put recursively in the above set of definitions, are expressible iteratively. It is for this reason that the types <integer>, <decimal>, <number>, <name>, and <fname> may appropriately be treated as lexical atoms. On the other hand, referring to the BASIC grammar in Chapter 2, Section 5, we may say that the syntactic type <signed> is not appropriately treated as a lexical atom, since an expression like -3 must be analyzed differently depending on whether it occurs in an expression like "A-3", in which case -3 is the union of the operator "-" with the integer "3", or in a DATA statement of the form

DATE A = -3

in which case -3 is interpreted differently, i.e. as a signed number. Of course, a suitably sophisticated lexical scan could make this distinction lexically; however it is more convenient to let the required analysis wait until the syntactic analysis process. The syntactic type <expr> of the BASIC language, being recursive and involving complex generation, is not at all an appropriate candidate for reduction to the status of lexical atom. In consequence, a syntactic type such as<forstat>, which includes <expr> as a subpart, is also to be treated syntactically rather than lexically. A syntactic type like <var> which in the BASIC language has a very simple construction, can be treated lexically. On the other hand, since the generators which must be invoked on encountering a <var>, in case this variable is dimensioned are somewhat more complex than those normally performed during the lexical process, the analysis of this linguistic element would normally be handled syntactically rather than lexically.

Three principal classes of symbols and their boundaries are normally detected during lexical scan.

The first class of symbols are the 'constants'. These are symbols which denote decimal, octal, or hexadecimal integers, Hollerith constants, etc.

The second principal class detected during lexical scan is the class of 'variable names'.

It is worth making a few remarks concerning the distinction between 'constants' and 'variables'. Some symbols in the input string have, by convention, assigned values which are set initially and which never change during the running of the program. Thus, for example, the string

123

has a conventional value equal to the machine-internal representation of $1 \times 100 + 2 \times 10 + 3$, and this value, once initially established, will be retained throughout the running of the program. Other of the symbols used in a programming language

are formal names distinguished merely by identity or non-identity with other formal names. During the running of the program the values corresponding to these formal names or "variables" will be changed repeatedly.

The three classes of words referred to above do not play equally weighty roles in natural language as in mechanical languages, and it is worth the price of a digression to comment on the reason why this is so. In natural language the first class of constants (also including "noun constants") exhausts normal usage almost entirely. The second class, the blank logical variables, normally occur only as pronouns, never very numerous in any natural language, and in the specialized diction of the mathematical scientist, i.e. as the "P" in such phrases as "Let P be a number...". The basis for this difference in usage is, of course, the fact that enunciations in natural language refer implicitly to a vast text of remembered sentences and linguistic information fragments. Thus the words occurring in natural language sentences normally are not logical blanks, but relate to a rich context which they evoke. Natural language sentences are in thise sense keys for the retrieval of remembered information; the usage of words in a mechanical information retrieval system would correspondingly have a much more natural flavor than the usage typifying other programming languages. Moreover, natural language is rarely used for the description of complex algorithms within which many quantities are varied (though, as noted, the special diction of the mathematical scientist forms an exception to this assertion.) In contrast, the sentences of a mechanical language are normally presented to a machine whose memory is blank except for the presence in it of a stored compiler. These sentences therefore consist of a small number of key words (which invoke particular processes in the compiler) and a large number of distinguishable blank pronouns: the variable names of the programming language. The grammatical mood of the sentences of a mechanical language is normally imperative, and aims to describe complex process.

These are the circumstances that give natural language on the one hand and programming languages on the other hand their rather different flavors.

However, let us return to our main subject. In the lexical scan process we aim to accomplish two results:
1) break down each input string present to us into "words"
2) determine the lexical type of every "word".
One may accomplish word definition merely by inserting a series of explicit end of word marks into the input string. Accordingly, the lexical scan process will normally maintain two pointers: the "scan pointer" indicating the next character to be scanned, and the "word beginning pointer" indicating the beginning of the word currently being constructed (or, equivalenlly the end of the word last constructed). The lexical scan algorith, when it detects the end of a given word, inserts a logical mark indicating this word end and moves the word "beginning pointer" forward over the word just constructed. A little additional discussion will put in in position to indicate the precise manner in which these two pointers are used.

As has been noted above, the lexical scan process is a straightforward one; so straightforward in fact that it is normally performed by a programmed "finite state automaton". Such an automaton is always in one of a relatively small number of logical states. Given the state of the automaton, its next state is a strict function of the existing state and of the next input character scanned. The basic heuristic for establishing the pattern of states required for an automaton intended to perform a lexical scan whose target word-types are known is as follows. At any moment during lexical scan, the scan process will either have completed the construction of a particular word of a given type or will be in the midst of constructing such a word. The type of the word under construction may either be fully known or may be partially known. Thus, for example, in standard FORTRAN, after the characters

123 ,

which we assume to be the initial portion of a word, have been scanned, it known that the word under construction must be either an integer or a decimal constant. Later in the same scan, if the characters

123.45

have been scanned, the lexical ambiguity is removed and it is definitely known that a real constant is under consrruction. This real constant will be terminated by the next following occurrence of a special symbol or alphabetic character. In general, if any definite or ambiguous word type is under construction, the next character scanned will either (resolve any ambiguity and) terminate the word or will (reduce the ambiguity of the word under construction and) continue the construction of the word. While the construction of a word is in process, we advance the "scan pointer", but not the "word beginning pointer". Whenever the construction of a word is complete the present positions of the two pointers define its beginning and end. We use this information to establish a symbol table entry describing the word, and then advance the "word beginning pointer" to the present position of the scan pointer.

We will describe each lexical scan process by a state transition table and an associated short program of lexical instructions.

We may establish a state transition table for the finite automaton which is to perform a given lexical scan as follows:

1) By enumerating all the lexical types provided in the language to be scanned, and all the possible initial ambiguities between these lexical types, arrive at a full set of both definite lexical states and of ambiguous lexical states which can arise during a left to right lexical scan.
2) Each one of these states, be it definite or ambiguous, will correspond to one row of the state transition table.

3) The columns of the state transition table then correspond to the various possible distinct lexical character classes belonging to the given lexical scan process.

Here, two distinct characters are considered to belong to the same lexical class if they play identical roles in the construction of words of all possible lexical types. In the contrary case, characters are considered to belong to distinct lexical classes. Thus, for example, in normal FORTRAN all alphabetic characters but H belong to one single lexical class; the character H belongs to a distinct lexical class because of the special role which it plays in the formation of Hollerith constants.

4) The entries in the state transition table are transfer labels indicating particular algorithmic steps to be performed when a character of a given lexical type is encountered in a given lexical state. The two most common entries in a lexical table will normally be an entry called "continue", which transfers to an instruction in the lexical scan program that merely advances the scan pointer; and a label called "end" which transfers to an instruction in the lexical program that marks the end of a word, enters the newly constructed word into the symbol table, and flags the word as being of a lexical type determined by the lexical state at the time that the word-end condition is detected.

One additional remark will be helpful in understanding the details of the particular example considered below. Any lexical type which has the property that the words which belong to it are always complete as soon as the type is definitely known may be omitted from the state table. Thus, for example, in many languages, any single special character (as *, -, +, /, etc.) will act as a delimiter. Such special characters always constitute single character words; the type of such a word, namely "special", is known as soon as the character is detected. For this reason it is not necessary to include a "special character" column in the lexical analysis table. Instead, it is merely

necessary to include a transfer to a "generate special character" instruction in the "next subunit to begin" row of the lexical state transition table.

As a first example to be analyzed in detail we consider a hypothetical FORTRAN whose list of allowed lexical types is as follows.

1) Integer: any sequence of digits, imbedded blanks allowed.
   Examples: 127, 0359, 0 359 275 851
2) Real number: an integer, followed by a decimal point, and optionally followed by a second integer.
   Examples: 37., 37.0, 37.8910, 3 7.89 10
3) Name: any string of nonspecial characters beginning with an alphabetic; no embedded blanks allowed.
   Examples: JIM, TIM23, T23IM.
4) Special character: any character other than blank or period.
   Examples: *,), (, etc.
5) Period-delimited operator: any string of nonspecial characters, beginning with an alphabetic, containing no blanks, and delimited fore and aft by a period.
   Examples: .GE., .AND., JØIN., .SHIFT.
6) Hollerith constant: any number of digits, followed by the letter H, followed by an arbitrary character string of the length specified by these two digits.
   Example: 5HHØØHA .

A finite-state process which can perform the lexical scan for our hypothetical FORTRAN will have one state corresponding to each of the above lexical types, and, in addition, will have one state corresponding to each possible state of lexical ambiguity which can arise during lexical scan of an input string. The possible states of lexical ambiguity are

12. Integer or real number? (Since those two lexical types can begin with strings of identical form.)
126. Integer, real number, or Hollerith constant? (This additional ambiguity is possible until more than two digits have been scanned.)

25. End of decimal or beginning of period-delimited operator? Consider e.g. the hypothetical statement IF(30.GE.31.0) 1,2,3.

In addition, we require a state, 10, indicating that the first character of the next following lexical unit has not been scanned.

The relevant character subclasses for the lexical scan are: alphabetic other than H, H, digit, special other than period, period, blank, end-of-card mark.

The following lexical transition table indicating the actions to be taken in each lexical state on the occurrence of any given input character may now be drawn up.

Table I. State transitions for FORTRAN-like lexical scan.

lexical states	Character Classes: Alph. ≠ H	H	Digit	Spec. ≠ .	Period .	Blank	End of Card	End of File
2. Real	END	END	CONT	END	END	CONT	END	END
3. Name	CONT	CONT	CONT	END	END	END	END	END
5. Period-delimited operator (PDØ)	CONT	CONT	CONT	END	END1	END	END	END
12. Integer or Real (IR)	END	END	CONT	END	GODCOP	CONT	END	END
126. Integer, Real or Hollerith (IRH)	END	HOLCON	CONT	END	GODCOP	GOIR	END	END
25. Decimal or PDØ (DØP)	ENDX	ENDX	GOR	END	END	GOR	END	END
10. Next subunit to begin (NXT)	GONAM	GONAM	GOIRH	END1	GOPDO	SKIP	NEXT	EXIT

The procedures to be taken on transfer to the various points indicated in the above lexical transition table are shown in the following table.

Table II. Generative lexical actions for FORTRAN-like scan.

End: Enter the immediately preceding block of characters into the symbol table, calculating the internal representation of its value if appropriate.
Flag the symbol as real if state = 2; name if state = 3; operator if state = 5; integer if state = 12; integer if state = 126; real if state = 25; special if state = 10.
Set state = 10.

Advance the 'beginning of next lexical block' pointer to 1 less than the current position of the scan pointer, and go to JUMP.

END1: Advance the scan pointer by 1, and go to END.
ENDX: Reduce scan pointer by 1; set state = 12; go to END.
GODCOP: Set state = 25, go to CONT.
GOIR: Set state = 12, go to CONT.
GOR: Set state = 2, go to CONT.
GONAM: Set state = 3, go to CONT.
GOIRH: Set state = 126, go to CONT.
GOPDO: Set state = 5, go to CONT.
SKIP: Advance 'beginning of next lexical block" pointer by 1; go to CONT.
NEXT: Read in next card, re-initialize, go to CONT.
EXIT: Enter E of mark in lexical output string, and terminate lexical scan.
HOLCON: Convert the immediately preceding block of characters to an integer value; advance the scan pointer by this amount.
Enter the block of characters thus delimited into the hash table,

flagging it as a Hollerith constant. Set state = 10. Advance the 'beginning of next lexical block' pointer to 1 less than scan pointer. Go to CONT.

The reader, using any suitable algorithmic language, will have no difficulty in converting the above algorithm into an explicit program.

As a second example of a lexical scan, we consider the scan belonging to the Dartmouth BASIC language described in section 4 of Chapter 2. The lexical types in this language are as follows. A name is any string of characters and digits, beginning with character, and not beginning with the pair of characters FN. A function name is any string of characters and digits beginning with the two letters FN. An integer is any string of digits. A real number is any string of digits including a period, and possibly including an exponent part, which has the form E, denoting exponent, followed by a string of digits, or, alternatively, E followed by a minus sign followed by a string of digits. A message is any string of characters beginning with and ending with a quote mark. Special characters form the final lexical type; each such lexical element is only one character long. We include an end-card mark and an end-file mark amoung the special characters. None of our lexical types, other than message, are allowed to contain blanks. Table III below shows the pattern of state transitions required for the recognition of the above-described family of lexical types.

Table III. State-transitions for BASIC lexical scan.

	Letter ≠ F,N,E	F	N	E	Digit	Special ≠ ,.,-	Period .	Minus -	Blank	Quote '	End	End File
1. NORF1 (name or function)	Go3	Go3	Go2	~~End~~ Go3	Go3	End	End	End	End	End	End	End
2. NORF2 (name or function)	Cont	Cont	Cont	Cont	Cont	End	End	End	End	End	End	End
3. NAME	Cont	Cont	Cont	Cont	Cont	End	End	End	End	End	End	End
4. IRE (integer or real)	End	End	End	End	Cont	End	Go5	End	End	End	End	End
5. RE1	End	End	End	Base	Cont	End	End	End	End	End	End	End
6. RE2	Exp2	Exp2	Exp2	Exp2	Cont	Exp2	Exp2	Go7	Exp2	Exp2	Exp2	Exp2
7. RE3	Exp3	Exp3	Exp3	Exp3	Cont	Exp3	Exp3	Exp3	Exp3	Exp3	Exp3	Exp3
8. MSG	Cont	Cont	Cont	Cont	Cont	Cont	Cont	Cont	Cont	Gen	Gen	Gen
9. NXT	Go3	Go1	Go3	Go3	Go4	End1	End1	End1	Skip	Go8	Nxtd	Exit

In the above table, state 1 is the scan state reached after an initial F has been scanned; state 2 is the scan state reached after an initial F and N have both been scanned. RE1 is the lexical state reached after an integer followed by a period has been scanned; RE2 is the state attained after an integer followed by a period and the letter E has been scanned; and RE3 is the lexical state reached after an integer followed by a period and E and a minus sign has been scanned.

Table IV describes the generative actions which correspond to each of the entries in the above transition table.

Table IV. Generative lexical actions for the BASIC scan.

end:	Enter the immediately preceding block of characters into the symbol table, calculating the internal value of its representation if appropriate. Flag the symbol as a function name if state=2; a name if state = 1 or 3; an integer if state = 4; a real number if state = 5, 6, or 7; a message if state = 8; a special character if state = 9.
	Set state = 9.
	Advance the "beginning of next lexical block" pointer to 1 less than the current position of the scan pointer, and go to JUMP.
cont:	Advance the scan pointer by 1;
jump:	Go to NORF, NORF2, NAME, IRE, RE1, RE2, RE3, MSG, or NXT depending on the current lexical state.
go3:	Set state = 3 and go to CONT.
go2:	Set state = 2 and go to CONT.
go5:	Set state = 5 and go to CONT.
go7:	Set state = 7 and go to CONT.
go4:	Set state = 4 and go to CONT.
go8:	Set state = 8 and go to CONT.

base: Convert quantity to real number, save value, save value of scan pointer, go to CONT.

exp2: Calculate integer exponent value using saved value of scan pointer;

realpnt: Calculate real value using exponent and saved real. Go to END.

exp3: Calculate negative integer exponent value using saved value of scan pointer. Go to REALPNT.

gen: Enter the preceding block of characters into message storage buffer. Replace this block with characters referencing beginning and end of buffer area. Go to END.

nxted: Read in next card, re-initialize, enter endcard mark in lexical atom string, and go to CONT.

exit: Enter eof mark in lexical output string, and terminate lexical scan.

The reader will note that the two lexical scans described above resemble each other rather closely. A language which admits a larger variety of lexical types (e.g., alternate forms of Hollerith constants; quoted strings in a variety of forms, additional constant forms including octal, hexidecimal, etc. constants) may require a somewhat more elaborate lexical scan algorithm; though lexical scans rarely deviate very far from the pattern discussed in the above examples. However, various somewhat more sophisticated types of processing may conveniently be incorporated into a lexical scanner. One of the most useful and interesting of these is a substitutable parameter feature. Detailed parameter substitution conventions may be set up in a variety of ways; in what follows, we shall, for the sake of definiteness, describe one particualr set of conventions.

The basic idea in lexical parameter substitution is to allow certain lexical atoms to act as logical abbreviations for larger strings. Suppose, for example, that, in the basic lexical context described by Tables I and II above, we agree that any symbolic name may be established as a parameter by

writing

$$//\text{NAME} = \langle \text{string} \rangle //\quad ,$$

and that, at each subsequent occurrence of the NAME occurring in the above formula as a lexical atom in our input text, the string on the right of the above displayed formula is to be substituted for the NAME. This may be accomplished as follows. On scanning past a pair of slash marks and encountering a name, one enters the name in the ordinary lexical symbol table, but flags it in a reserved bit position as a parameter. The string of characters immediately subsequent to an equal-sign following the scanned name defines the value of the parameter. This string is scanned in the normal way, but the resulting lexical atom designators are placed in a parameter values buffer rather than in the lexical output string, as normally would be done. Whenever, in the remainder of lexical processing, a name is scanned, the routine which enters this name into the symbol table checks to see if the name is already present and flagged as a parameter. If this is the case, the corresponding string of lexical atoms recorded in the value buffer, rather than the single lexical atom designating the name, is generated into the lexical output string.

In order to avoid an accumulation of parameter names, it is useful to have some method for switching a given name back from "parameter" status to "ordinary" status. This is conveniently accomplished by writing a parameter definition of the special form

$$//\text{NAME}//$$

Table V below gives the pattern of state transitions necessary to implement the parameter substitution feature which we have just described.

Table V. State-transitions for lexical scan with parameter feature.

		Alph. ≠ H	H	Digit	Special ≠ /,.,=	Period .	Slash /	Equal Sign =	Blank	End of Card	End of File
2.	Real	End	End	Cont	End	End	End	End	Cont	End	End
3.	Name	Cont	Cont	Cont	End	End	End	End	End	End	End
5.	Period-delimited operator	Cont	Cont	Cont	End	Endl	End	End	End	End	End
12.	Integer or real Integer	End	End	Cont	End	Go25	End	End	Cont	End	End
126.	Integer, real or Hollerith	End	Holcon	Cont	End	Go25	End	End	Gol2	End	End
25.	Decimal or PDO	Endx	Endx	Go2	End	End	End	End	Gol2	End	End
78.	Special or Definition Switch	End	End	End	End	End	Defsw	End	End	End	End
8.	Definition	Cont	Cont	Cont	Endz	Endz	Offdef	Endy	End	End	End
10.	Next sunbunit to Begin	Go3	Go3	Gol26	Endl	Go5	Go78	Endl	Skip	Next	Exit

Table VI shows the generative lexical actions which correspond to each of the entries in Table V. Note that, in the algorithm of Table VI, we have deviated slightly from the style employed in the coding of Tables II and IV, in that we allow the principal generative action taken in the section of code following the END label to depend on an internally maintained "generator switch."

Table VI. Generative lexical actions for scan with parameter feature.

End:	Enter the immediately preceding block of characters in the symbol table if necessary. If the symbol is flagged as a parameter, go to SUBSTIT. Else flag the symbol as real if state = 2; name if state = 3; operator if state = 5; integer if state = 12 or 126; real if state = 25; special if state = 10 or 78. Place symbol table pointer in lexical output buffer if generator switch = 0; but in parameter values buffer if generator switch = 1.
Adv:	Advance to "beginning of next lexical block" pointer to 1 less than the current position of the scan pointer; go to JUMP
Cont:	Advance the scan pointer by 1;
Jump:	Go to 2, 3, 5, 12, 126, 25,. 78, 8, or 10, depending on the current lexical state.
Endl:	Advance the scan pointer by 1 and go to END.
Endx:	Reduce the scan pointer by 1, set state = 12, and go to END
Endy:	Enter the immediately preceding block of characters in the symbol table, flagging it as a parameter. Advance the "beginning of next lexical block" pointer to the present value of the scan pointer. Set generator switch = 1, state = 10, and go to CONT.
Endz:	Enter the immediately preceding block of characters in the symbol table, flagging it as a parameter. Set generator switch = 1, state = 10, and go to ADV.
Offdef:	Enter the immediately preceding quantity into the symbol table, dropping the parameter flag attached to this quantity if set. Go to SKIP.
Defsw:	Advance the "beginning of next lexical block" pointer by 2.

(continued)

If the generator switch is 1, set it to zero, set state = 10, and go to CONT.
If the generator switch is 0, set state = 8 and go to CONT.

Substit: Copy the string of lexical atom pointers referenced by the parameter symbol from the parameter values buffer to the output buffer. Go to ADV.

Go25: Set state = 25; go to CONT.

Go12: Set state = 12; go to CONT.

Go2: Set state = 2; go to CONT.

Go3: Set state = 3; go to CONT.

Go126: Set state = 126; go to CONT.

Go5: Set state = 5; go to CONT.

Go78: Set state = 78; go to CONT.

Skip: Advance 'beginning of next lexical block pointer' by 1; go to CONT.

The parameter-substituting lexical scan algorithm shown in Table VI may be regarded as a very simple symbolic macro expander. Of course, very much more general and powerful macro processor features might also be designed into a symbolic input program; at their most general, these would provide a fully recursive set of arithmetic and symbol manipulating routines executable at compile time using an interpreter which may very conveniently be combined with a compiler using the same lexical scan. This may be accomplished by providing a switch which will toggle all the generator routines comprising the compiler back and forth between a normal "compile" mode and a direct "interpretive" mode. In the compile mode a generator routine will, when called, emit a block of code; in the interpretive mode the same generator routine when called will execute the corresponding instructions directly. At a relatively small expense in additional programming, this scheme can provide for a very powerful system of compile time calculations and corresponding conditional macro expansions, written in what is substantially the language being compiled. A discussion of this idea would take us rather far from the questions which we have studied in the present chapter; consequently, we choose not to pursue this line of thought here.

We conclude the present chapter with a number of remarks concerning departures from the strict "state transition" style described above which may be used to increase the power and efficiency of lexical scan programs.

1. Internal flags or switches may be maintained, and may be used to govern the action of the lexical scan at particular points in its course. Table VI above has already shown the use of this feature. More generally, if the state transition table rows corresponding to two or more lexical states are identical except for the actions initiated by a few characters, we may combine the two states as sub-states of a single nominal state, and correspondingly combine two rows of the state table into a single row, introducing a state flag to distinguish the two sub-states within the single nominal state into which they have been combined. Input characters which would be treated differently in the two states which have been combined will cause the distinguishing state flag to be consulted and the correct action as shown by this flag to be taken. Using such a technique we could, for example, combine the two rows IRH and IR of Table I into a single row. To this end we could, of course, employ a flag J whose values 1 and 0 distinguish what in Table I is the IRH state from what in Table I is the IR state.

2. Various useful internal counts can be maintained and incremented or decremented from time to time depending on the state of the scanner and the input character to be processed. This device enables the lexical scanner to distinguish, e.g., balanced-parenthesis substrings within an input string, a possibility useful in the pre-processing of various languages.

3. A sufficiently powerful lexical scanner can, in certain cases, be used to perform simple syntactic operations, thereby enhancing the efficiency of the full parse which is to follow. For example, a lexical scanner which pre-scans

an entire "statement" in advance of syntactic analysis may be used to determine the type of the statement, thereby simplifying the subsequent syntactic analysis and making it more efficient.

4. Special processes enhancing efficiency can be built into a lexical scanner where appropriate. For example, incorporation into a lexical scanner of a trick machine-code sequence forbypassing machine words full of blanks, or more generally for locating the next following non-blank character in an input string, will in many cases yield significant reductions in total compilation time.

5. Certain syntactically irritating special conventions featured in particular languages, as for example the continuation-card convention in FORTRAN, can be handled by a lexical pre-scanner and thus eliminated as a cause of syntactic complication.

Note finally that a lexical scanner will normally deal with its input string one character at a time; thus its "front end" can be a "next character" routine. In particular, no large buffer for input string storage is logically required.

CHAPTER 4. DATA DIRECTED PARSING METHODS

1. The General "Bottom-Up" Parsing Method.

The methods of syntactic analysis discussed in Chapter 2 are often called "top-down" methods. The parsing strategy which these methods employ is essentially goal directed. That is, starting with an initial goal, that of constructing a sentence valid according to the BNF grammar at hand, the parsing program builds up a stack of sub-goals. As a sub-goal is pursued, additional sub-goals are generated and recursively added to the stack. Before the algorithm will return to any higher level goal, all parts of all sub-goals generated by this higher level goal must be completely and successfully accomplished.

The 'bottom-up' method is, in comparison, data directed. The action of a bottom-up parsing algorithm is at each point guided by the characters constituting the input string. This string, and more precisely a particular character in the string, is taken up for analysis. Starting with this string and character, the parsing program attempts to find a production of the grammar by reversing which one may combine the given character and a number of its adjacent characters into an intermediate symbol of the grammar, which, according to the given production, might have generated the characters in question. When such a substring is found, the algorithm condenses it, replacing the substring by the single intermediate symbol which has generated it, and thus replacing the given input string by a (generally shorter) string containing both terminal and intermediate symbols of the grammar. This process of condensation is applied iteratively; the over-all goal of the parsing program is of course the complete condensation of the initial sentence into a single character representing the root type of the grammar at hand. Note that in the bottom-up analysis

method no detailed set of intermediate goals is maintained; the inverse of any particular grammatical production is applied when possible.

Since for a general Backus normal form grammar any particular application of a grammatical production in the reverse direction may be in error, that is, may not lead to a full parse of the input sentence, the bottom-up method, like the top-down method, must in general be prepared to "back up." In consequence, the parsing program must keep track of the sequence of tentative combinations made during its attempt to parse a string, in order that any step taken may be undone if this proves necessary in the light of string elements subsequently encountered. As a general method, therefore, the bottom-up parsing strategy has no real advantage over the top-down procedures developed in Chapter II. However, for certain particular types of unambiguous grammars, in which the reverse-direction applicability of a given grammatical production may be inferred with certainty from the local structure of the input string, the bottom-up method can attain real advantages of speed and efficiency as compared to the top-down method. In particularly favorable cases the bottom-up method will permit a language to be parsed by the iterative application of condensing transformations of the kind alluded to above without any back-up at all. Most of our attention in the present section will be directed toward those particular languages, the so-called bounded-context and precedence languages, for which a bottom-up syntactic analysis strategy leads to an efficient, unambiguous, and complete parse of every possible input sentence. We begin, however, by discussing a form of the bottom-up parsing algorithm which is applicable to general Backus normal form grammars.

A bottom-up parsing algorithm is, at each of its stages, concerned with a string. This string is given initially as a string of terminal characters to be parsed. A parsing

algorithm attempts, as indicated above, to condense the string with which it is concerned, stage by stage, through a sequence of shorter strings involving both terminal and non-terminal symbols of the grammar, until the original string has been condensed to a single character. We will find it convenient in discussing our general bottom-up parsing algorithm to assume that, at all times, the string is divided into two parts, a front and a back, which are kept in separate pushdown stacks which we will call stack1 and stack2. Stack1 will always contain the front part of the string being parsed, the first character of the front part being "deepest" in the stack. Stack2 will contain the back part of the string being parsed, the last character of the string being deepest in stack2.

As the parsing algorithm scans forward past the various characters in the input string, these characters will be moved from the top of stack2 to the top of stack1; scanning in the reverse direction will move characters back from stack1 to stack2. The use of two stacks in this fashion makes the insertion of characters into a string and their removal from the string quite convenient.

In searching through a string to be parsed for a point at which a given production of the grammar may be applied in the reverse direction, we shall assume, as a matter of convenient standardization, that only substrings whose final character is the top character on stack2 will be condensed. This convention establishes an orderly left-to-right flow of the parsing process. When a condensation is performed, we replace the top character on stack2 by that intermediate character which, according to the Backus normal form grammar with which we are working, expands directly into the substring encountered. Condensation of a substring will also involve the removal of the top few characters from stack1; these are the characters which immediately precede the top character on stack2 in the string under analysis. The characters

constituting the condensed substring become nodes in the collection of syntax tree fragments which the parse is developing. Management of these tree fragments is most conveniently accomplished by the use of yet another stack, which we will call stack3. This constitutes a useful basis for the "back-up" operations, which, as we have already noted, will occasionally be necessary. The intermediate character generating the most recently condensed string is, as noted above, to be placed on stack2 when a condensation is performed; this intermediate character may then be associated with a pointer indicating the position of the top of stack3 at that moment when the condensation is performed. A second field indicating the number of characters that have been condensed should also be associated with the same intermediate character. If these two pointers are placed on stack3 along with every character transferred to this stack, stack3 will always contain a complete description of the set of tree fragments constructed by the parse. When the parse comes to a successful conclusion, stack3 will contain a complete representation of the parse tree for the initial sentence.

By associating the pointer and block size fields described above with each character of a string to be parsed, we arrive at a basic data item of the following form:

(3)

character	stack 3 pointer	stack3 block size

It is most convenient, in describing a bottom-up parsing algorithm, to assume a serial enumeration of the productions of the Backus normal form grammar, and to assume that all the possible alternative definers of a given intermediate grammatical type are separately enumerated. That is, instead of writing

(1) ⟨term⟩ = ⟨factor⟩ | ⟨factor⟩ * ⟨term⟩ | ⟨factor⟩**⟨term⟩

in the conventional Backus style, we shall prefer to write

(2) 1. ⟨term⟩ = ⟨factor⟩
2. ⟨term⟩ = ⟨factor⟩ * ⟨term⟩
3. ⟨term⟩ = ⟨factor⟩ ** ⟨term⟩ .

Such a list, extended to all the productions of a grammar, provides a convenient serial enumeration by which we may refer to particular productions as necessary.

The complete working of a general bottom-up parsing algorithm requires the addition of a fourth field to the three fields shown in (3). This fourth field is necessary in connection with the back-up procedure which a general parsing algorithm must occasionally employ. If a given condensation turns out in the subsequent course of a parse to have been inappropriate, it must be undone, this requires the return from stack3 of previously stacked characters. The returned characters are of course to be put back on stack1 and on the top of stack2. After this back-up operation has been performed, the parsing algorithm will again attempt to proceed; of course the algorithm is not to repeat any condensation which has already been attempted. For this reason, we shall associate with each character the number of the last Backus production whose inverse has already been applied to condense a substring ending with the given character. Equivalently and somewhat more conveniently, we may associate with each character the number of the first production in the Backus grammar which has yet to be applied to a substring of characters ending with the given character. Adding this fourth field to the three fields shown in (3), we arrive at a basic data item of the following form:

(4)

character	stack3 pointer	stack3 block site	next alternative production

This data item contains all of the necessary fields for the operation of the full bottom-up parsing algorithm whose details are given in Table I below. A number of additional comments will make the structure of the algorithm shown in Table I somewhat more plain. At the beginning of a parse, stacks 1, 2 and 3 are initialized. Stacks 1 and 3 are empty. Stack2 contains a collection of items of the form shown in (4), representing the atomic words of the initial input string, in reverse order. All the "pointer" fields for these items are set to zero; all the "block size" fields are set to zero.

In each case the "next alternative production" field is set to 1, that is, is set to reference the first porduction of the Backus grammar defining the parse.

Table I. Details of the General Bottom-Up Parsing Algorithm.

initially:	stack2 contains input string, first element on top all next-alternative pointers set to 1 all stack3 reference pointers set to 0 stack1 and stack3 are empty	stack2 contains input string, first element on top for j = 1 till p2-1 do: (next(stack2(j))=1; point(stack2(j))=0) p1=0; p3=0
advance:	take top element of stack2 n = next production to try	n = next(stack2(p2-1))

areprodns:	if n exceeds total number of
	productions, go to fail.
	if n-th production matches
	substring ending with given
	character, go to succeed.
	else set n=no. of next
	production in grammar ending
	with same character, and go
	to areprodns.
		if (n.gt.maxprodns)go to fail
		
		if (matches(n,p1,p2))go to succeed
		
		n = nextry; go to areprodns

succeed:	set next production field of top of stack2 to number of next production ending with same character. move top of stack2 and appropriate number of elements of stack1 to stack3. insert stack3 top pointer in appropriate field of inter- mediate character. place new intermediate symbol at top of stack2. if stack1 is empty and stack2 contains only a single character, parse is complete.	p2 = p2-1; next(stack2(p2))=nextry stack3(p3)=stack2(p2); p3=p3+1 for k = 1 till number do: (p1=p1-1; stack3(p3)=stack1(p1); p3=p3+1) point(intermed) = p3 stack2(p2) = intermed; p2=p2+1 if((p1.eq.1) .and.(p2.eq.2)) return successfully

	else go to advance to continue parse.	go to advance
fail:	insert n in next field of item at top of stack2 move item from top of stack2 to top of stackl. if stack2 is not empty go to advance; else enter backup loop.	p2=p2-1; next (stack2(p2)) = n stackl(p1) = stack2(p2); p1=p1+1 if (p2.gt.1) go to advance

backup:	if stackl is empty, parse has failed.
	if top element of stackl is not intermediate symbol, reinitialize its next production field to 1, and move it to stack2; then go to backup. Else re-expand intermediate symbol as follows:
		if(p1.eq.1) return unsuccessfully
		
		p1=p1-1;
		if((point(stackl(p1)).ne.0)go to expand
		next(stackl(p1))=1
		stack2(p2)=stackl(p1); p2=p2+1
		go to backup

expand:	return appropriate number of elements from stack3 to stackl; and one additional element to stack2. then go to advance.	k = numb(stackl(p1)) for j=1 till k do: (p3=p3-1; stackl(p1)=stack3(p3); p1=p1+1) p3=p3-1; stack2(p2)=stack3(p3); p2=p2+1 go to advance

---end---

In advancing to the right, the parsing algorithm takes up the top character on stack2 for consideration. This character will be part of a stacked data item having the additional fields shown in (4). In particular, the next production in the total list of productions constituting the Backus grammar defining the parse will be specified by the last field of this data item. If the number contained in this field exceeds the total number of productions in the grammar, then it is certain that every possible way of condensing a substring terminating at the particular character under consideration has already been tried. When this circumstance arises, the given character must merely be transferred from stack2 to stack1; the parsing algorithm must then proceed to consider the next character which rises to the top of stack2. If, on the other hand, the integer contained in the final field of the data item (4) does not exceed the total number of productions in the grammar, then it is possible that the production referenced by this integer is applicable to that substring of the string at hand which consists of the character on the top of stack2 and it's immediately preceding characters; these are, of course, the top few characters on stack1. The parsing program then checks to see if this is indeed the case. If it is the case, a condensation is performed. On the other hand, if condensation according to the specified production is impossible, the production number contained in the final field of (4) is advanced, and the next alternative production is tried. When an applicable production is found, the characters condensed are removed from the top of stack1, and the character at the top of stack2 is removed and replaced by the single intermediate character which, according to the production that has been used, can generate the newly condensed substring. All the characters removed are placed in order on the top of stack3, and the second and third fields of the items inserted at the top of stack2 are set. The parsing algorithm proceeds iteratively

in this fashion until either the initial character string has been condensed to a single character, in which case the parse is successful, or until an end-of-string condition not permitting any further condensations is encountered. If, in this latter case, the initial string is not ungrammatical, it follows that at a certain stage of the attempted parse a condensation which is not a part of the true parse tree of the input sentence has been made. To discover the point at which this fatal error has occurred, the parsing algorithm must undo as many of its preceding decisions as necessary. This is accomplished by reviewing the characters which occur in the condensed input string, in reverse sequence, and by re-expanding any intermediate character which corresponds to a substring saved on stack3 into its original form. When such a re-expansion of or "back-up" operation has been performed, the fourth field of the data item standing at the top of stack2 references a condensing production which has not yet been tried. The parsing algorithm then proceeds to try this production; the iterative use of such a back-up and re-try procedure will lead the parsing algorithm down every possible parse path and will, if the initial sentence is in fact grammatical, eventually bring the algorithm to discover a full parse tree.

Table I gives all details of the parsing algorithm which we have described informally in the last few paragraphs.

The following comments will aid the reader to understand the algorithm given in Table I. In the first place, it is to be noted that we have written out the algorithm twice, in two parallel forms. The left hand column of Table I contains a relatively informal description of the various steps of our bottom-up parsing algorithm. The right hand column in Table I shows the same steps represented in a somewhat more formal way, and in a hypothetical language close to ALGOL or FORTRAN. The algorithmic steps shown in the left hand column may therefore be regarded as explanatory

comments pointing out the over-all significance of the corresponding steps found in the right hand column of Table I. In the algorithm of Table I, stack1, stack2, and stack3 are regarded as arrays. The quantities p1, p2 and p3 are integers defining the first empty push-down entry in stack1, stack2 and stack3 respectively. Thus, the top entry in the push-down stack1 is always the entry referenced by p1-1. Each stack entry is assumed to consist of the four fields shown in (4) above. Where necessary, the "next alternative production field" is referred to formally by the subfield function "next." Similarly, the "stack3 pointer field" is referred to as necessary by the subfield function"point"; while the "stack3 block size field" is referred to as necessary by the sub-field function "numb."

The algorithm shown in Table I uses a principal subroutine called matches whose three arguments are, respectively, the number n of a particular and two pointers p1 and p2, one to the top of stack1, the other to the top of stack2. This subroutine is assumed to examine the string S under analysis, specifically the substring ending with the character at the top of stack2, to see if the n-th production can be applied in the reverse direction to achieve a condensation of S. If condensation is possible, the value "true" is to be returned; if condensation is impossible the value "false" is to be returned.

We also assume that when the subroutine matches is called, the values of three additional quantities are set. These three quantities are called 'nextry', 'number', and 'intermed' in the algorithm shown in Table I. 'Nextry' is assumed to be the number of the next production in the grammar having the same terminal character as production number n, or, if no such production exists, to be equal to the total number of productions, plus one. The quantities 'number' and 'intermed' are required only in case the attempted condensation succeeds. 'Number' is assumed to be set equal

to one less than the total number of characters involved in the successful condensation. 'intermed' is assumed to be an item of the form shown in (4), in which the character field is pre-set to show the intermediate character which generates the sub-string which is to be condensed, in which the block size field is appropriately set, (i.e. is set to equal to the quantity 'number'), and in which the 'next alternative production' subfield is pre-set to the value 1. The algorithm itself then sets the 'stack3 pointer field' appropriately, using the known value of the quantity p3.

The subroutine 'matches' used in the algorithm of Table I can be simple and efficient if each of the productions constituting the Backus normal form grammar defining the required parse is maintained in the form shown below:

(5)

c_1	c_2	...	c_m	c_o	k	m-1	1

In (5) we assume that c_1, c_2,..., c_m are the successive characters constituting the right hand side of a grammatical production, that c_o is the intermediate character occuring on the left hand side of the production, that k is an integer identifying the next production in the grammar which has the same terminal character c_m as the production (5), and that m-1 is one less than the number of characters $c_1,\ldots,c_m$. Note then that the last four fields in the data structure shown in (5) will constitute an item of precisely the form (4), which the subroutine matches may return without modification as the value of the quantity 'intermed'. The subroutine 'matches' will function most efficiently if it is also provided with a table indicating, for each character c, the number of the first production in the grammar possessing c as right hand terminal character.

An appropriate input routine will make the use of an algorithm of the form shown in Table I quite convenient. This input routine may, for example, read a set of cards containing a grammar specified in Backus normal form and

written in any convenient notation. The input routine can then sort the productions of the grammar into convenient order and establish a table of productions in which each individual production is represented by a set of fields of the form shown in (5). When the input routine has completed these initializations, the process defined by the algorithm of Table I may be called to parse sentences according to the grammar just read.

2. Bounded context grammars and associated deterministic scans.

The back-up procedure employed by the general bottom-up algorithm shown in Table I is rather inefficient; note in particular that the algorithm will always fail to detect the need to back up until it has scanned its string all the way through to the final character. The top-down algorithms discussed in Chapter II will normally begin to correct a developing parse error at a much earlier stage. It is therefore important, in order to improve the efficiency of algorithms of the general type shown in Table I, to avoid backing up as much as possible. This can be done particularly successfully if the parsing algorithm is modified so as to perform a condensation only when a local context in the string being surveyed is such as to insure that such a condensation necessarily corresponds to the true structure of the syntax tree of the string under analysis. Of course, not every grammar will permit a simplified procedure of this kind to be employed. However, those grammars whose languages may be parsed by such a "no back-up" procedure can be handled particularly efficiently.

Given a grammar in Backus normal form, it may be the case that in every possible parse of a string of the particular form

(6) $$S_\ell\ p_1 \cdots p_{\ell'}\ c_1 \cdots c_n\ q_1 \cdots q_r\ S_r\ ,$$

the substring $c_1 \ldots c_n$ must necessarily be derived from a particular intermediate character x. In such a context, the condensation of (6) into

(7) $\quad S_\ell \; p_1 \ldots p_\ell \; x \; q_1 \ldots q_r \; S_r$

can never be in error. Thus, for example, in various standard grammars for the ordinary arithmetic expression the sequence of elements

(8) $\quad \ldots + \langle \text{element} \rangle * \langle \text{factor} \rangle + \ldots$

can only be derived from the condensed sequence

(9) $\quad \ldots + \langle \text{factor} \rangle + \ldots$

via the reverse of the Backus production

(10) $\quad \langle \text{factor} \rangle = \langle \text{element} \rangle * \langle \text{factor} \rangle$

If, in the context (6), the substring $c_1, \ldots, c_n$ can only be derived from the intermediate symbol x, then $p_1, \ldots, p_\ell$, $c_1, \ldots, c_n$, $q_1, \ldots, q_r$ is called an (ℓ, r)- unambiguous context. Using the existence of such contexts, the parsing algorithm of Table I may be modified to eliminate back-up as follows. First, we assume that a complete collection of all (ℓ, r)-unambiguous contexts for which $\ell \leq$ some upper bound ℓ_0, and for which $r \leq$ some upper bound r_0, is available. Given this table of unambiguous contexts, we revise the algorithm of Table I so that before applying any condensation, the parsing program will verify that the context in which the sub-string to be condensed appears is one of the available list of unambiguous contexts. By restricting the attempted set of sub-string condensations in this manner, we insure that the parsing algorithm need never undo any condensation which it has performed. Unfortunately, for certain grammars, the full set of unambiguous contexts will be 'thin' in the sense that the bottom-up parsing algorithm, restricted in this way, will be unable to condense every grammatical string to a single character. However, if a grammar has the

special property that the restricted bottom-up parsing algorithm which we have just described will succeed in condensing every grammatical string, we call the grammar an ℓ_0, r_0 bounded context grammar.

In Chapter V, we shall show that not every context-free grammar has this bounded context property. We shall at that time discuss the relation between general context free grammars and the bounded context grammars as a sub-class in somewhat more detail.

In the present chapter however, we will be interested not in fundamental theoretical discussions, which we postpone to Chapter V, but in relatively pragmatic matters. In particular, we shall, in the following paragraphs, develop various useful, algorithmically verifiable sufficient conditions which guarantee that a grammar is a (k,ℓ)-bounded context grammar with relatively small k and ℓ.

We begin by defining three relations

$$a \cdot\!> b, \quad a \doteq b, \quad a <\!\cdot b$$

between symbols of a grammar, terminal or non-terminal. These relations are defined as follows:

i) The relation $a \cdot\!> b$ holds if there exists a grammatical string in which a and b are successive elements and in whose syntax tree, a is the final element of a sub-tree.

ii) The relation $a \doteq b$ holds if there exists a grammatical string in which a and b are successive elements, and in whose syntax tree both a and b are immediately derived from a common node.

iii) The relation $a <\!\cdot b$ holds if there exists a grammatical string in which a and b are successive elements and in whose parse tree there exists a node μ with the following property: a is immediately derived from μ, while b is indirectly derived from μ.

Note that, if a and b are consecutive characters in any grammatical string, at least one of these three relations must necessarily hold. Indeed, let τ be the parse tree of the given string. Consider the smallest sub-tree ν to which both of the successive characters a and b belong, and let μ be the root node of this sub-tree. Then either both a and b are directly derived from μ, in which case (ii) holds; or a but not b is directly derived from μ, in which case (iii) holds, or a is derived from sub-tree σ of ν which, by definition of ν, cannot include b; so that in this case, a is necessarily the terminal character of the sub-tree, and (i) holds.

We call our context-free grammar a precedence grammar if the three relations $a \cdot\!> b$, $a \doteq b$, $a <\!\cdot\ b$ are mutually exclusive for every pair of symbols a, b of the grammar. Suppose that the grammar in which we are interested is a precedence grammar, that we are given a string S which is grammatical according to this precedence grammar, and that $a_1 \ldots a_n$ is the left-most sequence of characters in the string S which is directly derived from a minimal sub-tree of the parse tree of S. In this case, it follows immediately from definition (ii) that we must have $a_1 \doteq a_2 \doteq \ldots \doteq a_n$. Let a be the character of the string S which immediately precedes a_1, and let b be the character of the string S Which immediately follows the character a_n. Then it is clear from definition (i) that $a_n \cdot\!> b$. Moreover, the relationship $a \doteq a_1$ is impossible, since some other relation holds in the parse tree of S; while $a \cdot\!> a_1$ is impossible since $a_1 \ldots a_n$ is the left-most sequence of characters in S derived from a subtree of S.

Thus, we must have $a <\!\cdot\ a_1 \doteq \ldots \doteq a_n \cdot\!> b$.

Moreover, since $a_1 \ldots a_n$ is the left-most sequence of characters of the string S derived from a minimal sub-tree, no character preceding a_1 can be the terminal character of the sub-tree of the parse tree of S. Thus, the relationship $c \cdot\!> d$ can never be satisfied by an adjacent pair of characters c,d of the string S which preceed a_1. Therefore, for the

special class of precedence grammars, the first sub-string $a_1 \ldots a_n$ which is to be condensed according to the parse of the string S is uniquely defined as the left-most string for which the sequence of relations $a \lessdot a1 \doteq \ldots \doteq an \gtrdot b$ holds. If we assume of our grammar, in addition to the standing assumption that it is a precedence grammar, that distinct intermediate symbols never have the same definer, then the condensation operation

(11) $\quad a_1 \; a_2 \ldots a_n \longrightarrow \mu$

affecting the sub-string $a_1 \ldots a_n$ is uniquely defined in its context. It follows at once that every precedence grammar is a (1,1)-bounded context grammar in the sense of the final paragraphs of the preceding section.

The above discussion may be generalized in a very useful way. If σ is any string of terminal or intermediate symbols of a grammar, we may introduce the three following relations, corresponding to the relations (i), (ii), (iii) defined above:

i') We write $a > \sigma b$

if there exists a grammatical string in which the substring... σa b... occurs, and in which a is the final element of a sub-tree of the parse tree.

ii') We write $a = \sigma b$

if there exists a grammatical string in which the sub-string ...σab... occurs, and in which a and b are both immediately derived from a common node of the parse tree for the string.

iii') We write $a < \sigma b$

if there exists a grammatical string in which the sub-string ...σab... occurs, and in which a is immediately derived from some node μ of the parse tree, while b is indirectly derived from the node μ.

Let S be a grammatical string, and let a_i and a_{i+1} be successive characters in the string S. Let σ_i be the substring of S consisting of a few characters immediately preceding a_i. Then one of the three relations $a_i <_{\sigma_i} a_{i+1}$, $a_i =_{\sigma_i} a_{i+1}$, or $a_i >_{\sigma_i} a_{i+1}$ must always hold. The proof of this assertion is similar in all regards to the proof of the corresponding assertion in the more special case treated above; cf. the two paragraphs following definitions (i), (ii), (iii). We call a grammar a <u>generalized k-precedence grammar</u> if for each string σ of characters of length k, the relationships $a <_\sigma b$, $a >_\sigma b$, and $a =_\sigma b$ are mutually exclusive. (In making this definition, we intend to allow, among the strings σ appearing in the definition, strings consisting actually of a smaller number of characters than k, but nominally padded out to length k by whatever number of prefixed "beginning of string" characters are necessary). Suppose then that for each character a of a string S to be parsed, we let σ_a be the immediately preceding k character sub-string, supplemented as indicated with prefixed beginning-of-string characters as necessary. We may deduce just as in the special case explicityl treated above that if $a_1 \ldots a_n$ is the left-most sub-string of S belonging to a minimal sub-tree of the parse tree of S, then $a_1 \ldots a_n$, together with the immediately preceding character a and the immediately following character b, constitutes the first occurrence in the string S of the set of relationships

$$(12) \qquad a <\cdot_{\sigma_{a_1}} a_1 \doteq_{\sigma_{a_2}} a_2 \doteq_{\sigma_{a_3}} \cdots =_{\sigma_{a_n}} a_n \cdot>_{\sigma_b} b$$

Assuming once more that no two intermediate symbols of the grammar exist which have identical definers according to productions of the grammar, it follows at once that the condensation operation which applies to the string $a_1 \ldots a_n$ in the parse tree of S is unique. It follows that every generalized k-precedence grammar is a (k,1)-bounded context grammar.

Table II below gives the details of an algorithm for parsing such a grammar.

The following comments will help the reader understand the algorithm shown in Table II. Note, to begin with, that, as in Table I, we have written out the algorithm twice in two parallel forms: a left hand column of informal description and a right hand column showing the same steps in a somewhat more formal way. As in Table I, each step shown in the left hand column may be regarded as an explanatory comment defining the over-all significance of the corresponding step in the right hand column of Table II. The algorithm makes use of four push-down stacks, called stack1, stack2, stack3, and stack4. The first three of these stacks have essentially the same significance as the three stacks used in the algorithm of Table I. Stack4, however, is used to contain a chain of pointers to positions in stack1; the use of these pointers will be explained shortly. The position pointers for stacks 1, 2, 3, and 4 are called p1, p2, p3, and p4 in the right hand column of Table II. Since the algorithm shown in Table II, like most algorithms for parsing bounded context grammars, does not involve any back-up process at all, the fourth field of the data item shown in(4) is unnecessary. Thus, we may take the data items contained in stacks 1, 2 and 3 to have the simpler form shown in (3) of the preceeding section. As in Table I, the "stack3 pointer" field in a data item will be referenced by the subfield-function "point."

The over-all flow of the algorithm is as follows. Starting with the first character of the input string, the parsing program scans through the characters of the input string, in turn, transferring them from stack2 to stack1. As each character is examined, the algorithm determines whether it stands in the relationship $a \gtrdot_{\sigma} b$, $a =_{\sigma} b$ or $a \lessdot_{\sigma} b$ to the preceding character a, and to the immediately preceding string σ of characters. This basic operation is

performed by the subroutine "compare" invoked by the algorithm shown in Table II. This subroutine returns a flag, called n in Table II, and having one of the four values 1, 2, 3, 4. The first three values of the flag n correspond to the three relations $a \gtrdot_\sigma b$, $a =_\sigma b$, $a \lessdot_\sigma b$. The value n = 4 indicates that b stands in none of these three relations to the preceding character a of the input string, and hence can only occur if the input string involves a syntax error. Once the relationship between the character b being examined and the immediately preceding character has been determined, a corresponding action is taken. If $a \gtrdot_\sigma b$ holds, then a terminates a substring of the input string, for which condensation must be possible. In this case, the parsing program (after checking to see whether the parse is in fact complete) condenses an appropriate portion of the input string terminating with the character a, and replaces it with the intermediate symbol of the grammar which is the unique immediate generator of this input string. The beginning of the portion to be condensed is the last preceding character d which stands in the relationship $c \lessdot_\tau d$ to tis predecessor c. The position of d in the input string will already have been recorded at the top of stack4 in a manner that we shall describe shortly. Thus the condensing procedure, which is indicated in Table II as a call to a functional subroutine 'condense,' need merely remove the top element from stack4, and use the pointer which it contains, together with the current value of the pointer p1 to the top of stack1 to determine the portion of the input string which is to be condensed. If such condensation is impossible, that is, if there exists no production whose right hand side is identical with a given substring, then the input string is ungrammatical. Correspondingly, it is assumed in the algorithm shown in Table II that, when the subroutine "condense" is called, it returns a flag "success" which is used by the algorithm of Table II to determine whether or not an error

of this type has occurred.

Successful call to "condense" returns a data item "intermed" representing the intermediate symbol from which the condensed substring is generated. It is also assumed that the subroutine 'condense' transfers the appropriate number of elements from the top of stack1 into stack3 when a condensation is performed. Following on successful condensation of a substring, the "stack3 pointer" field of the returned intermediate element "intermed" is set to the appropriate value, the intermediate element transferred to the top of stack2, and the parsing program, starting from this new position, continues iteratively as before.

If comparison of a given input string element b with its immediately preceding character a reveals that the relation $a =_{\sigma} b$ holds, a different procedure is followed. In this case, a and b will eventually form part of a string to be condensed into a single intermediate symbol. For this reason, the parsing program merely passes over the symbol b, transferring it from the top of stack2 to the top of stack1. Finally, in case the character b at the top of stack2 is related to the preceding element a by the relationship $a >_{\sigma} b$, it follows that b initiates a new subsection of the input string. This subsection, when complete, will be condensed into a single intermediate character. In order to avoid recalculating the beginning of sections of this sort repeatedly, the parsing algorithm can stack the location of every character b related to its preceding character a by the relationship $a <_{\sigma} b$ on stack4. When a section to be condensed is complete, the top initial address on stack4 is removed, and the beginning address of the next most inclusive section eventually to be condensed is revealed.

Table II. Parsing Algorithm for Generalized Precedence Grammars.

initially:	stack1 contains a single beginning-of-string character	p1=1; stack1(p1)=beginstring; p1=p1+1
	stack2 contains string to be parsed, first character on top, terminated by end-of-string character	stack2 contains input string terminated by end-of-string character, first element on top
	all stack3 reference pointers set to zero	for j=1 till p2-1 do: (point(stack2(j))=0)
	stack3 and stack4 are empty	p3=1; p4=1
compare:	compare top character of stack1 with top character of stack2, using preceding context;	n=compare(p1,p2)
	depending on result of comparison, go to (bigger,same,smaller,error)	go to(bigger,same,smaller,error)n
bigger:	stack current value of p1 on stack4	stack4(p4) = p1; p4=p4+1
same:	move element from stack2 to stack1; go to compare	p2=p2-1; stack1(p1)=stack2(p2); p1=p1+1; go to compare
smaller:	if only 1 character besides begin- and end-of-string characters are left, parse is complete	if((p1.eq.2).and.(p2.eq.1)) return successfully
	attempt to condense portion of stack1 from character referenced by stack4 to top; if this fails, a syntax error has occurred	p4=p4-1; intermed=condense(stack4(p4),p1); if (not.success)go to error
	insert stack3 pointer field in new intermediate symbol	point(intermed) = p3-1
	place new intermediate symbol on top of stack2	stack2(p2) = intermed; p2=p2+1
	go to compare to continue processing	go to compare
error:	return unsuccessfully	return unsuccessfully

The algorithm shown in Table II examines any given symbol of an initial input string for its syntactic context only once. The parsing program described by the algorithm of Table II is consequently extremely rapid.

If the handle of a parse tree is defined as the left-most minimal sub-tree, then the algorithm that we have described will first discover and condense the handle of a parse tree into a single character; and proceeding iteratively will eventually condense the parse tree into a single node. At any rate, the order of condensation is left to right, the left-most minimal subtree always being condensed first. Since the order of condensation is entirely predictable, a generation procedure may readily be associated with the parsing algorithm of Table II. Code generation procedures may conveniently be inserted as subroutines to be called by the function "condense" of Table II. Each call to this subroutine corresponds to the condensation of a particular sub-tree. Instead of transferring condensed symbols onto stack3, the condensation subroutine, having decided on the particular production which is applicable, may call a generator routine associated with this production. A reasonable metalinguistic convention expressing the necessary semantic specifications might consist, for example, in the association, with each of the productions constituting a context-free grammar, of the name of a generator subroutine to be called by the "condense" subroutine when that production is applied in the reverse direction. Such generator names can, by an evident extension of the Backus metalanguage, be appended to the end of the definer constituting the right hand side of a syntactic production, following some special symbol or mark used to indicate the end of the definer.

The following simple scheme for the generation of compiler diagnostics is applicable to precedence parses of the type shown in Table II. At the point at which a transfer to the algorithmic label "error" is made by the parsing program, all those characters of the initial input

string which have been condensed into intermediate symbols of the input string may be regarded as constituting a subsection S_1 of the input string whose parse is probably correct. The parsing algorithm can then flag the characters belonging to the string S_1. Flagging these characters will show the approximate point of the initial input string at which a syntax error has occurred. The standard diagnostic procedure thus specified is general and applies without any special adaptation, to an arbitrary precedence grammar. On the other hand, since a top-down procudure will always be aware of a local goal, and can report the failure of this goal as a part of the diagnostic information it provides to the compiler user, the diagnostics obtained by the procedure just outlined may be cruder than the diagnostics which can be obtained by a top-down parsing procedure.

3. Storage of precedence information for a bounded context parse.

We now turn to consider the form in which the tables of precedence relations to be used by the subroutine "compare" called in the algorithm of Table II are to be initialized and maintained. These tables may be taken to consist of two basic portions, the first of which is a simple (character pair) precedence table, and the second of which is a supplementary (extended context) precedence table. The first of these tables may be regarded as a two-dimensional array, which, for every pair a, b of characters, specifies a value which is either 1, 2, 3, 4 or 5.

If the value 1 is entered in the first of these tables against a pair a,b of characters, it signifies that the relation a $\lessdot$ b holds, and that neither of the relations a $\doteq$ b or a $\gtrdot$ b holds. If the value 2 or 3 is entered against the pair in this same table, it signifies, in a similar way, that the relationship a $\doteq$ b (resp. a $\gtrdot$ b) holds,

and that no other of the three relations $a \lessdot b$, $a \doteq b$, $a \gtrdot b$ holds. If the value 5 is entered against the pair, it signifies that none of the three preceding relations holds. This value is equally decisive for the comparison routine, since, if the comparison routine is called upon to compare two successive characters a, b during a parse, and if none of the three preceding relations holds, then a syntax error is implied. Finally, if the value 4 is entered against the pair a,b of characters in the first precedence table, it indicates that at least two of the relations $a \lessdot b$, $a \doteq b$, $a \gtrdot b$ hold. In this case simple (pair-wise) precedence is insufficient to determine the parse, the grammar is necessarily a generalized precedence grammar rather than a simple precedence grammar, and the supplementary tables must be used.

In this latter case, the "compare" subroutine of the parsing algorithm of Table II must proceed as follows. The string σ is first set equal to that character in the full string being parsed which immediately precedes the character a. The triple σ, a, b of three characters is then looked up in the supplementary precedence table, which, for the sake of definiteness, we may imagine to be maintained as a list accessed through a hash function. Either this triple will not be found at all in the hash table, or the hash table will assign one of the four values 1, 2, 3, 4 to the triple σ,a,b. These values have the same significance as before; 1 indicates that the relation $a \gtrdot_{\sigma} b$ holds, and that none of the relations $a =_{\sigma} b$ and $a \lessdot_{\sigma} b$ holds, etc. If an entry for the triple σ,a,b is found in the hash table, then the 'compare' routine may return a value directly upon finding this entry. If, on the other hand, no entry for σ,a,b is found in the hash table, then the "compare" routine must attempt to use at least one additional character of context. This is done as follows. Setting τ equal to that substring of the string being parsed which consists of the two characters immediately preceding a, the compare

routine looks for an entry in the supplementary precedence table (hash table) against the set of four characters τ,a,b. If an entry exists, the hash table will supply one of the values 1, 2, 3, 4, and the compare routine may proceed as indicated above. If, on the other hand, no such entry exists, then the "compare" routine must attempt to use the three characters of context immediately preceding the character a,... etc. An a prior decision as to the maximum integer k which is allowed in the consideration of the given grammar as a generalized k-precedence grammar will limit the maximum number of characters of context allowed. The integer k will normally be chosen quite small, generally equal either to 1 (the case considered by McKeeman, to whom the notion of generalized precedence grammar is due) or to 2. Larger values of k would require excessively large supplementary tables and are therefore of less interest.

For a generalized k-precedence grammar with k = 1 or 2, the simple precedence table described above will normally be as large as or larger than the supplementary table, for which reason a simple device, due to Floyd, for reducing the size of this table is of interest. This device is as follows: For each pair a,b of terminal or intermediate symbols, we keep a single authorization bit B(a,b). The value of this bit is 1 if exactly one of the three alternatives $a \lessdot b$, $a \doteq b$, $a \gtrdot b$ holds, but is zero if either none or more than 1 of these alternatives holds. Given the set of mutually exclusive relationships $a \lessdot b$, $a \doteq b$, $a \gtrdot b$, holding between those pairs a,b of characters of a given grammar for which B(a,b)=1, we attempt to find two positive, integer valued functions f(a), g(b), such that the relationship $a \lessdot b$ implies the numerical relationship $f(a) < g(b)$, such that the relationship $a \doteq b$ implies the numerical relationship $f(a) = g(b)$, and such that the character relationship $a \gtrdot b$ implies the numerical relationship f(a) $f(a) > g(b)$. A pair of functions satisfying these three conditions is said to have the precedence property.

If such a pair of integer valued functions f,g is available, then if B(a,b)=1, the "compare" algorithm can return the value 1,2, or 3 depending simply on whether $f(a) < g(b)$, $f(a) = g(b)$, $f(a) > g(b)$ which it finds. Of course, if B(a,b)=0, the "compare" algorithm must consult the supplementary precedence table described above to complete its action. The two-dimensional simple precedence table may be replaced by the union of a table of authorization bits and of two linear arrays, with consequent saving of space.

Next, we shall describe Floyd's algorithm for finding a pair of functions f and g having the precedence property. Define sequences of functions f_n and g_n inductively, by putting $f_1(a) = g_1(a) = 1$ to start the induction, and b defining f_{n+1} and g_{n+1} in terms of f_n and g_n as follows: if there exists any pair a,b such that $f_n(a) \leq g_n(b)$ and $a \cdot> b$, choose some such pair, put $f_{n+1}(a) = g_n(b)+1$, and put $f_{n+1}(a') = f_n(a')$ for all other a', and $g_{n+1}(b) = g_n(b)$ for all b. Otherwise, if there exists any pair a,b such that $f_n(a) \geq g_n(b)$ and $a <\cdot b$, choose some such pair, put $g_{n+1}(b) = f_n(a)+1$, put $g_{n+1}(b') = g_n(b')$ for all other b', and $f_{n+1}(a) = f_n(a)$ for all a. Otherwise, if there exists any pair a,b such that $a \doteq b$ while $f_n(a) \neq g_n(b)$, put $f_{n+1}(a) = g_{n+1}(b) = \max(f_n(a), g_n(b))$, and let $f_{n+1}(a') = f_n(a')$, $g_{n+1}(b') = g_n(b')$ for all other a' and b'.

If $f_{n+1} \equiv f_n$ and $g_{n+1} \equiv g_n$, then $f = f_{n+1}$ and $g = g_{n+1}$ are evidently a pair of functions with the precedence property. Conversely, suppose that there exists a pair of functions f,g with the precedence property. It is plain that both sequences $\{f_n\}$, $\{g_n\}$ are monotone nondecreasing. Plainly, $f_1(a) \leq f(a)$, $g_1(a) \leq g(a)$ for all a. We shall prove inductively that $f_n(a) \leq f(a)$ and $g_n(a) \leq g(a)$ for all n. Suppose by inductive hypothesis that this holds for a given value of n. Then, if $a \cdot> b$ while $f_n(a) \leq g_n(b)$, then

$g_n(b) \leq g(b)$, while $f(a) > g(b)$, so that $f(a) \geq g_n(b)+1 \geq f_{n+1}(a)$. If $a \lessdot b$ while $f_n(a) \geq g_n(b)$, it follows similarly that $f(a) \geq f_{n+1}(a)$, $g(a) \geq g_{n+1}(a)$. A similar observation is valid in case $f_n(a) \neq g_n(b)$ and $a \doteq b$. It therefore follows in all cases that $f(a) \geq f_{n+1}(a)$, $g(b) \geq g_{n+1}(b)$, completing our inductive argument. Thus, if any pair of functions f,g with the precedence property exists, the monotone nondecreasing sequences $\{f_n\}$, $\{g_n\}$ will be bounded above, and hence for sufficiently large n we will have $f_{n+1} \equiv f_n$, $g_{n+1} \equiv f_n$. Consequently the construction described in the preceding paragraph always leads to a pair f,g of functions with the precedence property, if any such pair exists.

Note finally that, if f,g is any pair of functions with the precedence property, and if m is any strictly monotone function defined on the union of the ranges of f and g, then the transformed pair m(f(a)), m(g(b)) of functions also has the precedence property. If N is the total number of terminal and intermediate symbols of our grammar, then the range of each of the functions f and g contains at most N integers. Therefore, if there exists any pair of functions with the precedence property, there exists a pair whose ranges are both contained in the interval $[1,2N]$. It follows, if a pair f,g of functions with the precedence property exists, that the sequences $\{f_n\}$ and $\{g_n\}$ of the preceding paragraph are bounded above by 2N. Since these sequences are monotone nondecreasing, the contrary hypothesis that $f_{n+1} \equiv f_n$ and $g_{n+1} \equiv g_n$ never hold together implies that the pair of sequences $\{f_n\}$, $\{g_n\}$ must be unbounded above. Thus, construction of the two sequences $\{f_n\}$, $\{g_n\}$ will reveal directly and algorithmically whether a pair of functions f,g with the precedence property exists, and gives the functions f,g when they do exist.

The above discussion terminates our analysis of parsing methods for precedence grammars. In the following section, we shall discuss initialization algorithms for precedence grammars, that is, we shall give an account of the "read-in"

algorithm which can be used to examine an arbitrary context-free grammar, decide whether or not this grammar is a generalized k-precedence grammar for given k, and, in case the grammar is found to be a precedence grammar, set up the two precedence tables needed by the parsing algorithm developed in the present section.

4. Parse initialization for generalized precedence grammars.

As we shall show explicitly in the present section, the condition that a context free grammar be a generalized k-precedence grammar for a given k is algorithmically verifiable. It is therefore appropriate, in applying the parsing algorithm developed in the preceding section, to supplement this algorithm by a "read-in" or parse-initialization algorithm. Such an initialisation procedure will examine any given context free grammar, verify that the grammar is a generalized k-precedence grammar, and set up the precedence tables required by the subroutine "compare" used by the parsing algorithm of the preceding section. (c.f. Table II).

Tables III A,B,C below give an account of such a syntax initialization algorithm. The algorithm, which is somewhat lengthy, is divided in the various Tables III into a main routine, described by Table IIIA, which calls various principal subroutines. Algorithms for these subroutines are given in Tables IIIB,C, etc. The over-all structure of these algorithms is as follows. For each pair a,b of terminal or non-terminal symbols of the context free grammar for which tables are to be set up, and for a variety of strings forming potential left-hand contexts of a, the algorithm aims to discover whether or not one of the three relations $a =_{\sigma} b$, $a \gtrdot_{\sigma} b$ and $a \lessdot_{\sigma} b$ holds. If at least one of these relations holds, then the algorithm must discover whether more than one of these relations holds.

Corrected May. 31, 1982

Table IIIA. Main routine for verifying that a grammar has the generalized k-precedence property.

```
          program precgram(k)
          for all character pairs a,b of grammar, do all instructions
               till end:
          put σ = null; overflow = false
          m = resolve (σ,a,b)
          table1(a,b) = m
          if(m.ne.4) go to end
          empty pushdown stack
          let σ = first character of symbol set of grammar
try:      m = resolve(σ,a,b)
          if (m.ne.4) go to enter
          put σ on stack
          let c be first character of symbol set of grammar
append:   prefix character c to σ to get new value of σ {σ = c cat σ}
          if(length(σ).gt.k) overflow = true and go to diagnose
          else go to try
enter:    if(m.eq.5) go to testdone
          enter value m against σ in hash table
Adv: c = head(σ); σ = TAIL(σ); IF (c .NE. Lastchar) {c = newtchar; goto append;}
testdone: if stack is empty go to end
          else {let σ = string at top of stack; goto Adv;}
          let c = first character of σ; let σ = portion of σ following c
          if c is last character of symbol set of grammar, go to testdone
          else advance c to next character of symbol set of grammar,
               and go to append;
end:      continue
finish:   if(overflow) emit diagnostic "grammar violates specified
               precedence depth limit" and call exit
          else reduce newly constructed tables to standard form and
               copy them onto the appropriate output medium, then call exit.
diagnose: delete initial character from σ
          print out diagnostic message  "σab constitutes unresolvable
               context of maximum length"
          go to testdone
```

Table IIIB. Precedence condition resolving subroutine.

```
            function  resolve (σ,a,b)
            m = 0
            if(equal(a,σ,b)) m = 1
            for all n = 1 till total number of productions, do all
                 instructions till loop1;
            put λ = intermediate symbol forming left hand side of n-th
                 production;
            if (not.begin(λ,b)) go to loop1
            if (not.equal(a,σ,λ)) go to loop1
            if(m.gt.0) m = 4 and go to back
            else m = 2 and go to testgreater
loop1:      continue
testgreater:for all n = 1 till total number of productions, do all
                 instructions till loop2
            put λ = intermediate symbol forming left-hand side of n-th
                 production
            put c = last character of right-hand side of n-th production;
            put ρ = right-hand side of n-th production, with last
                 character c deleted;
            if (c.notequal.a) go to loop2
            if σ is a terminal substring of ρ , put σ1 = null and go
                 to gramtest;
            if ρ is a terminal substring of σ, put σ1 = portion of σ
                 preceding ρ and go to gramtest;
            else go to loop2
gramtest:   if(not.gramsub(σ1λb)) go to loop2
            if(m.gt.0) m = 4 and go to back
            m = 3
loop2:      continue
            m = 5 if (m == 0) m = 5
back:       resolve = m; return
```

Table IIIC. Subroutine to determine the validity of a $=_{\sigma}$b.

```
          function equal(a,σ,b)
          do all instructions till continue1, for all n=1 till
               total number of productions:
          put λ = intermediate symbol forming left-hand side of
               n-th production
          if right-hand side ρ of n-th production contains no
               occurrence of ...ab..., go to continue1
          else for each occurrence of ab in ρ do all instructions
               till continue2:
          let σ2 be the substring of ρ preceding given occurrence
               of ab;
          if σ is a terminal substring of σ2, put σ1 = null and go
               to gramtest;
          else if σ2 is a terminal substring of σ, put σ1 = portion
               of σ preceding σ2 and go to gramtest;
          else go to continue2
gramtest: if(gramsub(σ1λ)) go to yes
continue2: continue
continue1: continue
          equal = false; return
yes:      equal = true; return.
```

to perform begin(a,b),
build connectivity matrix

Table IIID. Algorithm for the gramsub function.

function gramsub(σ)

if σ contains only a single character, put gramsub = from(λ_0,σ), where λ_0 is the root character of the grammar being processed and return.

otherwise let a be the first character of σ ; let b be the last character of σ, and let σ_0 be the part of σ that remains after the deletion of its first and last characters.

generate all strings τ which may be obtained by replacing:
a by any λ such that end (λ,a),
b by any μ such that begin (μ,b),
any character c of σ_0 by any ν such that all (ν,c).

for each of these strings τ, do all instructions to loopend:

write τ in all possible ways as $\tau = \tau_1\tau_2$, where τ_1 contains at least 2 characters, and where there exists a production $a \longrightarrow \alpha$ of the grammar being processed such that τ_1 is identical with a terminal string of α.
for each such production and each such decomposition of τ do the following instruction:

test1: if (gramsub($a\tau_2$)) go to isgram;

write τ in all possible ways as $\tau = \tau_1\tau_2$, where τ_2 contains at least 2 characters, and where there exists a production $a \longrightarrow \alpha$ of the grammar being processed such that τ_2 is identical with an initial string of α. for each such production and each such decomposition of τ_1 do the following instruction:

test2: if (gramsub($\tau_1 a$)) go to isgram;

write τ in all possible ways as $\tau = \tau_1\tau_2\tau_3$, where τ_2 contains at least 2 characters, and where there exists a production $a \longrightarrow \tau_2$ of the grammar being processed. For each such decomposition of τ, do the following instruction:

test3: if (gramsub($\tau_1 a \tau_3$)) go to isgram;

for each production $a \longrightarrow \alpha$ of the grammar being processed such that τ is a substring of α do the following instruction:

```
test4:   if (from( o,a)) go to isgram;
         gramsub = no; return.
isgram:  gramsub = yes; return
```

Any triple a,b, for which either none or at most, one of the three relations $a \cdot b$, $a \doteq b$, $a \cdot b$ holds, constitutes a determinate case, in the sense that no left-hand context of the pair a, b, beyond the context furnished by σ, is necessary for the parsing algorithm shown in Table II of the preceeding section to proceed. On the other hand, a triple a,b,σ for which more than one of the three above relations held, and for which no additional left-hand context as available, would lead to indeterminacy in the parsing algorithm of the preceding section, The initialization routine, on encountering such a triple, must therefore extend the left-hand context string σ by all possible characters of the grammar at hand, each such character to be appended to the left end of σ, attempting in this way to resolve the indeterminacy.

The over-all flow of the table initialization process is determined by the main routine shown in Table IIIA. This routine uses the principal subroutine 'resolve,' an algorithm for which is given in Table IIIB. The 'resolve' subroutine has three arguments σ,a,b, and returns a value m. The returned value m is equal either to 1,2,3,4 or 5. A returned value 1 indicates that the relation a = b and no other of the three relations $a >_\sigma b$, $a <_\sigma b$, $a =_\sigma b$ holds; the values m = 2 and m = 3 have a corresponding significance. The value m = 5 specifies that none of the three above relations hold. The value m = 4 specifies that more than one of the three possible precedence rel ations holds; in this case, additional left-hand context for the character pair a,b is necessary if indeterminacy is to be avoided. Note that the three arguments of the "resolve"

function are respectively a string and two characters; the string is of course to be regarded as a potential left-hand context for the character pair formed by the two final arguments.

The main routine shown in Table IIIA. initializes two tables, referred to in its algorithmic description as "table1" and "the hash table". Table1 is a two-dimensional table, and contains an entry for every pair a,b of characters. These entries have the values 1,2,3,4,5, which have the significance explained above. The hash table contains whatever supplementary information is needed to resolve precedence ambiguities between character pairs on the basis of left-hand context. Each entry in the hash table corresponds to a triple σ,a,b, and specifies one of the four values 1,2,3,4 for this triple. The "left hand context string" is extended by the algorithm of Table IIIA. to whatever length, up to a specified maximum limit, is necessary in order that at most one of the three relations $a = {}_{\sigma} b$, $a > {}_{\sigma} b$, $a < {}_{\sigma} b$ should hold. If there is any pair a,b for which more than one such relation holds for a given left hand context string σ of length k, then it follows that the grammar being treated by the algorithm shown in Table IIIA. is not a k-precedence grammar at all. In this case, the program "precgram' sets an overflow flag, emits a diagnostic specifying that the string σab at hand violates the specified precedence depth limit k, and processing continues with the next string σab which would normally be considered. No entry is made in the hash table for a triple σ,a,b for which a unique precedence relation holds with a shorter context string than σ, nor is any entry made for triples σ,a,b for which the subroutine "resolve", when called, will return the value 5 indicating that none of the three relations holds. This convention is reasonable, in that it packs a maximum amount of information into a hash table of given size.

The main routine of Table IIIA. begins by initializing the two-dimensional "Table1" using whatever value m is returned by the "resolve" function of the three arguments null, a, b. If a value of m which is different from 4 is returned, it indicates that at most one of the three relations a $\cdot\!>$ b, a $\doteq$ b, a $<\!\cdot$ b holds; in this case, no entry need be made in the hash table against the pair a,b, and the algorithm goes on directly to consider another pair of characters. On the other hand, if the "resolve" subroutine called with a null first argument returns the value 4, then additional context is necessary in order to establish definite precedence relations for the pair a,b and entries reflecting this fact must be made in the hash table. In this case, an auxiliary push-down stack is emptied, the string is initialized to the first character of the symbol set of the grammar, and the "resolve" function is again called. If a value equal to 1,2, or 3 is now returned, then an entry for the triple σ,a,b can be made in the hash table. After making the necessary entry, the main routine shown in Table IIIA. will then advance the first character of which the string σ consists to the next symbol of the grammar in sequential order, and loop back to call resolve again and to make a new entry. If, on the other hand, the subroutine "resolve," when called, returns the value 5 for a given triple σ,a,b, the sequence ... ab... cannot occur in a sentence, and no entry need be made in the hash table. Finally, if "resolve" returns the value 4 for the triple σ,a,b, then additional context beyond the context contained in σ is required in order to establish a definite precedence relation for the characters a,b. In this case, a somewhat different procedure is followed. This procedure is motivated as follows. On the one hand, the given left context string σ must be extended by additional characters for the reason we have mentioned. On the other hand, after this has been done, one will eventually want to return to the left context string σ, advance its first character from an existing value

to the next character in order, and loop again to make any necessary entry for the triple σ,a,b thus obtained. In order to keep track of the over-all process flow, one stacks the string σ at the top of an auxiliary push-down stack. After this is done, the first character of the grammar is prefixed to the string σ at its left-hand end, and the algorithm proceeds as above, using, however, the extended string σ. When all possible first characters for a string σ have been run through in this way, the algorithm examines the auxiliary push-down stack. If the stack is empty, then every necessary case has already been treated. If, however, a symbol is found on the auxiliary push-down stack, then this symbol is retrieved, its first character advanced to the next following character, and processing continues as before. Note that the same procedure could have been written without explicit use of a push-down stack in a language permitting recursion.

The algorithm for the principal subroutine 'resolve' is given in Table IIIB. This subroutine in turn uses three further principal subroutines, 'equal,' 'begin' and 'gramsub.' The function 'gramsub' has a single argument σ, and tests σ to see if there is any valid grammatical string containing σ as an embedded substring. The function 'equal' has three arguments, a, ,b, and returns the value 'true' or 'false,' depending on whether the precedence relation $a =_{\sigma} b$ holds or not. An algorithm for this function is shown in Table IIIC.

The 'begin' function has two arguments λ and b, both characters of the grammar, and tests to see whether λ can generate a string whose first character is b. This function need merely consult a bit-matrix of "begin bits", which has an entry for each λ and b giving the value of the begin function. This matrix can be calculated as follows: Start with a matrix $A(\lambda,b)$ equal to 1 if b is the first character of a string derived directly from λ; then repeat

the transformation $A(\lambda,b) \longrightarrow \min((\Sigma\ A(\lambda,\mu)A(\mu,b)), 1)$ a sufficient number n of times. Here, n can be any integer such that $2^n > r$, where r is the total number of symbols of the grammar being processed. The 'resolve' algorithm shown in Table IIIB. operates as follws. The value m to be returned by the function is initialized to zero. The subroutine 'equal' is then called; if it returns the value 'true', then the quantity m is set to 1. Next follows a loop to determine whether or not the relation $a <_\sigma b$ holds. This loop tests all the productions of the grammar to see if the grammar contains any character λ with the following properties: the leftmost character of the same string derived from λ, must be b, and the relationship $a =_\sigma \lambda$ must hold. Note that this corresponds exactly to the definition of the relationship $a <_\sigma b$ as given in the preceding section. If the quantity m has not been set to 1, and it is found that $a <_\sigma b$ holds, then we set m to 2; on the other hand, if the same relation is found to hold, and m has already been set to 1, then it follows that two conflicting precedence relations are valid. In this case, we set m = 4 and return immediately. After a first loop to test for the validity of the relation $a <_\sigma b$, there follows a second loop, of a rather similar structure, to test for the relationship $a >_\sigma b$. The second principal subroutine, 'gramsub', is called in this second loop. The reader is directed to Table IIIB. for a detailed account of this portion of the 'resolve' procedure.

A recursive algorithm for the subroutine 'gramsub' is shown in Table IIID. This routine recursively condenses the string which it is to test until either it has been condensed into the single root symbol of the grammar, or until no further condensation is possible. "gramsub" makes use of the 'begin' function which we have already described; also of a similar function 'end' which tests two characters λ and b

to see whether λ can generate a string whose last character is b; of a related function 'all', which tests two characters λ and b to see whether λ can generate a string consisting entirely of b; and of a function 'from' which tests λ and b to see if a string containing the character b can be derived from λ. We leave it to the reader to elaborate algorithms for the calculation of these functions.

The function "equal" is called by "resolve"; like "resolve" it makes use of the subroutine "gramsub". "Equal" checks the triple a,σ,b to determine the validity of the relationship $a =_\sigma b$. This is done by searching the total set of productions to find a production with the fillowing properties: The right hand side of the production must contain an occurrence of ...a b...; if σ_2 is the substring of the right hand side of this production preceding the occurrence of the symbol pair a b, then the string $\sigma_1 \lambda$ consisting of whatever part σ_1 of the left hand context string σ precedes σ_2, together with the intermediate symbol λ, must be grammatical. Additional details are shown in Table IIIC.

5. Restrictions, Extensions and Generalizations. Additional Comments.

In the preceding section we have called a context free grammar a generalized k-precedence grammar if the use of left-hand contextual strings σ of length at most k is sufficient to insure that at most one of the three relations $a =_\sigma b$, $a \gtrdot_\sigma b$, $a \lessdot_\sigma b$ can hold. We noted that every generalized k-precedence grammar is a (k,1)-bounded context grammar. It is worth observing that these three relations can easily be extended to relations involving not only a left hand context for a pair of symbols a,b but a right hand context as well.

The precedence relations which must be defined in order to do this are as follows:

i) we write $a =_{\sigma,\tau} b$ if there exists a grammatical string in which the sub-string ... σ a b τ... occurs and in

which a and b are both immediately derived from a common node of the parse tree for the string.

ii) We write a $\gtrdot_{\sigma,\tau}$ b if there exists a grammatical string in which a sub-string ... σ a b τ ... occurs and in which a is the final element of the sub-tree of the parse tree.

iii) We write a $\lessdot_{\sigma,\tau}$ b if there exists a grammatical string in which the substring ... σ a b τ ... occurs, and in which a is immediately derived from some node μ of the parse tree, while b is indirectly derived from the node μ.

If in setting the condition that at most one of the three above relations hold between characters of a grammar, we allow left hand contextual strings of length at most k and right hand contextual strings of length at most ℓ, we speak of a generalized (k,ℓ)-precedence grammar.

An easy adaptation of the arguments of the preceding section will show that every generalized (k,ℓ)-precedence grammar is in fact a $(k,\ell+1)$-bounded context grammar. Generalized (k,ℓ)-precedence grammars may be parsed by an algorithm adapted from the algorithm shown in Table II, Section 2, using tables which may be initialized by adaptations of the algorithms outlined in the various tables III of the preceding section. We leave all these adaptations to the interested reader.

Since a generalized precedence grammar is defined as one for which the three precedence relations alluded to above are mutually exclusive, and since the addition of new productions to a grammar increases the number of such relations which are valid, it follows that, if a given grammar is a precedence grammar, every one of its sub-grammars must also be a precedence grammar. Even if a grammar is not a generalized precedence grammar, it may be that an appropriately chosen sub-grammar is a precedence grammar. Such sub-grammars may be selected as follows. Starting with some intermediate symbol λ of a grammar Γ other than its root symbol, we make

up a sub-set Γ_0 of the total set of productions defining Γ, as follows. First enter into Γ_0 all productions of which λ is the left hand side. Next, form the set of all intermediate symbols appearing on the right hand side of the productions of Γ_0, and enter, as additional members of Γ_0, all productions of which one of these characters is the left hand side. This, in turn, brings about an extension in the set of all intermediate symbols occurring on the right hand side of productions in Γ_0. Forming this set of symbols we add to Γ_0 all productions in which one of these characters appears as the left hand side. Proceeding inductively in this way, we eventually arrive at a collection Γ_0 of productions having the property that if an intermediate symbol c appears on the right hand side of one of the productions in Γ_0, every production of Γ of which c is the left hand side is also contained in Γ_0. The collection Γ_0, which we shall also designate as $\Gamma(\lambda)$ when we wish to emphasize its construction, is then a sub-grammar of Γ. Even if Γ is not itself a generalized precedence grammar, the sub-grammar $\Gamma(\lambda)$ may very well be one. It may also be that various other sub-grammars, Γ_1, Γ_2, etc., of Γ, constructed in the same way, are also precedence grammars.

If this is the situation, an efficient parsing algorithm for the language at hand may be constructed as a combination of the top-down and the precedence parsing algorithms described in Chapter II and in the earlier sections of the present chapter, respectively. The procedure to be employed would have the following structure. The "main" routine of the parser would simply be a recursive top-down algorithm of the kind described in Chapter II. On the other hand, whenever this routine reached, as a subgoal, a point at which a substring constituting an intermediate symbol x whose associated subgrammar $\Gamma(x)$, was a precedence grammar, then the top-down algorithm would switch into an alternate mode in which an efficient precedence parse along the lines of Table II would be used. If a complete precedence parse

is obtained, so that a substring grammatical according to $\Gamma(x)$ is found, the parsing program will return to its "top-down" mode and continue. On the other hand, if an error is discovered before return from one of the precedence sub-parses employed, it follows that the top-down goal which the precedence parse was employed to accomplish is in fact not attainable, given the characters which follow in the input string. In this case a negative return to the top-down mode from the precedence mode must be and the top-down portion of the parser must take whatever back-up or diagnostic action is necessary. In order to facilitate this, it may be convenient to have every precedence subparse generate all its code into an auxiliary array. Successful return from a precedence subparse can then cause the transfer of all this code into the main code string; unsuccessful return from a precedence sub-parse can merely erase all the code generated during the sub-parse.

The "lexical" analyses described in Chapter III can be considered as syntactic analyses corresponding to precedence grammars of a special kind. Call a string σ of characters a prefix if there exists a grammatical string γ of the form $\gamma = \sigma\tau$.

The prefix σ is irreducible in γ, or simply an irreducible prefix, if no sub-tree of the parse tree of γ has all its terminal nodes in σ. Note that in a precedence parse of the sort discussed in the preceding sections, which scans over the input string, collapsing the sub-string of characters generated by the left-most complete sub-tree of the parse tree of γ to a single character, the set of scanned but uncompressed characters always constitutes an irreducible prefix in the sense we have just defined. Conversely, a precedence parse will always have to scan over the whole of an irreducible prefix before it finds any sub-string of characters generated by a single node. The maximum number of scanned, uncompressed characters built up during a precedence parse (which is equal to the maximum number of characters built up in stackl of the algorithm of Table II,

Section 2) is thus equal to the maximum length of an irreducible prefix. If this maximum length is finite, the precedence grammar is said to be of finite depth.

A grammar which is not of finite depth is provided by the following rudimentary example.

(1) $\langle nest \rangle = ( ) \mid (\langle nest \rangle)$

This grammar permits the generation of arbitrary strings consisting of a number of left parentheses, followed by a precisely equal number of right parentheses. Any string of left parentheses constitutes an irreducible prefix; the grammar (1) will generate irreducible prefixes of arbitrary length.

If a grammar is of finite depth, it may be parsed lexically, that is, by a finite state machine. We define such a finite state machine as follows: Since the maximum depth which the pushdown stack 'stackl' of the algorithm of Table II, Section 2, may reach is bounded, and since each item on the stack may assume only one of finitely many values, we may enumerate all possible states of 'stackl' as a collection

(2) $\tau_1, \ldots \tau_\ell .$

This collection of states is in fact identical with the collection of all possible irreducible prefixes of our finite state grammar. Suppose that the pushdown stack is in one of its possible states τ_j. Receipt of an additional, grammatically legal, input character in this state of the stack will transform the stack state, either by stacking an additional item or by unstacking a number of items. In view of the finiteness of the collection (2) we may summarize these possibilities in a finite set of transition rules

(3) $\tau_i \xrightarrow{c} \tau_j .$

Those particular transformations which cause elements to

be unloaded from the push-down stack will in general also cause other generative actions; these generative actions may be indicated in any convenient notation. Receipt of a grammatically illegal character when the pushdown stack is in any given state will cause a diagnostic to be emitted and the stack to be cleared.

It is plain from the above that a finite state machine whose internal states are in formal correspondence with the states of the list (2), and whose transition rules are those given in (3), can simulate all the steps of a parse for a finite depth grammar without actually using pushdown stack. The simulation can be programmed in the style of Chapter III, Section 1, i.e., can use a table of indexed transfers. As noted in Chapter III, such coding makes good use of the hardware of the computer on which it runs and attains high efficiency.

Note also that transformation of a finite-depth grammar into a code written in terms of indexed transfers can be performed mechanically by a suitable metacompiler.

We may define the finite depth property of a grammar in an equivalent, but more convenient, way as follows. Consider the parse tree of the grammatical string γ, and the sequence of nodes

$$(4) \qquad \nu_0, \nu_1, \nu_2, \ldots, \nu_k$$

in this parse tree leading from the root node ν_0 of the tree to the particular "twig" node ν_k matching the last character of the irreducible prefix σ. Call the node ν_ℓ, $\ell \geq 1$, <u>primary</u> if ν_ℓ is reached from $\nu_{\ell-1}$ according to the left-most branch of the set of branches proceeding downward from the node $\nu_{\ell-1}$.
If the node ν_ℓ is not primary, we call it <u>secondary</u>.
Since σ is an irreducible prefix, no branch of the syntax tree proceeding from the node $\nu_{\ell-1}$ and preceding the branch which goes to the node ν_ℓ can be the top-most node of a

sub-tree of the full parse tree. Thus, every such node of the syntax tree of Γ must be a twig, and will correspond to precisely one character of the irreducible prefix σ. It follows that there exist irreducible prefixes of arbitrarily great length if and only if there exist sequences (4) containing indefinitely many non-primary nodes ν_ℓ. Next note that the sequence (4) of nodes completely determines a smallest syntax tree containing the sequence of nodes (4) and generating a sentence γ, to wit, that tree containing the chain (4) of nodes and in which every node not belonging to the chain (4) is a twig. Since a given grammar consists only of finitely many distinct productions, it follows that if a grammar is not of finite depth, there exists a sequence (4) containing two nodes ν_m, ν_n of the same type. Moreover, if the subsequence $\nu_m, \nu_{m+1}, \ldots, \nu_n$ of (4) contains only primary nodes, then the sequence

$$(5) \qquad \nu_0, \nu_1, \ldots, \nu_m, \nu_{n+1}, \nu_{n+2}, \ldots, \nu_k$$

of nodes determines a minimal tree of the sort described just above, generating a sentence containing the same irreducible prefix σ as the minimal tree corresponding to the sequence (4) of nodes. Thus, if our grammar is not of finite depth, it admits a sequence of nodes $\nu_0, \ldots, \nu_m, \ldots, \nu_n$ in which ν_m and ν_n are of the same type, but in which not all the nodes in the subsequence $\nu_m, \ldots, \nu_n, \ldots$ are primary. Conversely, if this condition is satisfied, then the chain

$$(6) \qquad \nu_0, \nu_1, \ldots, \nu_m, \nu_{m+1}, \ldots, \nu_{n-1}, \nu_m, \nu_{m+1}, \ldots, \nu_{n-1}, \nu_m, \ldots$$

determines, in the above sense, a minimal tree with an indefinitely long irreducible prefix.

We may therefore make the following statement: a context free, bounded-context grammar fails to be of finite depth if and only if there exists a chain of productions

$$(7) \quad a_1 \rightarrow \ldots a_2 \ldots ; a_2 \rightarrow \ldots a_3 \ldots ; a_3 \rightarrow \ldots ; \ldots ; a_n \rightarrow \ldots a_1 \ldots .$$

in which not every a_j is the first character of the production (7) on whose right hand side it appears.

Note, for example, that using this condition it is easy to verify that the following grammar, which describes a language to which the lexical parsing methods of Section 1 of Chapter III can be applied, is of finite depth.

(8)
```
<integer> = <digit> | <integer><digit>
<real>    = <integer>. | <integer>. <integer>
<name>    = <letter> | <name> <nonspecial>
<nonspecial> = <letter> | <digit
<letter>  = A| B| C| D| E| F| G| H| I| J| K| L| M| N| O| P| Q| R| S
            T| U| V| W| X| Y| Z
<digit>   = 0| 1| 2| 3| 4| 5| 6| 7| 8| 9
<operator> = .<name>.
<hollerith prefix> = <integer> H
<octal>   = <integer> B
```

6. Bottom-up parsing by the method of nodal spans.

The quantitative problems which any parsing method capable of handling an arbitrary context-free grammar must face are dramatically illuminated by consideration of highly ambiguous grammars and, in particular, of the simple grammar

(1) $\langle\omega\rangle = a \mid \langle\omega\rangle\ \langle\omega\rangle$.

Any string w=aa...a is grammatical according to this grammar, but each such string has a highly ambiguous parsing; the set of all parses is equivalent to the set of all nested parenthesizations of w. The number of distinct parses according to this grammar of a string of length n may easily be seen by induction to exceed 2^{n-2}. Any general parsing method which in any exhaustive sense generates all parses of an arbitrary sentence must flounder in such cases, both in regard to the computation time required to generate all these parses, and in regard to the amount of space required for the storage of all the results which emerge. The exponential growth of computation time and storage requirements with string length noted above is closely bound up with the occurrence in highly ambiguous parses of imbedded ambiguities. That is, if a subsection of a string to be parsed has several grammatically ambiguous parses, all originating with a single node of a given kind ν; and if, in the full parse tree for the string, the string subsection in question can be condensed into a node of type ν, the full parse tree necessarily inherits the ambiguity of the sub-tree depending from the node .

If ambiguous nodes of this kind occur frequently and in nested fashion, the accumulation of independent ambiguities will lead to a degree of ambiguity for the total string which grows exponentially with the length of the string. This difficulty may be overcome by passing to a representation for the total set of parses, the so-called nodal span representation, which does not involve individual enumeration

of imbedded ambiguities. Bound up with this representation there is also an interesting and quite general bottom-up parsing method which it is the aim of the present section to describe and analyze. Suppose that Γ is a context-free grammar according to which we wish to parse a succession of sentences. Let ν be an intermediate symbol of Γ, and $w = a_1 \ldots a_n$ a string of terminal symbols which is to be parsed. If p and q are integers, $1 \leq p \leq q < n$, we call the triple (p,ν,q), which we will generally write for simplicity as $p\nu q$ a span of the grammar Γ. We say that the span $p\nu q$ is present in w if there exists a parse tree, representing a parse valid according to Γ, of the subsection $a_p \ldots a_{q-1}$ of the string w, and such that the topmost node of this parse tree is of type ν. Note that w is grammatical according to Γ if and only if the span $1\nu(n+1)$ is present in w. We call the grammar Γ standardized if it has the two following properties:

(1) no production of Γ has a right-hand side which is null;

(2) every production of Γ is either of the form $\eta \rightarrow a$, where a is a terminal symbol, or $\eta \rightarrow \nu\mu$ where ν and μ are nonterminal.

We shall show below that any grammar for a given language may be replaced in a mechanical and straightforward fashion by a standardized grammar. For the present, we merely suppose, in order to simplify our exposition, that the grammar Γ has already been standardized.

A representation of the full set of parses of a sentence w may be given in terms of a collection of lists as follows. Start with the span $p\eta q = 1\nu(n+1)$ present in w; consider all the productions of Γ having the form $\eta \rightarrow \mu\nu$. Since (inductively) $p\eta q$ is present, some pair of spans having the form $p\mu r$ and $r\nu q$ must be present. If the production $\eta \rightarrow \mu\nu$ and the integer r have this property, we add a triple consisting of the pair of symbols $\mu\nu$ and the integer r to a list (which we may call

the division list) belonging to the span $p\eta q$. We shall write such a division list in the form

(2) $\quad p\eta q \longrightarrow \mu\nu r_1,\ \mu\nu r_2,\ \ldots,\ \mu\nu r_k,\ \mu'\nu' r_1',\ \ldots\ .$

Then, recursively (and assuming that they have not yet been processed) we build up the division lists for all the spans $p\mu r_j$ and $r_j\nu q$, ..., etc. Continue processing in this way until no unprocessed spans remain.

We regard the full set of span lists developed by the procedure outlined above as a description of the set of all possible parses of the original sentence. To obtain a particular parse, we have only to proceed as follows. Start with the topmost span $p\nu q = 1\nu_0(n+1)$. Choose any element $\mu\nu r$ from the division list of $p\nu q$; then, recursively, choose an element from the division list of $p\mu r$ and $r\nu q$. This process will terminate when spans of the form $m a (m+1)$, which represent uniquely defined terminal symbols of the sentence w being parsed, are encountered. Note that the sentence w has an ambiguous construction according to the grammar Γ if and only if there exists a division list in the full collection of lists developed by the above procedure which contains more than a single element.

Assuming that a list of all the spans present in w is available, we may bound the amount of time and space needed to produce all the division lists as follows: the number of distinct spans is asymptotic to $n^2 g$, where g is the number of intermediate symbols of Γ. To produce the division list for the span $p\eta q$ we require a separate step for all r in the range $p < r < q$, and this for all productions of the form $\eta \longrightarrow \mu\nu$. Thus the total number of distinct steps required is bounded by a quantity asymptotic to $n^3 P$, where P is the total number of distinct productions in Γ. The space required for the storage of all the span lists is bounded by the product of P and the number of triples p,q,r with $p < r < q$, and hence also by the quantity asymptotic to $n^3 P$.

In the case of the elementary grammar (1) each r with

$p < r < q$ will actually appear in the division list belonging to the span $p\langle w\rangle q$; the full collection of span lists thus includes a number of elements bounded below by a quantity asymptotic to n^3P. By this consideration we see that the asymptotic bound n^3P given above is precise both in regard to time and in regard to space.

It appears clearly from what we have said above that, once a list of all the spans present in w is available, the set of all possible parses of w may be elaborated in an entirely elementary way. Our next task is to develop an algorithm for finding the set of all spans present in w. We may proceed toward this goal in the following way. The span $p\eta q$ will be present in w if and only if there exists an r with $p < r < q$ and the production $\eta \rightarrow \mu\nu$ of Γ such that $p\mu r$ and $r\nu q$ are present in w. We may therefore proceed in inverse order to build up all spans $p\eta q$ of greater and greater length $q-p$ present in w as follows. Start with all spans $p\alpha(p+1)$ of length 1, which necessarily represent single terminal symbols in Γ. Add new spans to a list of spans present in w by the following rule:

> if $p\mu r$ and $r\nu q$ are present, and Γ contains the production, η goes to $\mu\nu$, add $p\eta q$ to the spans present.

This algorithm, iterated, will clearly produce the set of all spans present in w.

An efficient arrangement of the flow of work in the algorithm which we have just described informally is as follows. For each k, $1 < k \leq n$, we generate the set of all spans present in the substring $w_k = a_1 \ldots a_k$ of $w = a_1 \ldots a_n$. We keep this information in the form of a set of <u>initials lists</u>, each list consisting, for each node type η and integer q, of the integers $p < q$ for which the span $p\eta q$ is present in w_{k-1}. To proceed inductively from k-1 to k, we begin by adding the span $k\alpha(k+1)$ to the collection of spans known to be present. Then, if a span $r\nu(k+1)$ has been added to the collection of spans known

to be present, and has not yet been processed, we process it by

(1) Finding all the productions $\eta \rightarrow \mu\nu$ of Γ, the terminal symbol of whose right-hand side is ν;

(2) For each such production and for each p on the initials list of μr, adding all the spans $p\eta(k+1)$ to the collection of spans known to be present; that is, adding all the elements of the initials list of μr to the initials list of $\eta(k+1)$. Each newly added and still unprocessed span $p\eta(k+1)$ must in turn be processed in this way; when no unprocessed spans remain, the set of all spans of the form $p\eta(k+1)$ which are present will have been generated. At this point, we increase k and repeat the above procedure, terminating the entire process when k has reached n+1.

The amount of time and space required for the above algorithm may be bounded as follows. As we have remarked, the total number of distinct spans is bounded above by a quantity asymptotic to n^2g, g being the total number of intermediate symbols in our grammar. The same formula gives, a fortiori, a bound for the total number of spans processed. In processing a given span $r\nu(k+1)$, we

a) check to see whether this span has already been marked as processed, in which case, since an earlier copy of the same span must have been found on the same initials list, we merely delete the span under examination from its initials list;

b) in the contrary case, process the span $r\nu(k+1)$ by enumerating all productions in Γ of the form $\eta \rightarrow \mu\nu$ and, for each such production appending the initials list associated with μr to the initials list associated with $\eta(k+1)$.

Note now that no more than c copies of an element $p\eta(k+1)$ will ever appear on an initials list, where c is the product of k+1-p and of the total number P of productions in Γ of the form $\eta \rightarrow \mu\nu$; thus the total number of elements $p\eta(k+1)$ which will ever appear on any initials list during the progress of

our algorithm from k-1 to k has the asymptotic bound k^2P. Since, as is clear from a) and b) above, each element appearing on such a list for a second time is processed in a fixed number of steps, while each element $r\nu(k+1)$ appearing on such a list for the first time is processed in a number of steps proportional to the number of productions in Γ of the form $\eta \rightarrow \mu\nu$, the total number of steps required by our whole parsing process has an asymptotic bound of the form n^3P.

Concerning the space required for the set of initials lists, note that each of at most ng initials lists can contain at most n elements; thus the amount of storage required has an upper bound asymptotic to n^2g.

If the grammar Γ is <u>unambiguous</u>, that is, if no grammatical string of terminal symbols admits more than one parse, the above time estimate may be significantly improved. In this case, the set I_1 of integers added to the initials list corresponding to $\eta(k+1)$ during the processing of a span $r\nu(k+1)$ must be disjoint from the set I_2 of integers already present in this list. If such were not the case, and assuming the integer p to lie in the intersection of I_1 and I_2, there would necessarily exist some span $r'\nu'(k+1)$ present in w, and some corresponding production $\eta \rightarrow \mu'\nu'$, such that $p\mu'r'$ is present in w, while $r'\nu'(k+1) \neq r\nu(k+1)$. But then the substring $a_r \ldots a_k$ of w would have two structurally distinct parses according to the grammar Γ, contradicting the assumption that Γ is unambiguous. This shows that for an unambiguous grammar, the algorithm described above will place at most a <u>single</u> copy of each element $p\eta(k+1)$ on an initials list. Thus, for Γ unambiguous, the total number of elements $p\eta(k+1)$ to be processed during the progress of our algorithm from k-1 to k has the asymptotic bound kg, g denoting the total number of intermediate symbols in Γ. It follows at once that the time required for the complete operation of our algorithm in the case of an unambiguous grammar and for the parsing of a string of length n has an asymptotic bound of the form n^2P.

It is interesting to consider, from the point of view of the present algorithm, the simple unambiguous grammar

(3) $$\langle\omega\rangle = \langle\omega\rangle\, a \mid a \ ,$$

which we know from the earlier sections of the present chapter to be prototypical of a class of grammars which permit bottom-up parsing of an arbitrary sentence of length n in a number of steps proportional to n. Taking $w = a^n$ as a string to be parsed, we find the spans present in ω to be all spans of the form $p\omega q$, $p < q$; a collection of $1/2\ n^2$ spans in all. This shows the upper bounds derived above to be precise, and makes it plain that our algorithm can only be expected to parse sentences in the time linearly proportional to their length if we sophisticate its action so as to prevent the formation of large numbers of spans (in the above example, all those spans $p\omega q$ with $p > 1$) not playing any role in the final parse of the sentence to be analyzed.

To improve the algorithm in this regard, we must sophisticate its action. The problem to be overcome is as follows. Suppose that, given a sentence w and a span $p\nu q$ present in w, we say that $p\nu q$ <u>belongs to a parse tree of</u> w if there exists some parse tree for w, containing a node of type ν from which there depends a subtree whose twigs are exactly the terminal symbols $a_p \ldots a_q$ of w. As the example considered just above indicates, many spans not belonging to any parse tree of w may be present in w. In many cases, the number of spans present in w will be proportional to the square of the length of w, while the number of spans belonging to a parse tree of w will be proportional only to the first power of w. We must therefore modify our parsing algorithm in such a way as to cause the modified algorithm to generate fewer spans not belonging to a parse tree of the sentence w being parsed. A technique accomplishing this may be described as follows. If μ and ν are intermediate symbols of Γ, write $\mu F \nu$ if there exists some span, valid according

to Γ depending from a node of type ν, whose first symbol is μ. In parsing a string $a_1 \ldots a_k a_{k+1} \ldots$, maintain a set of candidate lists L_k, one for each integer k; these will be lists of intermediate symbols defined inductively by the following rule:

Rule a: Let L_1 consist of all intermediate symbols μ such that $\mu F \omega$, the top node of Γ. For each span $p\mu(k+1)$ in $a_1 \ldots a_k$ and each production $\eta \rightarrow \mu\nu$ of Γ such that $\eta \in L_p$, put ν on L_{k+1}; if $\alpha F \nu$, put α on L_{k+1} also.

We now modify the algorithm described above by agreeing that an integer p is to be added to the initials list of a pair $\mu(k+1)$ if and only if

Rule b: our preceding algorithm would have added p to this same initials list, and;

Rule c: μ belongs to the candidate list L_p.

Our modified parsing algorithm has the interesting property stated in the following lemma.

Lemma 1: If the modified parsing algorithm just described places an integer p on the initials list corresponding to a pair $\mu(k+1)$, then there exists some sentence of the form

(4) $a_1 \ldots a_k b_1 \ldots b_\ell$,

valid according to Γ, to whose parse tree the span $p\mu(k+1)$ belongs.

Proof: We proceed by induction on the integer p. If p=1, the span $1\mu(k+1)$ is present in $a_1 \ldots a_k$ and our assertion is evident. If $p > 1$, $p\mu(k+1)$ is present in $a_1 \ldots a_k$ and $\mu \in L_p$, so that, by rule a) there exists a span $q\nu p$ and an intermediate symbol η of Γ such that either

i $\eta \rightarrow \nu\mu$ is a production of Γ, and $\eta \in Lq$, or

ii There exists an intermediate symbol δ of Γ such that $\mu F \delta$, while $\eta \rightarrow \nu\delta$ is a production of Γ, and $\eta \in Lq$.

In case i, q belongs to the initials list corresponding to the pair $\eta(k+1)$; thus, inductively, there exists a sentence of the form (4) in whose parse tree $q\eta(k+1)$ and hence a fortiori $p\mu(k+1)$ is present. In case ii, if we let $b_1 \ldots b_m$ be any grammatical string of terminal characters such that $\delta \rightarrow b_1 \ldots b_m$, then in the parse of $a_1 \ldots a_{k+1} b_1 \ldots b_m$ according to our algorithm we would plainly find that q belonged to the initials list corresponding to the pair $\eta(m+1)$; thus, inductively, there exists a sentence of the form (4) whose parse tree contains $q\eta(m+1)$ and hence a fortiori contains $p\mu(k+1)$. Q.E.D.

Following Knuth, we may readily define a class of grammars for which an upper bound on the length of an initials list may be deduced from the property stated in lemma 1.

Definition: A grammar Γ is said to have the LR(t) - property if every span of the form $p\nu(k+1)$ belonging to a parse tree of any sentence of the form $a_1 \ldots a_{k+t} b_1 \ldots b_n$ with given initial substring $a_1 \ldots a_{k+t}$ belongs to every parse tree of every such sentence.

Note that it follows trivially from the above definition that an LR(k) grammar is unambiguous; indeed, any span belonging to any parse tree of such a sentence belongs to every parse tree for such a sentence, so that the set of spans of a parse tree, and hence the parse tree itself, are uniquely defined by the sentence parsed.

The grammar of a given set of terminal strings may always be written either in a "left-recursive" or a "right-recursive" style. Thus, for example, the language whose sentences are all the strings a^n may be written either as

(5a) $\langle \omega \rangle = \langle \omega \rangle \; a \mid a$

or as

(5b) $\langle \omega \rangle = a \; \langle \omega \rangle \mid a.$

In general, the style (5b), which requires that terminal symbols be found before new "goals" are established, is preferable for top-down parses, while the form (5a), which leads to parse trees with handles tending to occur early rather than late, is preferable for bottom-up parses. Note that a grammar of the type 5a leads to a parse tree having the general appearance shown in Figure 1 below, while a grammar of the type 5b leads to a parse tree having the general appearance shown in Figure 2 below.

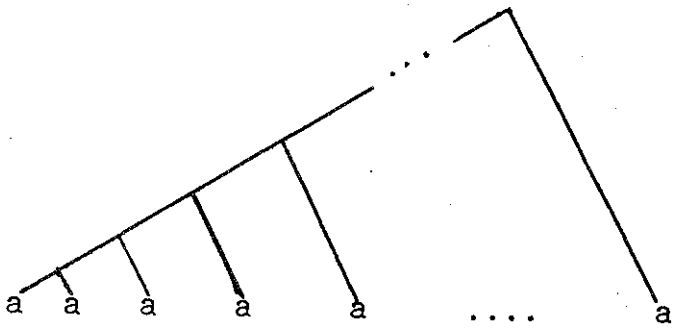


Figure 1. "Left-weighted" parse tree

For this reason, we call a grammar of the type 5a a left-weighted grammar, and call a grammar of the type 5b a right-weighted grammar.

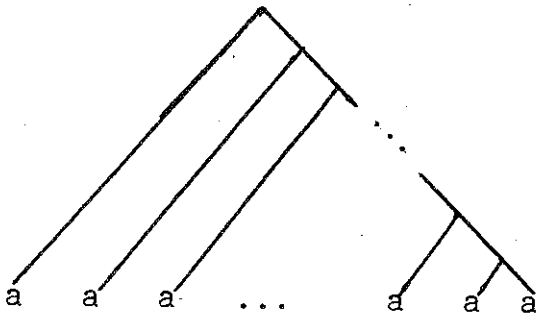


Figure 2. "Right-weighted" parse tree

More precisely, we make the following definition

Definition: A context-free grammar Γ is left-weighted if there exists no cycle of intermediate symbols

$\mu_1, \mu_2, \ldots, \mu_k$ and of productions such that

(6)
$$\mu_1 \rightarrow \nu_1 \ldots \nu_\ell \mu_2$$
$$\mu_2 \rightarrow \nu'_1 \ldots \nu'_\ell \mu_3$$
$$\mu_k \rightarrow \nu''_1 \ldots \nu''_\ell \mu_1 .$$

If Γ is a left-weighted grammar, it follows that there exists no infinitely long sequence of intermediate symbols $\mu_1, \mu_2, \ldots$ such that μ_{j+1} is the last symbol of a production whose left-hand symbol is μ_j. We may therefore define the length R of the longest such chain of intermediate symbols as the right-chain bound for the left weighted grammar Γ.

The following lemma shows that, for the class of left-weighted LR(t) grammars, the parsing algorithm defined by Rules a,b, and c will place at most nK integers on initials lists, K being a constant depending only on t and on the grammar Γ. This makes it plausible that, for such grammars, the operation of the parsing algorithm will require a number of steps proportional only to the first power of n, a fact that we will in fact establish later in the present section.

Lemma 2: If Γ is a left-weighted LR(t)-grammar, then the parsing algorithm described by the rules a), b), and c) above will, in the course of parsing a sentence $a_1 \ldots a_n$, place at most $RM^t n$ integers on initials lists. Here, M is the number of terminal symbols of Γ, and R is the right-chain bound for Γ.

Proof: It follows by Lemma 1 that, for each integer p placed by our parsing algorithm on an initials list, there exists a span pv(k+1) with $k \leq n$ belong to the parse tree of some sentence of the form $w = a_1 \ldots a_k b_1 \ldots b_t c_1 \ldots c_s$, and hence in the parse tree of every such w with given initial symbol sequence $w_o = a_1 \ldots a_k b_1 \ldots b_t$. If for each such w_o and for given $k \leq n$ we choose one sentence w, we obtain a collection W_k of sentences,

containing at most M^t sentences, such that the node $p\nu(k+1)$ is present in the parse-tree of one of these sentences. But if $p_1\nu_1(k+1)$, $p_2\nu_2(k+1)$, ..., $p_m\nu_m(k+1)$ all belong to the parse tree of a single sentence, while $p_1 \leq p_2 \leq \ldots \leq p_m < (k+1)$, it is easily seen that there exists a sequence of productions of Γ such that ν_{j+1} is the last symbol of the j-th production, and such that the left-hand symbol is of the j-th production is ν_j. It follows that $m \leq R$. Thus at most $R \cdot M^t$ spans $p\nu(k+1)$ belong to the parse tree of any of the sentences of our collection W_k. Letting k range from 1 to n, Lemma 2 follows. Q.E.D.

Corollary: If Γ is a left-weighted LR(t) grammar, then the parsing algorithm described by the rules a), b), and c) will never put more than M^t integers on any single initials list.

Proof: If we assumed the contrary, it would follow just as above that there existed a sentence w and two nodes of the form $p\nu(k+1)$ and $q\nu(k+1)$, both belonging to the parse tree of this sentence, and with $p < q$. Then, arguing just as above, we find that there exists a sequence $\nu = \nu_1, \nu_2, \ldots, \nu_k = \nu$ of productions of Γ such that ν_{j+1} is the last symbol of the j-th production, and such that the left-hand symbol of the j-th production is ν_j. The cyclic character of the sequence $\nu_1, \ldots, \nu_k$ means that these productions can be repeated any number of times; hence the existence of such a sequence contradicts the assumption that Γ is left-weighted. Q.E.D.

It is now easy to show that from the assumption that Γ is LR(t) and left-weighted, it follows that the action of the parsing algorithm described by rules a), b), and c) requires a number of steps proportional only to the first power of the number n of characters in the input string. Indeed, since we process only those spans $p\nu q$ for which p

is on the initials list belonging to the pair νq, it follows from Lemma 2 that at most $RM^t n$ spans are to be processed. The processing of a single span $p\nu q$ involves the following steps:

i. Find all productions $\eta \to \mu\nu$. At most P such productions exist, where P is the total number of productions in Γ.

ii. For each such production, examine all the integers r on the initials list belonging to μp. If η belongs to L_r, add r to the initials list belonging to ηq.

Assuming that the elements $\eta \in L_r$ can be found by direct addressing, process ii) above will, by the corollary to Lemma 2, require at most M^t steps. Thus, that part of the parsing algorithm concerned with the calculation of initials lists will require at most $RPM^{2t}n$ steps. The calculation of candidate lists according to Rule a) requires the following steps.

iii. For each p on the initials list belonging to μq, find all productions $\eta \to \mu\nu$ of Γ. This requires at most P steps for each $p\mu q$.

iv. For each such production, check whether $\eta \in Lp$. If this is the case, add ν to Lq. This requires at most P additional steps.

v. For each ν, maintain a list consisting of all those σ such that $\sigma F \nu$. On adding ν to Lq, add each element of this list to Lq also. This requires at most g steps for each ν, whose g is the total number of intermediate symbols present in Γ.

The processes iii), iv), and v) require in total not more than $P(g+2)$ steps for each $p\mu q$. Thus the operation of the entire parsing algorithm requires at most $RPM^t(M^t+g+2)n$ steps.

Note also that all the intermediate data lists required grow proportionally to n for LR(t) left-weighted grammars Γ.

The efficiency of the parsing algorithm described above will be enhanced if we note in regard to steps iv) and v) above that no symbol ν (resp. δ) need be added to the candidate list L_q unless $a_qF\nu$ (resp. $a_qF\delta$). Taking advantage of this fact we avoid the appearance on the lists L_q of unnecessary candidates, thereby securing a faster parse.

It is illuminating in connection with the above analysis to consider the simple left-weighted grammar

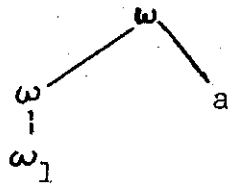
(7) $$\langle\omega\rangle = a \mid \langle\omega\rangle a \mid a \langle\omega\rangle b.$$

(Note that, if formal agreement with the terms of our preceeding discussion is required this grammar may be rewritten in our standard form as

(8) $$\langle\omega\rangle = a \mid \langle\omega\rangle a \mid a\langle\omega_1\rangle$$
$$\langle\omega_1\rangle = \langle\omega\rangle b.$$

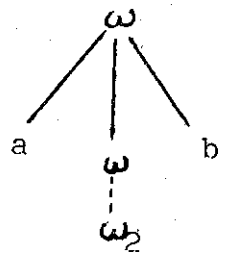
We will discuss the general question of grammatical standardization in somewhat greater detail below.) Note first of all that the grammar (7) is unambiguous. This may readily be proved by induction on the length of a sentence ω. If $\omega = \omega_1 q$ ends in the character a, its only possible parse is clearly

(9)



where on the left-hand side of (9) we have indicated that ω_1 is to be parsed. If $\omega = \omega_1 b$ ends in b, then its only possible parse is

(10)



where we have written $\omega = a\omega_2 b$, and where, in the center of (9), we have indicated that ω_2 is to be parsed. If we consider a sentence $\omega = a^n b^m$ and its parse according to the grammar (7), we find that the following spans belong to the parse tree of ω:

$$(11) \qquad p\omega(n+m+2-p) \text{ for } p = 1,2,\ldots,m$$
$$(m+1)\omega q \qquad \text{for } q = m+2,\ldots,n+1$$

Note in particular that the presence or absence of a span $p\omega q$ with $p<q \leq n+1$ in the parse tree of ω depends on the size of m, no matter how long the string a^n of a's prefixed to ω is. This shows that the grammar (7) does not have the bounded context property. It is also clear from what has been said that no left-to-right parsing method of the general sort described in the present section can avoid generating all the spans $p\omega q$ with $p<q \leq n+1$ in parsing $a^n b^m$ in accordance with the grammar (7); thus the LR(t) condition on a grammar Γ is necessary if sentences of length n are to be parsed in a number of steps proportional to n, by a left-to-right algorithm like those we have considered.

The algorithms discussed above may readily be organized by the use of a combination of list and bit techniques.

The lists and bit arrays required for the detailed algorithm are

a) For each μ and q, and initials list $I_{\mu q}$;
b) For each intermediate symbol μ of the grammar in question, a candidate vector of bits containing a 1-bit in the j-th position if and only if μ belongs to the candidate list Lj;
c) For each intermediate symbol μ of the grammar in question, a processed vector D_μ containing a 1-bit in the j-th position if and only if, at a given substage of our algorithm, the span $j\mu(k+1)$ has already been processed.

In addition to these lists we require a number of auxiliary tables. Some of these are best kept as lists; others may either be maintained as arrays, or since they will normally be sparsely populated, may more appropriately

be kept as hash tables. The required additional data structures are:

d) An array U with one entry for each intermediate symbol μ of Γ; this entry will be non-zero of, at a given stage of processing, $I_{\mu(k+1)}$ may contain elements representing still unprocessed spans, in which case it will reference the first possibly unprocessed element on $I_{\mu(k+1)}$.

e) An array V with one entry for each intermediate symbol μ of Γ; this entry will be non-zero if, at a given stage of processing, some span $p_{\mu(k+1)}$ has been generated. (Note: it is appropriate, both in the case of the array U and of the array V, to chain the non-zero elements together. If we do so we can test for the existence of a non-zero element of U or V very readily, and find such an element very readily if any exists).

f) A set of lists G_ν showing, for each intermediate symbol ν, all those η and μ for which Γ contains a production $\eta \rightarrow \mu\nu$;

g) A set of lists $H_{\mu,a}$ showing, for each intermediate symbol μ and each terminal symbol a, all those η and ν for which Γ contains a production $\eta \rightarrow \mu\nu$ for which $aF\nu$.

h) A set of lists $J_{\nu,a}$ showing, for each intermediate symbol ν and each terminal symbol a, all those intermediate symbols α for which $\alpha F\nu$ and $aF\alpha$.

We may express our parsing algorithm in terms of these data structures in the manner shown in Table IV below.

Table IV. Algorithm for parse by improved method of nodal spans

Initially: Lists $I_{\nu j}$ all empty;
C_ν has 1-bit on, all other bits off;
k=0; b=1'st input symbol;

Advance: b=k+1; a=b; if a is end-of-sentence return;

Readynext: D_ν=0 for all ν;
U=0; V=0;
b=k+1-st input symbol;
B=bit vector with only k+1-st bit on;
add k to $I_{\mu(k+1)}$ for each symbol μ such that Γ contains a production $\mu \rightarrow$ a;
for each such μ, place a reference to the newly added element k of $I_{\mu(k+1)}$ in the appropriate entry of U;

Process: If U-array has only zero entries, go to candidates; else
else let j,ν be that element of a list $I_{\nu(k+1)}$ referenced by the first non-zero entry of the U-array;

Do: If the j-bit of D_ν is 0, go to undone;

Drop: Else remove j from the list $I_{\nu(k+1)}$;
If $I_{\nu(k+1)}$ contains no subsequent element j' set the ν-entry of the U-array equal to zero, add ν to the ν-list, and go to process;
Otherwise, if $I_{\nu(k+1)}$ contains a subsequent element j', set $j = j'$;
Go to do;

Undone: Set the j-bit of D_ν;
If the j-bit of C_ν is off go to drop;
For each η and μ on the list G_ν, do all instructions until placed;
Append the list $I_{\mu j}$ to the end of $I_{\eta(k+1)}$;
If the η-entry of the U-array is zero, cause it to reference the first element in the list $I_{\eta(k+1)}$;

Placed: Continue;
Go to do;

)didates: If V-array has only zero entries, go to advance;
else let ν be some symbol represented in V-array;
remove ν from V-array;
for all γ and μ on the list $H_{\nu,b}$ do all instructions until added
for all j on the list $I_{\nu,k+1}$ do all instructions until added;
if the j-bit of C_γ is on, set $C_\mu = C_\mu$.or.B;
for all α on the list $J_{\mu,b}$ set $C_\alpha = C_\alpha$.or.B_j;
Added: Continue;
go to candidates;

The algorithm shown in Table IV parses a sentence but does not produce the division lists described earlier in the present chapter. It may, however, easily be modified so as to do so. To this end, we allow each entry p on a list $I_{\nu k}$ to include a field referencing the first item on an associated division list; this second list is a list of item pairs, each item of a pair referencing a particular item q on a particular initials list $I_{\alpha m}$. When (cf. the section "undone" in Table IV) the processing of a node $j\nu(k+1)$ causes $I_{\mu j}$ to be appended to $I_{\eta(k+1)}$, we attach, to the head of the appended list section, some indication that the associated "attachment parameters" are j, ν, and μ. When, subsequently, the elements $p\eta(k+1)$ of the appended list section are processed, we attach the pair determined by these parameters to the division list associated with the entry p of the list $I_{\eta(k+1)}$. (In order to be able to reference these lists quickly, reference to the last item on the division list corresponding to each span $p\eta(k+1)$ can be kept in a hash array $A_{p\eta}$ used during the k-th stage of the algorithm and re-initialized when k is incremented).

Note that the processing needed to build up the division lists adds only a finite number of steps to the processing of each span as shown in Table IV. Thus, all the computation-length estimates made above remain valid even if division lists are required. It is easily seen by inspection of the procedures suggested above that the following amounts of space are required for storage of the division lists in the three principal cases considered above (asymptotic estimates):

general ambiguous grammars: Pn^3
general unambiguous grammars: Pn^2
LR(t) left-tilted grammars: $P M^{2t} n$.

Throughout the present section, we have assumed the grammar Γ according to which input sentences are to be parsed to have the convenient special form described in (1) and (2) of the second paragraph of the present section. If

we begin with an arbitrary grammar, we can put it into this special form by the following sequence of steps.

A. Eliminate null productions. Determine all intermediate symbols of the grammar which can generate a purely null string as follows: mark every μ admitting a production $\mu \longrightarrow$ null string, and then, inductively, every μ admitting a production $\mu \longrightarrow \nu_1 \ldots \nu_k$ where all $\nu_1 \ldots \nu_k$ have already been marked, etc. Then given any production $\mu \longrightarrow \nu_1 \ldots \nu_k$ of Γ, add to Γ any production which can be obtained by omitting any subset of symbols, all marked, from the right-hand side of this production. Finally, drop all productions of the form $\mu \longrightarrow$ null string from Γ.

B. Eliminate productions of single intermediate characters.

If Γ contains any cycle of cycle of single-character productions $\mu_1 \longrightarrow \mu_2; \mu_2 \longrightarrow \mu_3; \ldots; \mu_k \longrightarrow \mu_1$ the characters $\mu_1, \ldots, \mu_k$ may be regarded, in an evident sense, as synonyms for each other. In this case, we may replace all occurrences in any production of a character μ_j, $j > 1$, by an occurrence of μ_1. Repeating this step until it can no longer be applied, we obtain a modified grammar for the same language as Γ which does not omit any such cycle of single-character productions. Suppose that this has already been done, so that Γ admits no such cycles of single character productions. If Γ contains any single-character production $\mu \longrightarrow \nu$, both μ and ν being intermediate symbols, choose such a production with the property that ν does not admit any single-character production $\nu \longrightarrow \eta$. Next, given any production $\eta \longrightarrow \alpha_1 \ldots \alpha_k$ of Γ in which $k > 1$, add to Γ any production which can be obtained by replacing by ν's any set of μ's occuring on the right-hand-side of this production. Given any production $\nu \longrightarrow \beta_1 \ldots \beta_\ell$ of Γ, add the production $\mu \longrightarrow \beta_1 \ldots \beta_\ell$ to Γ. Finally, drop the production $\mu \longrightarrow \nu$ from Γ.

The process we have just described eliminates one single-character production from Γ; repeating it as often

as necessary, we eliminate every single-character production from Γ.

C. Standardise the production of terminal characters.

For each terminal character a_i of the grammar, introduce a nonterminal symbol α_i and a production $\alpha_i \rightarrow a_i$. In each production $\eta \rightarrow \mu_1 \cdots \mu_k$ of Γ, replace all terminal characters a_i by the corresponding characters α_i.

D. Final steps. The grammar now includes productions of two types; productions $\mu \rightarrow a$, and productions $\nu \rightarrow \mu_1 \cdots \mu_k$ where $k \geq 2$ and all $\mu_1, \ldots, \mu_k$ are nonterminal. For every production of the second kind, introduce a sequence $\nu', \nu'', \ldots$ of additional intermediate symbols and productions

$$(12) \qquad \nu \rightarrow \mu_1 \nu', \quad \nu' \rightarrow \mu_2 \nu'' \ldots, \quad \nu^{(k-1)} \rightarrow \mu_{k-1} \mu_k .$$

With this last step we have brought our grammar into the required standardised form.

The parsing method described above admits of a generalisation useful in the efficient treatment of certain classes of large grammars. To see how such grammars may arise, consider a situation in which we are given two grammars Γ_1 and Γ_2, and a rule associating with each production of Γ_1 having the form

$$\nu \rightarrow \mu_1 \cdots \mu_k ,$$

one or several productions of Γ_2 having the form

$$\alpha \rightarrow \beta_1 \cdots \beta_k .$$

In this situation, we may form a kind of product grammar Γ as follows. For ν of Γ_1 and α of Γ_2 introduce a new intermediate symbol, namely the pair (ν,α); for every two associated productions as above introduce the production

$$(13) \qquad (\nu,\alpha) \rightarrow (\mu_1,\beta_1)\ (\mu_2,\beta_2) \cdots (\mu_k,\beta_k) .$$

Γ is the grammar consisting of all the productions (13).

Note that, if Γ_1 admits g_1 nonterminal symbols and Γ_2 admits g_2 nonterminal symbols, then Γ may admit up to $g_1 g_2$ nonterminal symbols, and a number of productions equally as large. Thus Γ may be a very large grammar, and its representation as a product grammar may be highly advantageous.

Even if a large grammar is not precisely a product grammar, but is only approximated by one, its most economical description may have the following form: specify a product grammar Γ' approximating Γ, and then list explicitly those productions of Γ differing from productions of Γ'. A reasonably general and flexible alternative scheme having essentially this flavor may be specified as follows. We make use of a context-free grammar Γ, but associate with each terminal symbol and intermediate symbol ν of Γ a set of <u>attributes</u>, whose values may either be single bits or bit-strings treated as boolean quantities or as integers. These same attributes attach to each span of the form $p\nu q$; for a particular span, the attributes assume particular values. With each production $\nu \rightarrow \mu_1 \cdots \mu_k$ of the grammar we associate a set (possibly null) of <u>tests</u>, which are arbitrary relations expressed in terms of the attributes of the set of spans $p_1\mu_1p_2$, $p_2\mu_2p_3$, $\cdots$ $p_k\mu_kp_{k+1}$ which are, in accordance with the inverse of the production $\nu \rightarrow \mu_1 \cdots \mu_k$, to be combined to produce a single span $p_1\nu p_{k+1}$. All of these tests must be satisfied for the use of the indicated production to be legitimate. With the same production we also associate a set of <u>rules</u> which specify the manner in which the attributes of $p_1\nu p_{k+1}$ are to be calculated from the attributes of the subspans producing this span. An easy adaptation, of the algorithm shown in Table IV, which we leave it to the reader to elaborate upon, will implement bottom-up parsing according to context-free grammars supplied with node attributes, tests, and rules in the manner indicated above. To show how much a grammar may be condensed by the use of such a scheme, we write out a grammar for the normal arithmetic expression as follows.

(14) <expr> = [1] <expr> <op> <expr> | [2] (<expr>) | [3] <name>

attributes: <expr> - level (4)
attributes: <op> - level (4)
1 : test - level(part1) ≤ level(part2) > level(part3)
rule - level = level(part2)
2 : rule - level = 0
3 : rule - level = 0

The conventions which have been used in (14) are as follows. Line 1 of (14) defines a simple context-free grammar; the integers in square brackets merely enumerate the productions of this grammar. The next two lines are merely declarations, and specify that each of the three symbols <expr> and <op> of the grammar have a 4-bit attribute field, called 'level' in each case. 'Level,' of course, refers to an order of operator precedence. The next two lines associate a test and a rule with the production [1]; the test merely specifies that, in each composite span constructed, the normal rules of operator precedence must be satisfied, while the rule specifies that the level of the composite node is to be calculated as the level of its operator part. Note that in every test and every rule associated with a production, we refer to the various subordinate spans to be combined into a single span by reverse application of the production as 'part1', 'part2,' etc.

The two final lines merely specify that both a parenthesised expression and an atomic name have dominant precedence in the parse of an expression. The reader should test his understanding of the formulae (14) by assigning levels to the five usual arithmetic operators +, -, *, /, ↑ in such a way as to cause (14) to coincide in its final effect with the normal arithmetic parse.

Near the end of the present work we will meet a still more powerful and flexible version of the above mechanism for the condensed expression of large grammars in an account of techniques of natural language parsing.

Section 4.7 - Notes and Comments

As computers came into wide use, efforts were initiated to use computers themselves to ease some of the burden of programming. This technique was first known as automatic programming. The main problems which programmers faced around 1951-52 were the small size of the high-speed internal memories, and the lack of any internal floating point operations in the machine hardware. The lack of floating point hardware forced early efforts in automatic programming to be directed to the development of interpretive floating point systems; the Comprehensive System (CS) for Whirlwind at MIT (cf. Adams and Laning [1]), the Speedcoding system for the IBM 701 (cf. Backus and Herrick [1]), and SHORT CODE for UNIVAC I all provided programmed interpretive floating point arithmetic.

In 1952-53, J.H. Laning and N. Zierler (cf. Laning and Zierler [1]) constructed an algebraic coding system at MIT for use at the Instrumentation Lab. This system allowed mathematical expressions and control statements of an elegance and simplicity which has hardly been equaled since. This system compiled a set of closed subroutines and an interpretive code for calling these subroutines. Because of the small memory size of the Whirlwind, each equation (statement) was read from a drum whenever it was to be executed. The language also included subscripted variables, computed switches, loops with an index variable running over a list or varied by an increment, and a natural way of expressing numerical integration of differential equations. The system used the CS floating point routines.

In 1952, H. Rutishauser at the Swiss Federal Institute of Technology (cf. Rutishauser [1, 2, 3]) had described methods

of compiling algebraic equations for the ZUSE computer, and later for the ERMETH computer (Rutishauser also described optimization by loop unwinding).

These early systems we e shortly to give great impetus to the subsequent development of algebraic compilers. For a time, work continued on interpretive systems which provided machine-like pseudo-instructions for floating point and various other convenient operations, e.g. the PRINT system for the IBM 705 (cf. Bemer [1]) and the A-2 and A-3 systems for the UNIVAC I.

Following this initial period, work on algebraic translators began in several places. In 1954 at IBM a group under J. Backus began the development of FORTRAN I for the IBM 704, a machine which was to have floating point hardware; in 1955 at Remington Rand, a group under G. Hopper and C. Katz began to develop MATH-MATIC (AT-3) for the UNIVAC I, a machine without floating point hardware and with only 1000 words of memory (cf. Taylor [1]). At the Boeing Aircraft Company, in 1955, M. Grems and R. Porter [1] described an algebraic system for the IBM 701, BACAIC.

FORTRAN, which was completed early in 1957 (cf. Backus et al. [1], Sheridan [1]) was destined to become the most widely used algebraic coding language, in part because of the large number of IBM 704 computers which were sold, in part because of the excellence of this pioneering compiler. The fact that the 704 had floating point meant that FORTRAN was a "pure" compiler, interpretive operations being performed only in the execution of the input/output and FORMAT statements. Since it was anticipated that programmers would object to the use of compiled code which was highly inefficient relative to hand tailored code, great pains were taken in the FORTRAN compiler to generate efficient object code. As a result of this the compiler was slower than it might have been; nevertheless, it set a standard of optimization for subsequent high-grade compilers. In 1958 the language was expanded to include independently compiled functions and subroutines.

This was a needed improvement both because of its naturalness and because it permitted larger programs to be written in parts. Eventually, some version of FORTRAN was produced for every IBM computer which was used for scientific computation and for computers of most other manufacturers as well.

MATH-MATIC, which was in several ways a more powerful although a less concise language than FORTRAN, might have been a substantial FORTRAN competitor. Its limited acceptance was due in part to the unsuitability of the UNIVAC I for scientific computation. It is noteworthy that MATH-MATIC provided automatic segmentation and attempted to keep loops entirely within one segment.

The IT (Internal Translator) developed for the Datatron and for the IBM 650 in 1957 (cf. Perlis, Smith and Zoeren [1]) demonstrated that substantial algebraic compilers could be written for a small machine,and a whole series of other compilers of this class then followed: RUNCIBLE, GATE, CORREGATE, GAT, FORTRANSIT.

Mehlan [1] and the papers which follow it in the journal in which it appears may be consulted for an account of an assembly language developed before the existence of true compilers.

The pioneering FORTRAN compiler of Backus et al. made use of advanced optimizations; the syntactic analysis methods which it incorporated, however, were ad hoc and relatively primitive. Some of the earliest systematic techniques were based on precedence schemes, particularly on the notion of operator precedence. Group [1] gives one of the earlier accounts of the use of stacks in the translation process. Arden and Graham [1] describe an early compiler based on a precedence parse and anticipating, in germ, some of the table-driven parsing methods developed later; cf. also Kanner [1] and Hamblin [1] for accounts of the precedence techniques as applied to algebraic expressions, and Knuth [1] for an account of an early compiler taking a simple algebraic language directly into machine language. Cf. also

Wegstrem [1] for an early, explicitly programmed formula translator. Samelson-Bauer [1] is an early paper introducing the bounded-context parsing method and stressing its applicability not only to algebraic formulae but also to the translation of other statements occurring in programming languages; this paper also discusses questions of target code style related to the compilation of multiply indexed expressions.

The systematic syntactic definition of ALGOL given in the two famous ALGOL reports (Naur [1,2] and Perlis-Samelson [1]), together with the work of Chomsky [1], aided greatly in popularizing the systematic view of syntax, in particular, the syntax of programming languages. The resulting parsing algorithms fall into two classes: the bottom-up schemes, including those utilizing precedence techniques: and the top-down schemes.

The literature on precedence and bounded context parses begins with the development of techniques for the parsing of arithmetic expressions as summarized, for example, in Randall and Russell [2]; cf. also Huskey and Wattenburg [1], which describes an elementary, explicitly programmed arithmetic expression scanner.

Systematic theoretical development of fast bounded context and precedence parse methods begins with Floyd [1,2]. The first of these papers gives a direct description of a class of bounded-context syntax analyzers which operate by condensing stacked symbols representing grammatical types in contexts known to be unambiguous; in the second paper, a general definition of precedence language is given and a high speed method for the parsing of such languages is described In Floyd [3], a more general class of bounded context languages is introduced, and the possibility of analyzing these languages rapidly by the method of "inverse grammers" is pointed out; cf. the comments at the beginning of Sec. 4.2. Graham [1] gives a general surveyoof bounded context translation with a discussion of problems of code production and

of a number of problems related to optimization, together with a bibliography of papers up to 1964. Evans [1] is a particularly readable account of bounded-context parsing, with a brief account of the relation of parsing to the remainder of the compilation task. See also Irons [4], Paul [1,2] and also Samelson-Bauer [1] which describes a 1-1 context parser; a similar 1-1 context parsing scheme is explained in Eichel, Paul, Bauer and Samuelson [1].

A history of the use of this compiling technique, with some comments on target code style, is given in Samelson-Bauer [2,1,3]; cf. the more general review in Samelson-Bauer [4].

The concept of generalized precedence grammar which is described in Sec. 4.2, together with the parsing method for these grammars described in that section, is due to Wirth and Weber [1]; cf. also McKeeman [1]. Ross [2,3] describes a precedence-like algorithm in which broader contexts can be considered using a back-up scheme. Leavenworth [2] outlines a precedence-parse scheme for translation of FORTRAN, with special attention to the parsing of Boolean statements. Cf. also Wegner [1], Keese and Huskey [1], Bandat and Wilkins [1].

The general parsing technique available for precedence grammars is very efficient; this method, supplemented as appropriate by special devices for the high speed determination of prefixed statement key words for languages in which such keywords exist, seems to be the best general method of syntactic analysis for application in situations where little language variation is expected and a high speed parse is desirable. The no-backup top-down parse of Chapter 2 has advantages if compiler flexibility is more important than parse speed.

The nodal span parse described in Sec. 4.6 is due in its full development to Earley [1,2]; this method goes back to earlier work of Cocke and Younger, cf. Younger [1,2], and

Hayes [ ] and is related to work of Knuth on LR(k) grammars, cf. Knuth [2].

Kanner [1] describes a non-backup top-down compiling scheme, and makes some comments on target code style, symbol table handling, and the general problem of name resolution. Lietzke [1] describes a top-down syntax checker for Algol. Kuno [1] describes a top-down parser, intended for application to natural language parsing, which works with grammars in which the right-hand side of each production begins with a terminal symbol (standard form grammars). Kuno observes that the speed of such a parser may be increased if a bit table of failed alternatives of the form [syntactic type, initial word position] is maintained, and if an alternative once failed is never retried. Cf. also Kuno and Oettinger [1].

Floyd [4] gives a brief survey of syntactic analysis methods as developed up to 1964, with an emphasis on top-down methods. This article includes an excellent bibliography of papers up to its date.

A continuing source of controversy has been the relative efficiency of the various parsing methods. Two different questions can be asked: how badly does the algorithm do in the worst case, and how badly does it do on "normal" statements. Although some data exists on the former question, the latter is much harder to answer. Griffiths and Petrick [1] discuss the number of steps required by various straightforward top-down and bottom-up parsing algorithms, as well as a simple nodal span parser, to parse languages defined by various very elementary grammars. They point out that the number of steps required by a top-down parser may grow exponentially with the length of the sentence to be parsed, and that the effectiveness of a parsing method may, in certain cases, depend sensitively on the particular grammar chosen for the language to be parsed. Modelling the various

parsing algorithms which they study in a standard way by Turing machines (not necessarily a very good way of measuring the efficiency of these parsers) they give a certain amount of empirical data on the lengths of simulated parses. Unfortunately, the algorithms considered by Griffiths and Petrick are sufficiently different from the optimal forms of the parsing methods that they study, and the empirical data presented sufficiently narrow, so that the only conclusion that may reliably be drawn from their work is that a naive top-down parser may be catastrophically inefficient. Cf., however, the objection of Brooker [1] to that conclusion. The Griffiths and Petrick paper also includes an extensive bibliography on parsing methods developed up to its date.

Kasami [2] exhibits a simple class of languages which cannot be recognized in less than $C(n.\log n)^2$ steps by a class of parsing methods modelled by Turing machines.

For a review of the application of these and other analysis techniques to English, cf. Bobrow [1].

Compilers organized around a systematic view of syntax analysis first emerge in the papers of Irons [1], Warehall [1], and Brooker and Morris [1,2,3,4]. Brooker and Morris describe an elementary top-down syntax-directed compiler scheme like those considered in the early sections of Chapter 2. This system includes a language, somewhat stiff in the view of the present author, for the description of generative actions. A summary of the Brooker-Morris work is contained in Rosen [2]. The compiler described by Irons is rather similar, and, like the Brooker-Morris system, includes a primitive language for the description of generator routines. Cf. also Cheatham and Sattley [1], McClure [1], Metcalf [1], and Reynolds [1]. Schoore [1] describes a similar top-down compiler-compiler incorporating a special, somewhat limited language for the output of macro-symbols during syntax analysis; Schneider and Johnson [1] describe a slightly extended version of this system able to accomplish certain local optimizations: Cf. also Pratt

and Lindsay [1] for an account of a similar system, somewhat specialized in the direction of interpretive execution rather than compilation. Brooker et al. [2] summarizes experience up to 1967 with the Brooker-Morris compiler-compiler; they indicate that compilers written using their system tend to be approximately twice as large and three times as slow as carefully designed hand-coded compilers. (Compilers employing lexical prescanners and faster syntactic analysis algorithms can do considerably better than this.)

Ledley and Wilson [1] descrive a bottom-up parsing method applicable to grammars containing no recursive definitions, together with a simple scheme for describing target language by expressions specifying local node transformations in a parse tree. Cheatham [3,4] and Dean [1] describe TGS-II, a compiler-writing system consisting of a conventional algebraic languages, supplemented with a powerful pattern matching facility. The memory of the machine has been reconstituted as a set of descriptors; these are the data of TGS-II. These descriptors impose a skeletal structure on the dictonary and other tables. The system of Domolki [1] is particularly interesting both because the user specifies the grammar in inverse form and because the associated parsing algorithm is elegant and efficient.

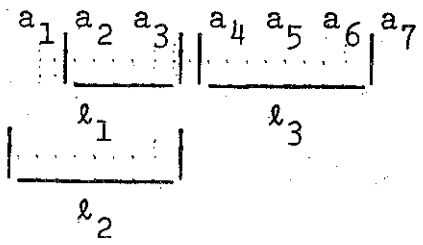
The input to the compiling system is an ordered set of inverse grammar rules. As usual, the right hand sides of these rules may have calls to compiling routines interspersed with the elements of the reduced string. Thus

$$\gamma_i: \quad \alpha_1\alpha_2 \cdots \alpha_k \rightarrow \beta_1\gamma_1\beta_2\gamma_2 \cdots \beta_\ell\gamma_\rho$$

is a typical rule, where the α_i, β_j represent elements of the vocabulary, terminal or nonterminal, and the γ_j represent calls to the compiling routines. The presence of such a rule implies that when $\alpha_1\alpha_2 \ldots\alpha_k$ is recognized

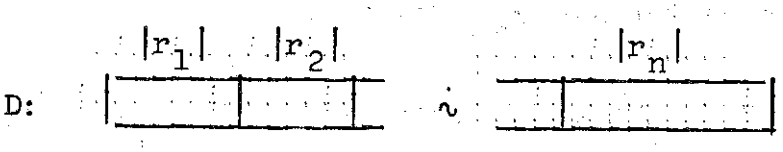
as a constituent of the input stream, it is to be replaced in the stream by $\beta_1\beta_2 \ldots \beta_\ell$ with $\gamma_1, \gamma_2, \ldots, \gamma_\ell$ executed at the indicated places.

Generally speaking, given a partially parsed string, the left hand sides of several rules may appear in it. For example, one might have



where 1, 2, 3 represent instances of left hand sides or rules of the grammar. Under these circumstances, the Dömölki scheme requires that the grammar give the "correct" parse if the rule of shortest length that terminates at the left most point is applied. In the above example, ℓ_1 would be applied. In case of a tie, the rule with smallest index is applied.

The elegance and efficiency of the Dömölki algorithm depend very heavily on the ability of computers to perform bit string operations in parallel. Let $r_1, r_2, \ldots, r_n$ be the rules of the grammar and let $|r_i|$ represent the length of the left hand side of rule r_i. The basic data structure D of the algorithm is a bit string of length $\sum_{i=1}^{n} |r_i|$. Such a string contains a bit position for every element of the left hand side of every rule:



For convenience, if b is a data structure of type D, we shall write

$$b(j,k) = 1$$

to indicate that the bit corresponding to the k^{th} word of the j^{th} rule is on.

Let $\{v_i\}_{i=1}^{m}$ be the vocabulary, terminal and nonterminal, used in grammar rules. To each v_i we correspond a data structure V_i of type D with the property that

$$V_i(j,k) = 1 \leftrightarrow r_i \text{ is the } k^{th} \text{ element of rule } j.$$

Thus the V_i structures are an encoding of the left hand sides of the rules.

In addition, we define two structures, F and L that respectively mark the beginning and end of rules:

$$F(j,1) = 1, \quad j = 1,2,\ldots,n;$$
$$L(j,|r_j|) = 1, \quad j = 1,2,\ldots,n.$$

Finally, the algorithm makes use of a stack Q of structures of type D. An element Q_t of the stack is associated with a pointer into the input stream. The significance of an element Q_t is defined by the following statement:

$$Q_t(j,k) = 1 \leftrightarrow \text{the last } k \text{ characters scanned are the first } k \text{ characters of rule } j.$$

The heart of the algorithm is the following. Suppose one has the structure Q_{t-1} and wants to calculate Q_t, the structure associated with the next input character. Let the next input character be v_p. Then

$$Q_t(j,k) = 1 \leftrightarrow Q_{t-1}(j,k-1) = 1 \text{ and } V_p(j,k) = 1.$$

More exactly, let R(Q) be the function that shifts the Q

structure right one bit. Then

$$(1) \quad Q_t = (R(Q_{t-1}) \vee F) \wedge V_p.$$

Observe that a left hand side has been found whenever

$$(2) \quad Q_t \wedge L \neq 0.$$

Using this notation, the algorithm can be described very easily. The algorithm used two stacks, one for the Q elements and one for the input characters.

A scan cursor p is started at the beginning of the input string, and Q_0 is set to 0. Then calculation (1) is made, and Q_t and the input character are placed on their respective stacks; the scan pointer is advanced. Then test (2) is made to see if any rule is complete; if not, the procedure is continued until a completed rule is found.

When a rule is found to have been completed, a check is made to find out which rule r_i is applicable. The vocabulary elements p_j of the right hand side of rule r_i are placed in front of the scan cursor p, which is then set to point to the first of these. Commands are executed as discovered. Then $|r_i|$ elements are popped from each of the stacks and the algorithm continues.

Conway [1], using a diagrammatic notation, describes a no-backup parser which again is much like those considered in Chapter II. Conway's parsing algorithm applies to a slightly restricted class of no-backup languages; in particular, the range of applicability of this compiler is restricted by the fact that no method for providing generator routine assistance to the syntactic analyzer is incorporated in his system. Conway's article also contains a discussion of some of the issues involved in a strictly

"one-pass" style of compiling. A discussion of some of the special issues involved in the treatment of COBOL qualified names is also included.

Continuing the line of development initiated by Brooker and Morris in the series of papers cited above, a number of other authors have described more or less complete compiler-compiler systems. The McKeeman system described by McKeeman, Horning, and Wortman and surveyed above is one such; Feldman [1] describes another such system incorporating a bounded context parse and a semantic metalanguage suitable for the description of the inverse grammars in terms of which such a parse is specified. Feldman's system includes a reasonabley general semantic language; generator routines written in this language may be called by the semantic analyser; the whole system is rather like the "generalized Backus" system described in Chapter II, except that it uses a bottom-up rather than a top-down parser. This system was developed at Carnegie-Mellon University for use there. Gilbert and McLellan [1] describe a compiler-generator system, related to the GENESIS system of the Computer Sciences Corporation, in which syntax is specified by direct listing of a family of significant contexts (inverse or "analytic grammar" technique). Recognition of any significant context will invoke corresponding generative actions; these generative actions are written using a macro language which refers to temporary registers, symbol string registers, to symbol attributes kept in a standardized hash table, etc. Generative output actions, specified in an output macro-language, and diagnostic output actions, specified in an appropriate form, may also be invoked. The system also incorporates a scheme for high-speed lexical analysis of an input string. Sibley [1] describes an experimental compiler-compiler system, with the acronym SLANG, developed at the IBM Corporation. This system, intended to emphasize machine independence, turns source code into relocatable intermediate language, which is then transformed into executable code

with the help of a collection of machine-dependent code emission routines written in a macro-language having a weak conditional macro-expansion facility useful in optimizing the local code-selection process. The system incorporates a "machine-features questionnaire" in which such basic machine features as word length, register number, etc., are specified to the code generator as parameters. However, as the optimization problem cannot be treated so simply, the SLANG system has not proved to be practical. Grau [1] describes a semi-symbolic language suitable for the description of target code in a compiler-compiler system. Warshall and Shapiro [1] describe a compiler-compiler system incorporating a macro-language for the description of code to be generated; this article emphasizes the problems of code generation and describes a number of simple code optimizations. Cf. also Bolduc et al. [1] and Lin Chang and Marks [1,2]. Barnett and Futrelle [1] describe a syntax analyzer and its input metalanguage. Burkhardt [1] presents a brief summary of some of the early compiler-compiler systems.

One of the most important tasks of a compiler is to provide error diagnostics to a programmer attempting to use a given source language. Unfortunately, the important problem of adapting the various syntax analysis methods available to us so that they also serve this end effectively is one that is hardly discussed in the published literature. We may, in attempting to elucidate this point, begin by observing that diagnostic messages may be separated into two approximate classes, according to the type of error which they record: syntactic diagnostics, which note the ill-formedness of program elements, the inconsistent use of declarations, etc., and which try to pin-point the particular statements or statement fragments responsible for the occurrence of an error, and semantic diagnostics, which note errors in program flow of a sort that can most easily be detected by the flow analysis section of a code optimizer (see below, chapter

on code optimization). In the present paragraphs, we shall only discuss error diagnostics of the first type.

In regard to diagnostics, a compiler ought to aim during its syntax analysis pass to produce as many valid diagnostics as possible; this will allow a programmer to proceed at maximum rate to complete the syntactic debugging of his program. This aim indicates the use of one or another scheme for recovering efficiently from the effects of a single error so as to allow compilation and the detection of additional errors to proceed (Note that the occurrence of a single error of a degree of severity sufficient to preclude execution can be used to set an internal switch in the compiler maintaining syntax analysis and declaration analysis but turning off code generation to gain speed.) The main feature of programming languages which facilitates partial error recovery is the almost universal definition of a <program> as a single list of <sentences> separated by easily distinguished end-of-sentence marks. This allows a compiler to continue past an error by issuing any diagnostic relevant to a single sentence and then by skipping to the next beginning-of-sentence point where the syntax analysis process is restarted. To indicate the probable approximate location of the error in any sentence in which an error is detected the compiler may underscore the erroneous sentence up to the last point to which the syntax analysis process is successful; depending on the parsing method used, this would be the first point at which an illegal precedence condition was detected in a generalized precedence parse, the first point at which an irreducible context pattern was detected in a bounded context parse, and the token marking ghe end of the longest input string which could form the initial portion of a sentence in a top-down parse. A non-backup top-down parser of the sort described in Chapter II will normally be able to associate a necessary syntactic goal and an unattainable syntactic subgoal with any error condition; printing out this infor-

mation as part of the associated diagnostic will generally aid the source language programmer.

Parsers with more sophisticated idagnostic aims may attempt to guess corrections for errors occurring within sentences to be parsed; correct or approximately correct guesses will allow other errors occurring in the same sentence to be diagnosed. The guessing algorithm can use some uniform heuristic or may be designed more irregularly to embody empirical experience of the likelihood of errors of various classes. It is to be noted, however, that over-insistent attempts to guess error corrections may result in the production of voluminous and confusing spurious diagnostics accountable rather to the correction guessing mechanism than to the programmer which it is attempting to aid.

Irons [3] describes an interesting heuristic for attempted diagnostic error-correction. Inaa bottom-up parse, one begins by condensing as many syntactically unambiguous substrings of the input as possible into single syntactic elements, thereby producing a maximally condensed input string α. If this normal parsing procedure fails to give a complete parse, i.e., fails to condense an entire input sentence into a single symbol, one examines α, searching for a point at which the interpolation of an appropriate syn= tactic element will allow the condensation into a single element of two or more symbols present in α. If none such exists, that element of α which represents the shortest substring of the original input string is deleted, and the process repeats, Note that since successive iterations of this process necessarily produce successively shortened intermediate strings α, Irons' algorithm is convergent.

Irons' idea may be adapted for use by a top-down parser as follows. To the longest initial substring σ of a sentence to be parsed we associate the most general goal Γ which the parser attempts to fill (unsuccessfully, of course)

after scanning the last character of σ. If the remaining portion of the sentence contains any token A which could form a valid first character of a syntactic element of type Γ, we drop out all elements of the sentence between the end of σ and the first such character A. In the contrary case, we interpolate some character valid as the first token of a syntactic element of type Γ. In either case, the operation of the parsing-error correction algorithm continues.

The very interesting paper of Freeman [1] describes a set of pragmatic error correction techniques which have been used successfully in a student-problem compiler at Cornell. The techniques used include the following: illegal punches are converted to a standard "error" character, serving as a flag to a subsequent "correct mispellings" subroutine. This subroutine incorporates a number of sophisticated error-correction guessing techniques, based on the comparison of a word in which a misspelling is suspected to a list of words likely to occur in its place; basic information obtained from such comparisons is supplemented by statistical information concerning common keypunch errors, and statistics on the usage of particular words within the program being compiled. Syntactical conflicts in repeated uses of a single word are treated as misspellings by adding a nominal "illegal punch" character to words used in conflicting fashion. An attempt is made to break up words misspelled by the omission of a blank separator into their component parts.

The subscript 1 or the subscript pair 1,1 is automatically attached to an array name erroneously used without subscripts; a variable not declared as an array but repeatedly used as an array is checked for near-agreement in spelling with a declared array, and, using this information, a decision is made to treat the variable as an array or not. If the variable is not classified as an array, expressions juxtaposed to it in parentheses are treated as implied multiplications. Missing operands in arithmetic expressions

are supplied as "1". Redundancy in the statement forms of the language is used when available, to supply missing parts of statements. Missing or misplaced labels are supplied by searching nearby statements and other relevant context for labels; misspelling correction for labels is guided by the occurrence of nearly matching words in syntactic settings implying those words to be labels. Similarly, misspelling correction for variable names may be guided by the occurrence of nearly matching pairs of words on the right and left sides of assignment statements.

The system described by Freeman also provides a number of execution-time debugging aids, including a cumulative statements-executed count, an individual labeled-statement-executed count, arithmetic trap messages, a check on illegal and excessive array indices, and a check on illegal repition and branch conditions. For other discussions of the correction of spelling errors, see Blair [4] and Damerau [1].

An amusing and informative discussion of various general issues which the diagnostic section of a compiler must face, together with a more general discussion of the pragmatic issues arising in compiler specification, is found in Hartman and Owens [1].

A successful counterpart to the Backus formalism for syntax description has yet to be found for the description of programming language semantics; consequently the work on code generation has for the most part been ad hoc and non-systematic.

A number of papers discuss various problems connected with target code style in FORTRAN, ALGOL, and other languages. Ingerman [3] makes some comments on target code style applicable to subroutine linkages in ALGOL. Irons-Feuerzeig [1] discuss the same question in slightly greater generality.

Warshall [1] describes, in broad outline, a code gen-

erator system which accepts as its input a standardized analysis tree produced by a syntax analyzer, and which yields assembly-language or machine-language code as its output. The system incorporates a language for describing tree-traversal and conditional code-generation actions, code generation being controlled by a set of tables describing registers in the machine to be compiled for, the sequences of operations available for affecting various basic operations, and the cost in time of executing these operation sequences. Various local optimizations (register selection, etc.) are incorporated in this system on a general basis; additional, essentially arbitrary, subroutines may be called from within the system.

Because of the effectiveness of the Backus formalism, there have been many attempts to find similarly effective mechanisms for the specification of programming languages. This effort, still relatively unsuccessful and not even fully focused, aims at two related goals.

1) To find a metalanguage in which elements of language which are syntax related but not entirely syntactic as, for example, conditions of consistency in the use of definitions, may be concisely expressed.

2) To find a language in which the meaning of programs, i.e., the rules defining the effect of executing a valid program, can be concisely expressed.

One may interpret each of these either in a strong or in a weak sense, i.e., either in the strong sense of procedural specification, or in a weaker non-procedural sense. Thus, for example, goal 2) can be attained in its strong sense only by providing (in some more "basic" language) a definitive program interpreter which is at once concise enough to be logically transparent and specific enough to be mechanically transformable into an acceptably efficient executable interpreter. On the other hand, goal 2) is attained in its weak sense if a program interpreter is pre-

cisely defined in a "basic" language of mathematically unambiguous meaning but with the use of dictions whose transformation into algorithms may be unspecified or whose algorithmic representation may be catastrophically inefficient. The value of such a specification lies in the fact that it can provide a concise and authoritative definition of the intended meaning of programs written in a language of interest; would-be authors of compilers for the language will then have an authoritative document against which the correctness of their detailed specifications can be checked, and one may even imagine that mechanical proof-algorithms could be used to verify that compiler produced code necessarily did have the effect required by the concise specification of program meaning for the language being compiled. Naturally enough, the farther we retreat from the strong goal of providing specifications which are both precise and algorithmic, the more conciseness and logical transparency become of prime significance. Among the precise formal languages contributed to us by mathematics, the language of sets and functions, in its various versions, is distinguished for generality and power, and it is no surprise to find aspects of this language incorporated in various schemes proposed for weak-sense semantic specification. Let us note that the set-theoretical language incorporates elements going in various degrees beyond those belonging to languages intended for actual compilation or interpretation: Among the facilities so provided we may note:

1) sets A to which elements may freely be added and whose unions and intersections may readily be formed;
2) functions f defined by sets of ordered pairs or n-tuples and the ability to form such sets as $\{x|f(x)\varepsilon A\}$, whose specific algorithmic construction may imply very extensive calculation;
3) predicates P of several variables in terms of which propositions such as $(\exists x_1)(\forall x_2)(\exists x_3)P(x_1,x_2,x_3)$, possibly implying very extensive calculations, may be formed;

4) "unspecified search" or "an elmment which" constructions such as (ix)P(x) which may imply extensive searches;
5) set-from-set constructions, like the power set or "set of all subsets" construction, which imply impractically enormous expansions of original data sets;
6) free use of recursion and other powerful auxiliary mathematical definition methods.

The value of using a powerful albeit non-implementable mathematical language for the definition of language semantics is noted by McCarthy [1] who discusses with relative clarity the issues involved in such an effort; cf. also van Wijngaarden [1], Steel [1], Garwick [1], Strachey [1], Landin [1] for related discussion. McCarthy proposes a scheme incorporating recursion and structured inter-node reference in tree-like data bodies as a mathematical basis for semantic definition, and emphasizes the fact that the semantic specification problem may be isolated from irrelevant syntactic detail by taking programs to be given in a syntactically preanalyzed tree form. An application of these ideas to a small subset of ALGOL is given in McCarthy [2]. McCarthy's proposal has subsequentlybbeen developed by workers at the IBM Vienna Research Laboratory and applied to define the semantics of the PL/1 programming language, cf. Lucas, Lauer, Stigleitner [1], and Bandat [1] for a brief summary of this work.

Many papers have been written on details of internal compiler organization. Several of these deal with various aspects of symbol table handling. Batson [1] describes a linear symbol table organization for ALGOL; Sadmine and Weinberg [1] and Kanner et al. [1] describe tree-structured dictionaries. Description of listing techniques are given in Buchholz [1] and Morris [1].

The handling of procedures in ALGOL is discussed in

Ingerman [4]. Irons and Feuerzeig [1] discuss the mechanisms required to hadle recursive procedures and block structures.

Storage mapping algorithms and the handling of user-defined data structures is another area with a large literature. Sattley [1] discusses the handing of dynamic arrays in ALGOL 60, and introduces the notion of dope vectors. Arden [1] and Galler and Fisher [1] give algorithms for resolving equivalence declarations.

The incorporation of macro-definition features into compiler languages to permit the extension of such languages, as well as the more general problem of allowing language extensibility through the definition of new data types, syntactic forms, etc. is an interesting problem which has been the subject of a number of papers.

Two principal issues which must be faced in designing a scheme for macro extension of source languages:

1) To what extent shall the macro processor interact with the source language compiler?
2) What range of application is the macro feature intended to cover?

In regard to 1), we may note that a spectrum of possibilities exists. Macro processing may use the tokens of the language to be compiled (i.e., use the lexical scan routine of the compiler with which it is associated) but otherwise be fully independent of the syntax analysis. In this simplest case, macro processing will be confined to a pass preceding the beginning of the translation process proper; the macro processor will transform an initial token string into an expanded string of essentially similar tokens. It is possible to use a macro processor of any one of several degrees of complexity for this purpose, beginning with a simple key word expander built directly into the lexical analyzer, ranging through a macro expander of essentially the same type but including a parameter-substitution fea-

ture, up to the use of a general recursive-macro transformation scheme like that described by Strachey [1]. For still greater generality in macro processing, a specialized string processing language with elaborate pattern-recognition features may be employed (SNOBOL, discussed later in the present work, is a good example of this sort of thing). Still greater generality in this direction would be attained by using a syntax pre-analyzer to transform the input string into a tree and by coupling the syntax analyzer to a recursive tree processor (like LISP) able to effect extremely general recursive transformations of trees recognized. The advantage of such schemes is that they are transparent and no more difficult to use than the string and list processors which they incorporate. Their disadvantage is that they cannot make use of any of the information gathered by the compiler with which they are associated unless the programmer using them re-specifies much of the syntax analysis which is to follow. For this reason, schemes allowing more interaction between a macro expander and an associated compiler have been proposed. In such a scheme, the course of macro expansion may in part be determined by information found by the compiler, as for example the syntactic role played in a given sentence by particular input tokens, the attributes assigned in the course of compilation to input tokens, etc. Such schemes must, however, face difficulties which arise from the fact that the particular set of grammatical definitions and the details of the particular method of syntactic analysis which a compiler employs may be relatively complex and unknown to the average language user. Unless a special effort is made to simplify and standardize both grammar and order of analysis in a compiler closely associated with a macro-processor, so large a mass of special information may be involved in attempting to use the macro-processor as to make it rather difficult to use. If a macro expander is to access attribute information gathered by a syntax analyzer with which it is associated, these

attributes must be maintained and referenced in a manner common to the syntax analysis program, the associated macro processor, and the user of the total system; this may in significant ways constrain the order in which the syntax analyzer establishes attributes, manipulates them, etc. Moreover, if the output of the syntax analyzer is semi-compiled code rather than strings of tokens in a form differing only slightly from the input form of these tokens, the would-be user of a macro expander may be forced into unpleasant interaction with the details of intermediate target-code style employed by the compiler with which the macro-expander is associated. In avoiding this while still combining macro-processing with syntax analysis one might be led to the use of a syntax analyzer which (at least on a sentence by sentence basis) keeps analyzed code in standard tree form convenient for transformation by a recursive macro-processor incorporating conventions for the description of local node transformations. Alternately, a syntax related macro processor may be designed to interact only with certain particularly simple subparts of a total syntax structure (as for example arithmetic expressions and assignment statements) and to be aimed specifically at some such special problem as the addition of new data types to an existing language.

In regard to question 2) above, we may note that, whereas macro-processors used in connection with assemblers are often used only to diminish the burden of repetition typically associated with assembly language coding, sophisticated macro packages can provide far-reaching language transformations which adapt an assembby language rather well to a particular field of application. In fact, Halpern [1, 2, 3] argues, although not entirely convincingly that a sufficiently sophisticated macro processor will make compilers unnecessary. For a more modest example of what the intelligent use of macro facilities permits, cf. Bennett and Neumann [1]. A

well-designed source language will itself be adapted to the application area for which it is intended, and in its typical use will therefore often be information-full and rather un-repetitive. This circumstance may tend to restrict the applicability of source-language associated macro-processors. On the other hand, a macro-processor well adapted to the source language to which it is associated, and reflecting likely directions of extension, may be very useful, and in the best case can allow the creation of what are almost new languages for special purposes. It may also be noted that special language features (e.g., the COMMON convention in FORTRAN) sometimes force the repetition of invocations, and that, in such situations, even a simple macro-processor may be quite useful. (Cf. Hopgood and Bell [1] for a straightforward practical application of such an idea.)

McIlroy [1] is an interesting early paper on macro-expansion of source languages. McIlroy describes a system intended for use in connection with source languages and resembling that customarily used within assemblers; this system is of the simplest type envisaged above, i.e., works with the tokens of a source language but is syntax-independent. It incorporates a set of conventions for the specification of compile-time calculations, the conditional expansion of macros, recursive macro-nesting, macro-definition nesting, and other powerful features allowing flexible definition of new language forms within the context of a fixed set of data types. Cf. also Cohen [1]. Leavenworth [1] describes a more restricted source-language macro-scheme, essentially providing only unconditional source language macros.

Cheatham [1] describes several plausible schemes for the extension of source languages by the inclusion of macro features. His proposals include not only a syntax independent macro expander, but also a macro-expander, called from

within the syntax analyzer, which uses the syntactic category to which a given token is assigned to decide whether any particular expansion mechanism is to be applied and to control the details of its application.

Leroy [1] describes a macro-expander for use in connection with programming source languages and allowing compile-time calculations on a particularly flexible basis. In this scheme, normal source-language programs are written; but variables occurring in these programs may be declared as symbolic, and the "execution" of a statement containing a symbolic variable causes a copy of this statement, with all non-symbolic variables replaced by their momentary values, to be placed in the macro-expanded output string. Calls on macros, in the ordinary sense, are replaced in this scheme by calls on subroutines containing symbolic variables.

Galler and Perlis [1] go beyond the schemes described by McIlroy and Cheatham, and describe a scheme for the introduction into ALGOL of new data types and of new binary and monadic operators. Their shceme thus belongs to the third general class of macro-extension methods described above. The inherent simplicity necessary to make such a scheme workable is provided by Galler and Perlis as follows:

1. The scheme is restricted to the definition of new declarations specifying additional data types within a basic ALGOL, to the introduction of new monadic and binary operations for the combination of these data types, and to the introduction of generalized assignment statemtnes by means of which the values of structured new data types may be established.

2. Arithmetic expressions are assumed to be analyzed by a precedence method; thus the syntactic problem involved in introducing a new operator reduces to that of defining the precedence of the new operator relative to previously defined operators. The syntax of assignment statements is, of course, evident.

3. New data types are defined by recursive application of the basic data-structure formation mechanism of ALGOL, i.e., as arrays of arrays of arrays of ... to any desired depth, the relevant dimensions being specified in straightforward fashion as macro parameters. Thus, one may write such definitions as complex a means array a [1:2]; matrix a(n) means array a(1:n, 1:n). These basic definitions would then allow the straightforward interpretation of complex matrix a(n) as array a(1:n, 1:n, 1:2).

4. The semantics of new newly introduced operations and of old operators in their application to newly introduced data types is defined by specifying for each operator and all its possible operand types an evaluation routine. This evaluation routine may depend both on the operation and on the data type of its operands; occurrence i- source text of an operator with its operands will lead to the interpolation as an open subroutine, of the corresponding evaluation procedure. If the substituted text still contains algebraic expressions requiring expansion, these expression will in turn be replaced by evaluation procedures, and so on until an open subprogram free of expressions requiring macro-expansion is obtained. The semantics of an assignment statement, in its application to nonstandard data types, is handled similarly; the meaning os such an assignment statement is defined by specifying a program text to be substituted for the assignment at each of its occurrences.

Galler and Perlis give a number of interesting examples of the use of this scheme, together with a fairly detailed account of the manner in which it can be implemented.

The central feature of the compiler-compiler languages described above is the presence of a method of specifying syntax patterns, together with a program for processing these patterns. The semantics specification portion of these systems is usually poorly developed; frequently all that is present is a mechanism for calling subroutines written in some unspecified language.

There is another group of languages which are also aimed, in whole or in part, at making compiler writing simpler.

In form, these languages resemble the general purpose algebraic languages such as FORTRAN and ALGOL. Instead of concentrating on the syntactic, pattern-matching, language facilities, they include features intended to make the construction and accessing of dictionaries, stacks, and other complex information structures easier; they frequently also include facilities for doing detailed storage management.

Two such systems are the AED-0 and AED-1 systems described by Ross and Rodriques [1]. Fundamental to these systems are the notion of the plex (cf. Ross [1,4] Tabory [1]), which permits the construction of very general list structures. A primitive macro facility is also included.

The CPL system described by Barrow [1] resembles ALGOL 60, but has more flexible block structuring and declarative facilities. Furthermore, lists, strings, and files are also included as data items. Richards [1] describes BCPL, distantly related to CPL, which is specifically designed for compiler writing. Its most interesting feature is that its "memory" consists of fixed length bit strings, stored contiguously. There are operators defined for treating these strings variously (and interchangeably) as strings, integers, and labels. The language also provides built-in recursion and a generalized conditional statement.

A language designed for systems programmnng that takes the opposite approach is described by Lang [1]. This language contains data of several types: character, bit, arithmatic, and logical. It also allows assembly code to be inserted at any point.

The POPS system, developed by the DIGITEK Corporation and since used by others, is a language, intended for interpretive execution, which embodies data structures and manipulation procedures useful in the programming of compilers. This system, which aims more at compact representation and machine independence than at high-speed compila-

tion, has been used to write a number of successful small-machine compilers; it attains a level of machine independence which greatly reduces the labor of carrying an existing compiler over to a new machine. The POPS language enables the expression in machine independent form of the first phases of a compiler, including parsing, diagnostic production, and the generation of standard form intermediate language code; subsequent passage from POPS standard intermediate code to actual machine operations must be expressed in some other language.

POPS provides two basic data forms, words (divided into a few standard subfields) and push-down stacks (consisting of sequential groups of words). A push-down stack may be further structured into segments, each segment being a delimited-sequential group of words in a stack; an entire segment may be moved by a single POPS operation from one stack to another. POPS provides various convenient stack manipulation operations as well as operations that move data between single words and stacks; both whole words and significant subfields can be moved. Instructions inserting constants at the top of stacks and into significant fields of given words are also provided, as are various other useful stack-transforming operations such as stack reversal. The system incorporates a one-bit condition flag, and contains compare operations applying to given locations and to the tops of stacks which allow the setting and unsetting of this flag; the conditions flag may then be used to control transfers. Various direct conditional transfers, as well as instructions capable of conditional execution, are also provided.

POPS includes several interesting stack search operations. These operations search all or a designated part of a stack for a field identical with a data pattern held in one or several data words. The search procedes serially through the stack by groups of words; the size of each group and

the location within it of the appropriate search key may either be specified by the search instruction itself or may be determined from attribute information attached to the stack being searched. Alternatively, a stack may be divided into word groups varying in size, the size of each group being in this case recorded with the group; such stacks may also be searched by the POPS search instructions. A search instruction sets the POPS condition flag, and, if successful, returns the location of the group of words which has been found. Note that the use of serial rather than high search techniques imposes a certain inefficiency on a compiler written using POPS.

Various logical entities naturally occurring in connection with compiler writing, such as an input character stream, an output intermediate code stream, an error message file, etc., are recognized as system entities within the POPS language and treated specially by certain instructions. Thus, for example, an instruction comparing a next incoming character with a specified single character is provided; another instruction comparing a group of input characters to a specified "key word" is also provided. It is also possible to associate attributes with every possible input character, and then to check any given incoming character for these attributes using a single instruction. Code-emission commands are included in POPS; these place intermediate form output at the top of a code stack, using operation-argument references in standard form, producing a standard form result reference, and automatically updating a standard intermediate-instruction "location counter." System provision is also made for treatment of error diagnostics, andinstruction being provided which transmits a current input character pointer and a diagnostic message number to an error stack. This same instruction also in-

crements a count of errors maintained by POPS, and brings a maximum error severity flag to its appropriate value. The system also provides a save-and-restore feature which allows backup and trial of alternate line of parse subsequent to the occurance of a parse error.

A POPS instruction may be applied either to a given stack, a given address, or a calculated stack or address; in particular, stack or address identifier may be specified indirectly. The POPS system provides basic algebraic, logical, and shift instructions applying to single words, as well as a method for linking POPS-written source text to arbitrary routines written in machine assembler language or some other source language. POPS written subroutines may be freely and recursively called; indeed, since stacks are used as a basic data type, the POPS system is systematically recursive.

As has been remarked above, POPS aims not so much at compilation efficiency as at small size. The POPS system provides about 100 basic macro instructions. The systemmatic use within the system of stacks significantly reduces the number of locations which must be addressed explicitly. Interpretable POPS code may thus be represented conveniently by a sequence of short words (16 bits is a quite reasonable length) divided into two halves, the first containing an order code, the second referencing either a stack or a one word variable. A similarly condensed representation of transfer operations is attained by dividing all unconditional and conditional POPS transfers into two classes, local and global. Local transfers go from one POPS instruction to a relatively nearby instruction and specify their target relative to their own location; global transfers may have any labeled POPS instruction as a target, but specify their targets indirectly via a sequential list of global labels. In either case, a short field is sufficient, conducing yet again to brevity. POPS allows the

construction of quite small compilers; figures concerning the size of typical POPS-workerscompilers will be found below as part of a more general description of various small compilers.

Summary descriptions of a number of interesting compilers have been given inthe literature. Backus et al. [1], which we have already cited, includes a rather brief description of the structure of the overall sequence of passes making up the pioneering IBM FORTRAN compiler; these begin with syntax analysis and proceed through optimization to the production of relocatable code; the internal structure of each pass is sketched in outline. Pyle [1] outlines a palnned FORTRAN compiler for the English ATLAS computer; this 3 or 4 pass compiler is intended to include common subexpression and reduction in strength optimization, together with some machine dependent local optimizations. An ALGOL compiler incorporating strength reduction, reordering of algebraic expressions for optimal register use, and subroutine linkage optimization by analysis of the pattern of subroutine calls occurring in a complete program is described in Huxtable [1], cf. also Huxtable and Hawkins [1]. Englund and Clark [1] give a short description of an algebraic language compiler incorporating various local optimizations (especially avoidance of unnecessary loads and stores) and an extension of these optimizations to regions containing forward branches only.

Gries, Paul and Wiehle [1,2] describe an ALGOL compiler for the 7090, built of four passes: lexical preprocessor, syntax analyzer, optimizing translator, and code selector. The main optimization performed is a special case of the "reduction in strength" optimization described in Chapter VIII. A special technique useful for handling the name scoping which occurs in ALGOL is described. Huskey [1] describes an early, complete compiler for a small algebraic language incorporating a subroutine call feature; this compiler uses a precedence parse for syntactic analysis. Cf.

also Franciotti and Lietzke [1].

Considerable attention has been paid to non-optimizing compilers which aim at short compile times; these are especially useful in university environments for use in the student programs. Rosen et al. [1] describes a fast FORTRAN load-and-gocompiler developed at Purdeu University, indicating the sequence and nature of the various compiler passes, the over-all layout of compiler code and data in core, etc. Shantz et al. [1] describes a FORTRAN IV load-and-go compiler (WATFOR) developed at the University of Waterloo. This is a quite successful high speed student-problem oriented FORTRAN compiler which incorporates a precedence parse following upon a fast statement-type determination using the FORTRAN key words. The parser calls binary into code generator routines which produce executable binary code directly. This compiler, running on the IBM 704, is capable of translating 6000 statements per minute. Arden, Galler and Graham [1] give a brief outline of a compiler for the University of Michigan algorithmic language MAD, which is an algebraic language having much the same flavor as FORTRAN but containing a number of interesting extensions.

Although many of the early ALGOL compilers were developed for small machines, the problems of compiling in this environment have not been covered in depth. Naur [4] describes a multipass ALGOL Compiler for a small machine. This compiler uses a precedence scan; about 50% of total compilation time is ascribed to lexical analysis, and about 14% to the (rather straightforward) generation of code. Haines [1] describes a FORTRAN compiler for an 8K IBM 1401. The compiler consists of 63 phases, each containing 150-300 instructions. The program is kept in core and continuously modified by these phases.

A special area of compiling that is receiving increasing interest in the literature is that of compiling programs to be run in an interactive environment. Here the principal concern is not efficiency, but rather flexibility. Two

qualities considered very important are the ability to modify the program as it runs and the ability to monitor its execution. The paper by Evans, Perlis, and van Zoeren [1] describes the use of threaded lists in producing (intermediate) code that is easily modifiable. Lock [1] describes a method of structuring programs so that local changes can be made without requiring global recompilation. The code produced is not machine code, but rather a simple interpretive code. Cf. also Ryan et al. [1] and Katzan [1] for variations and amplifications of Lock's work.

Only a few books dealing systematically with the construction of compilers have yet become available. The first of these was Randell and Russell [1] (cf. also Randell [1]), which gives a general description of an ALGOL compiler-interpreter system, with detailed algorithms and considerable attention to target-code questions. This book also contains brief descriptions of various of the arithmetic-statement analysis methods developed up to the time of its appearance. Unfortunately, the explanator strategy used by Randell and Russel (description of syntactic analysis methods in close involvement with code-generation issues relating to a particular machine) makes it difficult for their reader to sense the theoretical issues involved in syntactic analysis proper.

The excellent recent book of McKeeman, Horning and Wortman [1] gives a systematic theoretical and practical account of the well-designed XPL compiler-genrator system developed by them; cf. also the summary paper of McKeeman, Horning, Nelson and Wortman [1]. This system incorporates a generalized precedence parser making use of a (1,1) precedence parse capable of using supplementary (2,1) context as necessary. A small subset of PL/1 is provided for the expression of generative actions. A total compiler is described to the system by a grammar in Backus form which satisfies the restrictions imposed by the subsequent use of a precedence parse, and by a set of generator routines, one corresponding to each production of the grammar and invoked when the use

of this production is unambiguously recognized. A grammar pre-processing routine included in the system converts each input Backus grammar into a set of tables controlling the precedence parse, and, at the same time, checks that the grammar satisfies the conditions necessary in order that this be possible, issuing diagnostics concerning any syntactic errors that may occur in the grammar. McKeeman, Horning and Wortman also give an illuminating discussion of various devices for securing high efficiency of storage of necessary tabular information, and for increasing compile speed by keeping tables in appropriately sorted order. For an earlier account of these same ideas, cf. McKeeman [1]. A compiler for a PL/1 subset produced using the XPL compiler-generator attained a compilation rate of 3000 cards/minute on the IBM 360/67.

A book similar in spirit to that of Randell and Russell, but one which is somewhat more systematically worked out, is that of Grau [3], which describes an ALGOL compiler, and which contains a complete set of compiler algorithms written in a version of ALGOL extended by the inclusion of techniques for the description of target code. Mention should also be made of the book of Ingerman [1], which describes a top-down parser working directly from a tabular form of a Backus grammar and using certain additional tables for increasing the speed of the top-down parse; cf. also Ingerman [2] for a ALGOL program for such a parser. Cheatham [4] gives an introduction to many of the salient issues of syntax analysis and compiler writing more generally. In [5] Cheatham discusses many of the same issues; in particular, he gives a good account of how the symbol table is organized for one style of compiling. Lee's book [1] is a brief and elementary, but well written and highly readable introduction to the construction of compilers.

Hopgood [1] is a quite readable short survey of compiler-writing in general, and discusses many of the major topics belonging to this field, including lexical analysis and various methods of syntax analysis. Hopgood also discusses

a number of interesting topics in optimization of the sort to which Chapter VIII of this book will be devoted, especially those topics related to the optimization of linear code sequences or "basic blocks", and including methods for register allocation and for the elimination of stores in a single-register machine by the rearrangement of operation sequences in a basic block.

Rosen [1] is a well chosen reprint collection of papers including a number of the most noteworthy papers in the theory of compilers. The excellent survey article of Feldman and Gries [1] reviews the development of syntactic analysis methods, giving a technical account of various of the principal methods presently available, together with an extensive bibliography. This article also discusses questions relating to code generation. Rosen [3] surveys the early history of programming language and compiler development, with emphasis on the influence of hardware, and gives a useful bibliography.

Comments on Industrial Compilers

Compilers produced for serious commercial use must conform to high standards in a number of regards, and careful programming effort is often devoted to the optimization of such compilers. A commercial compiler must not be bulky; it must compile rapidly; it must be well debugged and documented. It must produce reasonably efficient output code. It must produce informative diagnostic messages, which catch most common programmer errors, and which do not require many passes over the source code to detect errors. A commercial compiler must be able to provide full listings, including such things as sorted symbol dictionaries, cross reference listings, allocation tables, etc. It must implement all features of a language in precisely specified form. It must be easily maintainable, which requires that appropriate debugging tools be built into the compiler, and that the coding practices used are such that changes made subsequently will not upset the whole compiler. It must be well integrated with the operating system of the machine on which it runs, make intelligent use of I/O, careful use of storage space, and use control statements consistent with other compilers running under the same operating system. It must provide output files in the precise form required by the system linking loader, and must conform carefully and effectively to all other system conventions regarding I/O, overlay handling, etc. (This problem of system interfacing often becomes a thorny point in an implementation of a commercial compiler, and a point to which surprisingly large amounts of effort come to be devoted.)

Precise definition of language is an issue often requiring considerable attention in commercial compiler design. Particular decisions concerning harmless looking language features, as, for example, rules regarding FORTRAN COMMON variables, may have serious consequences for optimization.

An important problem, often overlooked, is what to do about incorrect code: If an addition to the language is made to make a commonly made error legal, then compatibility with other past and future compilers for the same language may be affected. In compilers intended only for local use, all these issues may be settled in whatever manner is convenient for the moment. In writing commercial compilers, they often require careful attention.

Changes in machine structure may force slight variations in language, as e.g., the addition to a language of a class of half length integers. Calling sequences and storage allocation rules must be designed very carefully, as these usually affect the definition of the language itself.

In the following paragraphs, we shall outline some of the ways in which the writers of industrial compilers for higher level algebraic languages have attempted to solve their substantial design problems. Various existing compilers will be cited as examples. Although an attempt is made to note outstanding compilers, it is often true that a particular technique is best illustrated by an otherwise undistinguished compiler.

(a) Prescan and Lexical Scan.

Compilation speed can often be improved by pre-classification of statements by type; many FORTRAN compilers include a statement pre-scan having this goal. For ASA standard FORTRAN IV, this pre-scan can be accomplished by a finite state machine; such a technique is implemented in at least one existing compiler (CSC). For a FORTRAN which allows subscripted subscripts the finite state machine must be supplemented by at least a nesting level counter. A logically trivial issue, but one surprisingly important for efficiency, is the suppression of blanks, a function which can be accomplished during pre-scan. It has been found that comparing each input string word with a word

full of blanks so as to eliminate blanks in groups can increase the compile rate of simple compilers by up to 25 percent. This feature is found in most industrial compilers.

Lexical scan is normally not a separate pass in industrial compilers but is combined with first pass syntax scan. Typically, the syntax scanner calls the lexical scanner to obtain a next input segment in analyzed form. In most cases, the lexical scanner is a hand-coded routine, based on a table look-up of characters in one or more tables (the table dedpending on the current lexical state indicator). These tables reflect the structure of the language being compiled. The lexical scanner can therefore throw away comments, recognize and convert constants and keywords, etc. In any compiler, this tailoring of the lexical scan greatly increases efficiency. In a few cases (e.g., the SDS 910 ALGOL compiler produced by Programmatics, and all the CSC Genesis compilers) the lexical scan is based at least partly on a formal analytic grammar. The Genesis compilers allow the lexical scanner to interact to a certain degree with the syntax scan which follows along behind it.

(b) Syntax Scan.

The syntax scan accepts the source string, usually partially pre-digested by lexical scan, and produces as output a code string in some form -- often reverse Polish or a variant thereof. The number of "passes" used by a compiler will depend on the language, the space available to the compiler, and the quality of code desired. ALGOL, for example, is hard to compile in less than 2 passes, because uses of identifiers can appear before their definitions (and the definitions, when they do appear, can even influence parsing); a single pass ALGOL syntax scan is possible only at the cost of restricting the language, producing some rather bad code, or cleaning up the code later.

When a scan is done in two passes, the first essentially processes name definitions, and the second produces code in intermediate form. On the other hand, FORTRAN and JOVIAL are easily parsed in a single pass.

The syntax scan also produces almost all of the diagnostic messages developed by a compiler (these messages may be held in summary form for later printing, as in the CSC Genesis compilers, the CSC Univac 1107 FORTRAN compiler, etc.; nevertheless, the diagnostic information is developed during syntax scan).

A simple precedence scan, perhaps supplemented by some other method for classifying statements into types, is most commonly used in production compilers. This method has been used since before 1960; evidently the earliest production compiler to use it was that for Burrough's 220 ALGOL, a version of ALGOL 58 in a compiler largely designed by Joel Erdwinn. Since then, this standard scan has been used in the Univac 1107 and 1108 FORTRAN compilers, the IBM 360 FORTRAN H and TSS 360 FORTRAN compilers, and many others.

A number of industrial compilers (the SDS 910 ALGOL compiler written by Programmatics and the CSC Genesis compilers for ALGOL and JOVIAL) use parses defined by formal analytic grammars of the sort described above in connection with the Dömolki parsing algorithm. This scan method allows speeds comparable to that of the standard scan. The parsing methods available for generalized precedence grammars and based upon the use of precedence matrices have not been much used in industrial compilers, perhaps because of the large table sizes required; however, a JOVIAL compiler for the IBM 7090 written by SDC did use this technique.

Top-down analysis is used in a number of FORTRAN compilers. As noted above, this is the technique used in the DIGITEK POPS system.

(c) Optimization.

Many industrial compilers, especially those for small machines, omit optimization as a separate phase. In such compilers, what little optimization is done is often local, optimization being combined with syntax scan and/or code generation. By paying careful attention to machine details and using the simple technique of remembering register contents during code generation, it is possible to produce quite reasonable code without separate or global optimization. This level of optimization characterizes all the early Digitek compilers, the first crop of CSC Genesis compilers, most of the SDC JOVIAL compilers, the CDC 1604 ALGOL compiler, and many others.

In compilers aiming at a higher level of optimization, the exact degree of optimization performed may vary widely. Typically, common subexpressions will be eliminated with varying degrees of thoroughness, loop constant expressions removed from loops, and computations in loops will be reduced in strength. More will be said about optimization in industrial compilers following our later chapter on optimization techniques.

(d) Code Generation.

Code generation uses the symbol table to transform the internal form of a code to some representation of actual machine instructions. Except for the symbol table, only local information is used. Code generators typically consist of a rather large number of individual generation routines, corresponding to the various operators in the code generator input string. During code generation one typically performs a wide and miscellaneous variety of important machine-dependent object code optimizations. This includes various local coding tricks: deletion of an add of a zero or a multiply by a one; replacing a division by a constant with

a multiplication by the reciprocal; combination of constants; replacing by a shift a division by a power of two; combination of instructions as allowed by a machine order code set; and innumerable others desirable in virtue of the quirks of a particular machine.

The code generation process may be likened to conditional macro-expansion. A technique which may be used to speed the generation process is the preexpansion of conditional code macros into skeleton texts from which particular lines are subsequently selected by a system of indicator bits. This technique is used, for example, in the IBM FORTRAN H compiler.

In compilers not including a global register allocator, register assignment will often be combined with code generation. For some small machines, allocation may be a relatively simple problem, because only one or two registers are available; the Digitek SDS 910 FORTRAN compiler, for example, deals only with the contents of a single machine accumulator.

(e) Output Editing.

This phase produces the actual binary load modules representing generated object code, and may also produce an assembly language listing of the generating code. The "assembly" process, i.e., the substitution of addresses for symbols, is often made part of this phase. On the other hand, some compilers produce actual assembler input card images for submission to the system assembler, thereby avoiding the necessity for a separate edit phase. For production compilers, this technique is not to be recommended because of the speed sacrifice involved. It has, however, been used in the IBM 7094 and the BOS 360 FORTRAN compilers.

In a few compilers, storage allocation is deferred to the output editing phase. This approach is useful for compilers intended to produce code for more than one object

machine, and has been used in various of the CSC Genesis compilers. The edit phase of the compiler may also merge diagnostics produced by several previous phases into a source listing. The editor also produces any symbol tables used by the system for debugging and assorted symbol dictionaries and location maps used for hand debugging by the programmer.

Output editing is a surprisingly slow process, especially when a full output listing is produced. Generated instructions are likely to require quite a bit of processing to put them in final form, and most machines on which compilers run are somewhat ill suited to the character and bit string manipulations that an output editor must perform. It may be remarked that many industrial compilers produce either no listing of generated code or a listing in extremely unreadable form. Exceptions to this are the UNIVAC 1107 and 1108 FORTRAN compilers and the CSC Genesis compilers.

(f) Internal Tables and Their Treatment.

Symbol tables in commercial compilers are generally accessed using a hash-search technique; exceptions to this are the IBM FORTRAN H compiler and the Digitek POPS compilers. The POPS compilers use a tree storage method for source names. In this technique, a basic list contains all the characters which have occurred as the first character in a name, and for each such character a list containing all second characters which follow that particular first character in some name is available, etc. Where a hashing technique is used, the hash table is typically separate from the data storage area, and elements having identical hashes are chained together; this allows simple and efficient allocation of space for data storage. In an optimizing compiler, the same hash technique is often used to find formally identical subexpressions, and a dictionary of expressions may be combined with the symbol table. Most of the information stored within

a symbol table entry has a fixed length, and often a standard block of storage is reserved at the time an entry is made. Items often treated differently are the symbol names themselves (except in languages such as FORTRAN, in which name length is rigidly restricted) and such things as dimension information. In cases where name length is unrestricted, some compilers keep name tables separately; this has the disadvantage of splitting the available space into two parts. An alternate technique is to store each name just before the attribute information pertaining to it, and to keep the number of characters in each symbol name as a field in the corresponding attribute information area. This permits rapid access to attributes and also allows access to the name itself from the attribute block.

For languages which allow only a small maximum number of dimensions for an array, some compilers will allow space for the maximum number in every array entry. On the other hand, dimension information is often stored remotely, and referenced by a pointer field within the basic symbol table attribute block representing an array. An elegant technique used in the CSC Genesis compilers is to store the code sequence required for reference to an element of an array rather than array dimensions; this allows for rapid generation of code corresponding to occurrences of indexed arrays.

The symbol table technique used in a compiler will generally reflect the name-scoping rules of the language being compiled. For languages which, like ALGOL, permit nesting of name-scopes to arbitrary depth, all the entries corresponding to a single name can be chained together, the entries corresponding to innermost scopes being kept first in the chain and removed at scope-end; this provides the necessary recursive action. JOVIAL and those few FORTRANs which permit subroutines included within a main program can be treated by a similar technique but will have only

two scopes: main program and sub-program. Other FORTRANs, which require a strict serial order of routine and subroutines, can be handled simply by purging the symbol table of all but common block and subroutine names each time a subroutine is terminated; often some sort of entry-numbering scheme able to distinguish the current subroutine from the last few subroutines can be used to prevent over-frequent purging. A technique, useful when it is inconvenient or expensive to change list pointers in order to remove names in an inner scope from the range of reference of a given symbolic name, is to tag each symbol table entry with the number of the scope to which it belongs. A table of bits, each representing a particular scope, may also be kept, and the closing of each particular scope flagged. This makes it possible to locate the correct referent of a name as the first element in the chain of entries referenced by a given symbol which belongs to a live scope.

Most compilers use tables of fixed length except for their symbol tables, input and output files, and intermediate code files. A noteworthy exception to this is the storage scheme used the POPS compilers: here every table is always referenced through a pointer, so the tables can move freely. Whenever the block of space available for a table ("stack" or "roll") is exhausted, storage is reallocated and the collection of tables moved to create more contiguous space for the table about to overflow. This has the advantage of making maximum use of available core, which for the POPS compilers is typically very small.

All commercial multi-pass compilers make serious efforts to overlay sections of code and table areas no longer needed. Compilers running in a "paged" environment, i.e. in a hardware setting in which blocks of a certain size are moved automatically between main and secondary storage, find it important to use as few pages as possible, i.e. not to have

a number of pages part full if a smaller number of almost full pages would suffice. This consideration will in some cases dictate some special arrangement and sizing of tables and blocks of code. The IBM TSS-360 Fortran compiler was planned with this in mind.

(g) Compiler Flexibility and Debugging.

Except in compilers subject to rigid space constraints, it is reasonable to sacrifice a certain amount of space and speed in order to make the compiler modular and improve its debuggability. This aim will be easier to attain if a clean interface is maintained between all the compiler phases, i.e., if the number of tables, especially those passed between successive phases, is kept to a minimum. For example, in the Univac 1107 Fortran compiler and in all the CSC Genesis compilers the only information passed from the syntax analysis phase to the optimization phase is the symbol table, the code string, and half a dozen parameters. This approach is also to be noted in the IBM 7094 Fortran compiler, where only the assembly-format card file is passed between the syntax phase and the code generation phase.

Commercial compilers of medium to large scale will normally incorporate a few specialized tools used only for the debugging of the compiler itself. For example, the Univac 1107 compiler recognizes some unpublished "reserved words" as legal statement types; these cause to be printed a trace of the compiler analysis of the following statement. The CSC Genesis compilers include routines which dump compiler tables on demand in a form convenient for the compiler writer. These routines occupy part of the space allocated for the symbol table, so that when the compiler is operating in non-debug mode no space is required. An octal correction facility for parts of the compiler itself is also included.

The source string, various intermediate code strings, the final binary output string, and the listing file produced

by a compiler are often buffered in sections into external files, at least for compilations too large to fit into core. The careful buffering of external files, aiming at effective overlap between input-output operations and simultaneous in-core calculations can have an important effect on the performance of a compiler. Provision of adequate buffers may allow small compilations to proceed to completion without any external I/O becoming necessary.

The symbol table is normally kept in core throughout compilation. Compilers performing global optimization often require that much of an intermediate code string be kept in core during optimization, and often keep intermediate code in core while passing from syntax scan to optimization.

(h) The Choice of Language for Compiler Writing; Compiler-Compilers.

Until fairly recently, all commercial compilers except for the CSC JOVIAL compilers were written in assembly code; the highest performance compilers still are. The IBM 360 FORTRAN H compiler is written in an extended FORTRAN. The efficiency advantages of assembly code over source code are currently a matter of debate. Estimates of the performance advantage for an assembly-coded over source-coded compiler range from 50-100 percent, and estimates of the advantage in size of assembly-written over source-written compilers cover a similar range. Minimal space can be the most severe obstacle to the use of a source-written compiler (with the exception of POPS compilers, see below). However, the development of suitable systems programming languages and their optimization is expected to reduce the relative advantage of assembly-coded compilers very significantly, and one may look forward to increasing use of source language for the coding of compilers.

The desirable features of a compiler-writing language are:

(i) Efficient access to machine part words as variables in the language. This is very important if the compiler is to be able to use densely packed tables.

(ii) Data structures and allocation rules which permit control over placement of variables in memory including control of overlays, and which permit the combination of heterogeneous variable types in a single structure or entry (as in a symbol table).

(iii) Some form of based storage permitting convenient shifting of tables in core and allowing table structures to be extended to newly allocated blocks of core.

(iv) Name-scoping rules permitting easy combination of separate routines written by different people.

(v) Efficient and flexible calling sequences. In this connection, some of the ideas on subroutine linkage optimization discussed in a later chapter may be valuable.

(vi) Access to all machine instructions, hopefully in a form which does not obstruct global optimization of the compiler code. One way of providing this access is in terms of a package of subroutines.

(vii) At least a rudimentary system of macros permitting conditional compilation of the system source language should be provided. Such a tool is useful for a variety of purposes, including isolation of compiler parameters, avoidance of repeated and error-prone insertion of repetitive code blocks, and production of a number of slightly variant versions of a compiler.

(viii) It must be possible to initialize variables, and, in this connection, provision must be made for the convenient treatment of character and bit string constants.

(ix) Recursive routines are useful for the expression of a number of compiler processes.

PL/1 probably has more of the above features than any other language, but its calling sequences are inefficient, it does not give access in a convenient way to machine instructions, it may involve inefficient methods for access to data structures, and it is rather difficult to optimize. In general, PL/1 pays so high a price for its generality as not to be an ideal tool for the writing of efficient compilers. JOVIAL, FORTRAN, and ALGOL each have some but not many of the features listed above. The IBM BSL language and the CSC SYMPL language have more of these features, and systems programming languages of this sort may well become standard tools for compiler production in the next few years.

Formal compiler-writing systems, which have been the focus of a good deal of university research in the last few years, have not generally been used for the production of commercial-grade compilers. The two principal exceptions to this generalization are the CSC Genesis compilers, which, as noted above, use analytic grammars and an associated bottom-up parsing algorithm, and the compilers written using the Digitek POPS technique; the POPS compilers use a top-down syntactic scan. The Genesis system produces compilers of quite acceptable running efficiency which attain a considerable degree of machine independence. Genesis has by now been used to produce ALGOL compilers running on the IBM 7094, the UNIVAC 490 and 494, and the General Electric 635. A JOVIAL compiler running on the IBM 360 has been produced using the same technique. The POPS system which, as noted in the detailed description of it given in an earlier paragraph of the present section, involves interpretation of pseudo-instructions written in a dense format, can be used to produce very small compilers. It is also easy to move the POPS system from one machine to another by recoding the relatively simple POPS interpreter.

(i) Speed and Size of Data for a Few Principal Industrial Compilers.

We attempt in the following paragraphs to give an indication of the relative efficiency of a number of significant industrial compilers. Approximate figures are given in number of instructions executed per nominal source card. Where information is available, an indication of the level of optimization attained and the style in which the compiler was written is given.

Compiler	Language in which written	No. of Instructions Executed Per Nominal Source Card	Optimization Level
360 Fortran H	Fortran	37K	Good global
7094 ALGOL	Genesis	137K	Local
360 JOVIAL	Genesis	96K	Local
6600 JOVIAL	Genesis	100K	Local
7090 JOVIAL	JOVIAL	120K	Local
1604/3600 JOVIAL	JOVIAL	55K	Local
1107 JOVIAL	JOVIAL	10K	Local
1107/1108 FORTRAN	Machine	14K	Good global
TSS/360 FORTRAN	Machine	8K	Good global

The code produced for a typical arithmetic loop by a good optimizing compiler can out-perform that produced by a naive compiler by a factor of 4 to 1.

The following more detailed figures give a breakdown of the TSS FORTRAN compiler by phases.

TSS FORTRAN Compiler Breakdown by Phases. (Total Code 67K Lines.)

	Time	Code
Scan (including careful diagnostics)	30 %	30 %
Optimization (including register allocation)	28 %	18 %
Code generation (including local optizmiation)	28 %	45 %
Editing	6 %	7 %

(j) Additional details on various particular compilers.

In the present paragraphs, we outline the structure of a number of compilers; comparison of these descriptions will indicate the consequences of various possible design approaches and the influence on design of such external factors as source language and size constants.

We begin with an account of the IBM OS/360 FORTRAN IV compilers. IBM has produced three FORTRAN compilers to run under OS/360: the E compiler, which can be used on a machine with only 32K bytes of storage; the G compiler, which is designed for a 128K byte machine; and the optimizing H compiler, which requires that 256 bytes of storage be available. The compilers themselves do not use the entire memory, of course; several thousand bytes are occupied by the resident operating system.

Since the amount of memory available is such an important factor in determining the design of a compiler, it is interesting to compare the design of these three compilers.

The G compiler, whose design is most straightforward, occupies 80K bytes of memory. It was written for IBM by the Digitek Corporation and uses their POP technique, which is described above. The compiler consists of five logical phases which are, in general, always resident in core; if, however, additional space is required during processing, those phases not currently in use may be spilled. The first or parse phase transforms the source program into a roll (push-down stack) of instructions in Polish form and builds the dictionary, which is contained on a number of rolls. The second phase uses the dictionary information developed during parse to allocate storage for simple variables and arrays, format lists, parameter lists, etc. The third phase is an optimization phase. The optimization performed is not extensive; the compiler merely attempts to optimize register usage

for subscript quantities used in DO-loops. The fourth phase uses optimization information and the Polish roll to generate relocatable machine code. The fifth and final phase produces the actual decks and listings.

Essentially all of the storage used by the compiler during compilation is roll storage. This storage is acquired as necessary via the system allocator in 4096-byte blocks. A given roll is kept in consecutively-allocated blocks; this requires relocation of all rolls of larger index whenever a given roll fills up.

The G compiler produces rather cryptic, but effective, diagnostics. Whenever a syntax error is discovered, a "$" is printed under the character which caused the parsing routine to stop. If the routine can continue, it then does so; in this way it may happen that several positions in a statement may be marked with "$". For each position so marked, a one or two word diagnostic is given. For example:

```
                    ARY(J) = BRY
$                                $
```

(1) LABEL (2) SUBSCRIPT

The first diagnostic indicates, of course, that the compiler expected to find a label on this statement (perhaps it follows a GO TO statement). The second indicates that BRY appears in a DIMENSION statement and therefore requires a subscript.

The design of the E compiler, which operates in only 15K bytes of core, contrasts with that of the G compiler in an interesting way. The number of phases is larger, and they are less coherently organized. More explicitly, only a nuclear interface module is resident in main memory throughout compilation. This module performs compiler I/O and controls the loading of the other processing phases. It also contains a total compiler communications area. Other phases are loaded

singly, to be overlaid when they are no longer required. There are 12 processing phases, plus 3 so-called "interludes". These interludes are designed to recover space used by I/O processing routines and their associated buffers when certain intermediate files are no longer required. Since this operation is necessarily time-consuming, these interludes are executed only when the compiler must run in an absolute minimum of core.

The five phases of the G compiler correspond almost exactly to an obvious logical partitioning of a compiler. Consequently, all of the functions performed there are also performed by the E compiler. A given logical function is divided in the E compiler among several phases. The following chart relates functions to the phases that perform them.

Function	E Compiler Phases
Parse/dictionary build	8, 10D, 10E, 14, 15, 20, 30
Storage Allocation	10D, 12
Optimization	20
Code generation Assembly	12, 14, 20, 25, 30

Optimization in the E compiler is limited to optimizing register usage for subscript quantities occurring between referenced labels. Thus program flow structure is not considered.

The compiler first makes a pre-pass over its input to "reserve" keywords, remove embedded blanks, and insert blanks as separators so that the parsing phase can use a lexical scan.

The parse routine is divided into two phases: one to process declarative statements, which must appear at the beginning of the program, and another to handle executable statements. For the latter statements an intermediate text file is created. Diagnostic messages are encoded into this intermediate file for later processing. Expressions are only

checked at this point for well-formedness, so that a still later phase is required to modify the produced intermediate text to account for operator hierarchy.

Memory allocation for the E compiler is done in a single chunk. The amount of storage allocated is a function of the memory size of the machine on which the compiler is running. This storage space is used to contain the dictionary. Any main storage remaining is used to hold intermediate text; if there is not sufficient space in main memory to accommodate all of the intermediate text, it is spilled onto secondary storage.

The optimizing H compiler requires only 89K bytes of main memory in which to operate, although the compiler contains approximately 403K bytes of code. The compiler is divided into 5 logical phases, similar to those of the G compiler, which overlay each other; some of these phases are further subdivided into smaller overlay segments. Allowing space for dictionaries and intermediate text, the compiler requires a minimum of 150K bytes of storage.

Two factors account for the great expansion in the amount of code in the H compiler over than contained in the G compiler. First, the G compiler is written in the POPS language which is particularly compact. Second, the H compiler performs extensive global optimizations of a type not found in the G compiler.

The first H-compiler phase recognizes statement types and encodes the statements in an intermediate text of the operator-operand type. Simultaneously, it builds the dictionary. It is interesting to note that, like the E compiler, the H compiler only checks expressions for well-formedness, postponing detailed translation to a subsequent phase.

During the second phase storage allocation of programmer-defined variables and arrays is performed. In addition arithmetic expressions are changed from the operator-operand

format developed during phase 1 to a three-address format that properly reflects operator hierarchy. Each element in three-address format is of the form

op a b c ,

which represents

a = b op c .

Of course a, b, and c are references to dictionary items, either user-defined or temporary. Also at this point basic blocks and connectivity matrices are constructed as well as information on the usage of variables and constants within a block. This information is required for subsequent optimization phase.

Still following the pattern of the G compiler, the third phase of the H compiler is devoted to optimization. The H compiler can produce code "optimized" to one of three specified levels. At the lowest level minimal optimization is performed. This yields a stylized form of register allocation, with a fixed register for accumulation, another for subscripting, etc., and with redundant register loads and stores removed.

The two other optimization levels use related methods of register allocation. When the most highly optimized code desired, register allocation is done on a loop-by-loop basis, proceeding from innermost loops out. In this case, global subexpression elimination is also performed. The method of flow analysis and loop detection used by the H compiler are well summarized in Lowetz and Medlock [1] and are similar to the techniques explained elsewhere in this volume; consequently, we shall not discuss them here.

Register assignment, whether for a loop or for the whole program, proceeds in two phases: local assignment and global assignment. Local assignment is performed independently for each basic block. To be eligible for local assignment, a variable must be defined and used (in that order) within

the block. For variables that satisfy that criterion, a private register is assigned, assuming that one is available. If one is so assigned, it is "owned" by the variable from the definition point to the point of last use within the block.

Once local register assignment has been completed for all blocks in the loop/program, some eligible registers may remain for which no assignments have been made. These registers are then assigned on a global basis to the variables having the greatest number of references in the loop or program under consideration. When the highest level of optimization is specified, loops are processed from the inside-out, and, if a variable is eligible for global assignment in two nested loops, an attempt is made to assign it to the same register in both. When a variable is eligible for both global and local assignment, the global assignment takes precedence. Status bits are set for each of the operands of each text entry, indicating whether or not the operand may be assumed to be in a register and whether or not it must be retained in the register at the end of the operation.

When the highest level of optimization is specified for a program, three other types of optimization are performed.

i) common subexpressions are eliminated;
ii) code is moved out of loops, where possible;
iii) reduction in strength is performed.

Since the general methods used to perform these optimizations are described in a later chapter, we shall not go into detail here. There are certain aspects of the technique used which are of interest, however. For example, both local and global common subexpression elimination are based on formal identity. Thus the value-number scheme described in Chapter VIII is not used, even on a local basis. Moreover, no attempt is made to do constant propagation and indeed certain decisions concerning expansions of operators made during the first phase

make subsequent discovery of constant operands of little value. Of course, operations whose operands are seen without propagation to be constants are performed at compile time.

If the highest level of optimization has been specified, the user may request this phase to produce a structured source listing of the program. In the listing statements belonging to the same loop are so identified and nesting of loops is indicated by text indentation.

The fourth compiler phase uses the register allocation which has been set up to generate code. In code generation a rather interesting technique, amounting to a kind of conditional pre-examination, is used; this method combines compactness of data representation with high speed code generation. For each basic type of 3-address intermediate text entry operation a skeleton table describing the machine code instructions to be generated for it is specified. This table includes, in proper sequence, all possibly necessary register loads and stores, even though some of them may be redundant for a given text entry. Since System/360 allows both register-register and storage-register version of most arithmetic and logical operations, both forms are included in the skeleton. Furthermore, if the operation is commutative, both possible load orders are included in the skeleton. Associated with this skeleton text is a bit matrix containing as many rows as there are items of skeleton text. Using the status bits for the operands of an intermediate text entry, which we recall are set during the optimization phase, a column is selected from this bit matrix. The rows selected by 1's in this column define the skeleton entries which are to appear in the text entry expansion.

A final compiler phase is called only if errors have been detected during processing; its only function is to print these messages.

The Control Data corporation provides two very different FORTRAN compilers with their 6000 series computers: the FORTRAN RUN compiler, which attains a high compilation rate and good optimization of inner DO-loops, and the FORTRAN EXTENDED compiler, which emphasizes thorough machine-dependent optimization.

The RUN compiler, originally developed at CDC's Chippewa Laboratories, is a one-pass compiler, resident in core throughout a compilation. The object code it generates is stored immediately above the compiler code, and expands upward. Fixed-size buffers are kept at the top of the available core area; symbol tables start below these buffers and expand downwards. If the object code region ever overlaps the growing symbol tables, the compilation is aborted; no provision is made for storing intermediate data on external devices.

RUN's minimum core requirement varies between $33,000_8$ and $40,000_8$ (60-bit words), depending on the version of the compiler used.

All RUN compilers have allowed FORTRAN and assembly language subprograms to be intermixed in a source deck. In the Chippewa operating system, the assembler was an integral part of the compiler; in the more recent SCOPE systems, a separate assembler automatically overlays the compiler when assembly language code is encountered.

Code optimization, which played a very small role in the earliest RUN compilers, has improved with each new version. The compile-time optimization routines currently watch for sequences of statements, and in particular for inner DO-loops, which 1) contain no subroutine or function references, 2) include only assignment statements (or, under some circumstances, IF statements), and 3) can only be entered at the first statement of the sequence. When

such a sequence α is found, the compiler searches for operations which are performed several times or, in the case of a DO-loop, can be taken outside a loop. A "preamble" is generated immediately before the sequence to pre-compute expressions and array addresses and, for DO-loops, to load DO parameters and address increments into index registers. Next follows the code representing α after which is placed a "conclusion", which e.g., may contain instructions storing variables held in registers within α .

As the level of optimization provided by in the RUN compiler has increased, its compile speed has gradually decreased from approximately 20,000 cards/minute to a current rate of about 15,000 cards/minute. High compilation speed is attained largely by the use of direct, straightforward procedures for code generation, which often utilize fixed code sequences, sometimes with fixed register assignments.

FORTRAN EXTENDED, despite its name, does not include any significant language extensions beyond RUN FORTRAN. This compiler was developed by Control Data to take full advantage of the machine architecture of the 6600. It is a two-pass compiler which emits symbolic assembly code.

The first pass consists of two phases: Phase 1 processes declaration statements (which must precede all executable statements), building symbol tables and producing any necessary assembly language code. Phase 2 processes the remainder of a subprogram, adding to the symbol tables while generating an intermediate form of code, called RLIST. During the first pass, all references to a variable in an equivalenced set of variables are changed to references to a common "base variable"; this is necessary in order that the compiler may subsequently optimize references to

these variables during the second pass.

The symbol table used is of some interest. This table is maintained as a series of tree structures, accessed through a hash table. As symbols are added, each tree is rearranged to keep it as symmetric as possible; as a result, not more than four or five entries need be examined to locate an element in a thousand entry table.

The second compiler converts the RLIST intermediate code into assembly language code. During this process two basic types of optimization are performed: DO-loop optimization and code sequence optimization. "Well-behaved" DO-loops (containing no function references or unconditional jumps, and not having extended range) are analyzed to determine an optimal counting strategy, and to pre-load useful data such as DO-loop parameters and array addresses into index registers. In version 1 of the compiler, however, no attempt is made to take loop-invariant calculations outside of DO-loops.

A sequence is a series of instructions delimited by an unconditional jump, active label, or conditional jump terminating a DO-loop; it is the unit within which optimization is performed. Each sequence is optimized in three phases: First, all redundant operations in the sequence are deleted. Second, a dependency tree (similar to those used in PERT) is built and the operations comprising a code are ordered on a priority basis (e.g., an operation whose result is required for all other operations is given top priority). Finally, code is emitted. During the final phase, a careful timing analysis is made to determine the availability of registers and functional units (the 6600 contains 10 independent functional units, which can operate concurrently). Proper scheduling of functional units and registers allows reduction of the time required to execute a sequence by twenty to fifty percent.

Version 1 of FORTRAN EXTENDED compiled relatively slowly about an order of magnitude more slowly than RUN. About two-thirds of the compilation time was used by a standard two-pass assembler to assemble the code generated by the compiler. This defect was remedied in version 2, released in 1969, which includes a high-speed assembler as part of pass 2 of the compiler; the compiler pre-computes the addresses for all symbols, so that only one pass is required for final assembly. This improvement lay at the basis of the three-fold increase in compile speed from version 1 to version 2.

Control Data's current plans call for development of a third version of FORTRAN EXTENDED which is to allow a rapid compilation option (no optimization) and a global optimization option performed by the methods used in IBM FORTRAN H.

COBOL compilers, as a class, have a rather different structure from that of the FORTRAN compilers described above. This structure seems to have been influenced both by the history of COBOL and by the nature of the language itself.

In order to appreciate the style of translation employed in COBOL, it is necessary to have some idea of the structure of a COBOL program. Such a program is written in four sections, called Divisions. The Identification Division is used only to identify the program and to describe its use; we shall not be concerned with it in our discussion of the translation process. Of the other three Divisions, two are used to describe the nature of the data used by the program, particularly the files; the third contains the executable statements of the program. More precisely, the Data Division contains the description of files as they are seen by the user; for example, the buffer areas for each file are laid out and the access method for each file is described. The Data Division

also contains descriptions of all non-file variables. The Environment Division contains a description of the physical aspects of a file - the kind of device on which the file resides, its identification number and extent, etc. Finally, the Procedure Division contains the executable part of the program.

From this brief description it is obvious that compiling a COBOL program involves merging the information in the Data and Environment Divisions to form a complete description of the files and working storage areas of the program and then using this description to produce code for the program given in the Procedure Division. This the OS/360 COBOL(F) compiler manages to do, using six processing phases, four intermediate files, and a dictionary.

During the first phase the data descriptors from the Environment and Data Divisions are encoded and spilled out onto an auxiliary file. The Procedure Division text is reduced in a crude way, basically to the lexical level. Identifiers are kept in this text literally; only labels are entered into the dictionary. The procedure text is spilled out onto a second file.

The second phase uses the file of data descriptions to produce a dictionary. It also does a certain amount of storage allocation. Using the dictionary produced, the third phase reads the procedure text file and produces another file of procedure text, resembling the first, except that variable names have been replaced by their attributes from the dictionary. The dictionary is then discarded.

The fourth phase performs syntax analysis on this file. The result is another text file. The fifth phase uses this file to produce code in an assembly-like language. At the same time the literal table and the label table are each sorted and identical entries removed. The assembly language code is also spilled. The last phase assembles the program and produces the object module.

The IBM OS/360 COBOL (E) compiler is organized on a functional basis; however, because of the small amount of storage available for the compiler, each of the major functions is implemented using several overlay phases thus requiring frequent and repeated scanning of the (modified) text.

Overall, there are three major components: a data component, a procedure component, and a code generation component. The data component processes the ID, Environment, and Data Divisions; builds the dictionary; and allocates storage for the variables. The procedure component analyses the statements of the Procedure Division and creates an intermediate text from them. The final component generates code from this intermediate text.

Each of these components is implemented by a number of phases which overlay each other. Each of these phases performs a small amount of the total processing for the component. The data component contains eight phases, the procedure component six, and the code generation component nineteen.

The procedure component contains four primary processing phases, each of which is responsible for the translation of a given set of COBOL verbs. During each of these phases, the text, which is kept on secondary storage, is scanned in a linear fashion, and the statements corresponding to a given set of verbs are converted to intermediate form. Thus four scans of the intermediate text are required. Similarly, the code selection phases of the code generation component require ten passes of the intermediate text data set. This component also contains six phases devoted to final assembly of the program.

The OS/360 PL/1 (F) compiler has an overall design that closely resembles that of the COBOL (E) compiler. That is, it contains a small number of logical phases, each of which is subdivided into a series of physical phases. Each physical phase performs some small part of the total processing of the logical phase. This processing generally results in some modification to the current text of the program. In this way the source text is gradually transformed into the object text. There are eleven logical phases, with the functions explained below; there are over one hundred physical phases.

Control of the phase loading resides in a separate set of modules called the control phase. These modules are resident throughout compilation. Not all phases need be loaded for a given compilation; those phases concerned with language features not used in the program will be bypassed. The control phase is also responsible for all of the compiler I/O and for all internal storage management.

It is in the area of storage management, both primary and secondary, that this compiler differs most from COBOL (E). As we observed above, the COBOL (E) compiler keeps only the dictionary in primary storage. Indeed, the dictionary is never spilled. The program text, on the other hand, is kept in auxiliary storage. Every time it is scanned and modified, it is read in and written out again. Such a scheme has two obvious disadvantages; an especially large program requiring a large dictionary may cause the compilation to abort, and the compiler cannot take good advantage of extra primary storage that may be available for a given compilation.

The storage management scheme employed by the PL/1 (F) compiler is much more flexible. It permits compilation to take place when only limited space is available. On the other hand, if sufficient space is available, both text

and dictionary may be kept in primary store, with no spill at all. This is accomplished by employing a "software paging" scheme for both dictionary and text. In such a scheme, an address space large enough to hold the text for any anticipated program is used. This is sometimes referred to as a "virtual store." This space is generally larger than the amount of real storage space that can be devoted to text and dictionary entries. The virtual store is therefore subdivided in a uniform manner into blocks, called pages. The size of a page depends on the amount of memory available for the compilation; it is chosen so that at least two dictionary pages and two text pages can be accommodated in memory at the same time. Reference to an element in the virtual store is made via a subroutine. Using a page directory, this routine checks to see if the page containing the element is in storage. If it is not, a page is moved out to make room and the required page is brought in. The actual address of the element requested is then calculated and the element is accessed.

Such a scheme makes addressing rather uniform, but only at the expense of making each reference expensive. The cost can be cut down, however, by freezing a given page in core at a critical point and making the references indirectly.

It is interesting to look in some detail at the logical phases of the compiler, since their organization illustrates the relative complexity of a PL/1 program, when compared to either FORTRAN or COBOL.

1. Macro phase.

   This phase executes the compile-time statements and performs text replacements. The output is a PL/1 program. This phase is logically independent of the rest of the compiler.

2. Read-in phase.
This phase performs statement recognition and puts the statements into text pages in internal form. This form closely resembles the external form with keywords and redundant punctuation removed. This recognition activity requires five passes of the text. During this phase, all procedure and block entry points are chained together, as are all declarative statements.

3. Dictionary phase.
Using the two chains constructed in the previous phase, this phase first processes the explicitly declared variables (since PL/1 is a block structured language, one needs to know not only what variables have been declared, but in what block or procedure the declaration was made). Then the text is scanned for contextual declarations of variables (e.g., a variable used in a context where only a file name is permitted). Finally, using a fixed-floating convention similar to FORTRAN's, the attributes for all other variables mentioned in the program are filled in. Then each identifier in the text is replaced by its dictionary reference.

This logical phase contains sixteen physical phases.

4. Pretranslator phase.
Still working at the text level, this phase attempts to simplify the task of the remaining phases by rewriting certain PL/1 constructions in terms of other PL/1 or PL/1-like constructions. For example, array and structure assignments are expanded into Do-loops. Expressions used as arguments to subroutines

are assigned to temporaries. The I/O statements are modified and simplified. Finally certain debugging statements, if requested, are inserted into the program.

5. Translator phase.
These phases modify the text format in a significant way. All executable text is converted to "triple" form: an operator followed by two operands. At the same time, GENERIC function references are particularized to the appropriate family member. Also, certain string operations are marked as optimizable (read: amenable to in-line translation).

6. Aggregates phase.
At this point accessing functions for the elements of arrays and structures are computed, assuming that the size of the aggregate is known at compile time. For arrays and structures with bounds and lengths adjustable at run time, code is inserted into the text to calculate the function at run time. Uses of equivalencing statements are also checked at this point to insure that language rules are not violated.

Note that storage is not allocated during this phase.

7. Pseudo-code phase.
These phases make a second important change in the text format. At this point the text is replaced by a machine-like code that is essentially one-for-one with machine instructions. Registers are left in symbolic form; the last use of a register is noted by the appearance of the instruction DROP in the instruction stream.

8. Storage allocation phase.
Storage is allocated during this phase not only for source program variables and compiler-generated temporaries, but is also for data descriptors of various sorts: dope vectors, run-time symbol tables, flags, etc. Furthermore code is inserted into the text to allocate space for those objects whose allocation must be deferred until run time. In particular, the code for procedure and block prologues is generated.

9. Register allocation phase.
This proceeds in a straightforward manner with no particular attempt at optimization.

10. Final assembly phase.
This produces the final object deck.

11. Error editor phase.
Diagnostics are handled in an interesting way in this compiler. Every time a situation is encountered requiring a diagnostic, an entry is made into the dictionary one one of four chains, corresponding to the severity code of the diagnostic. This entry contains the number associated with the diagnostic as well as some skeletal information identifying the trouble spot; e.g., the line number in which the trouble was encountered and perhaps a small bit of text from the area of the trouble.

When the compilation has been completed, the error editor phases load in the full text of the diagnostics and expands the error dictionary entries into readable prose. These "full" diagnostics are then placed in the user's output file.

CHAPTER 5. RIGOROUS RESULTS CONCERNING THE PRINCIPAL SYNTACTIC ANALYSIS METHODS.

1. Turing Machines and Backus Grammars.

In the present chapter, we shall prove a variety of theorems which delimit the relative power of the various syntactic analysis methods introduced in the four preceding chapters. We shall see that some of the methods are substantially more general than others, e.g., that the bounded context syntactic methods of Chapter 4 are in fact not capable of analyzing as wide a class of languages as can be treated by the general recursive method described in Chapter 2. These results will be derived from the theory of Turing machines, and specifically from the proven existence in this theory of iterative symbolic processes for which certain problems are unsolvable.

Our approach will be as follows. We shall show that, using certain linguistic constructions, one can describe the most general Turing machine. Then, by showing that certain problems are solvable for context free languages, and by noting that these problems are unsolvable for general Turing machines, we shall show that a class of formal constructions beginning with languages describable by grammars in Backus normal form necessarily passes out of the domain of these languages. In this way, we will show that certain classes of strings are not describable by Backus normal form grammars, and, more generally, that certain processes describable by such grammars are not describable by grammars that can be parsed by the restricted syntactic method described in Chapter IV.

We begin with the theory of Turing machines. A Turing machine is by definition an automaton which, at each moment, exists in one of a finite collection of internal states $\{\sigma\}$ and which iteratively reads and writes a two-sided infinite

tape divided into discrete squares. Each square of the tape is occupied by one of a finite collection of print characters $\{c\}$. The tape is read and written by the automaton one square at a time. After reading and writing a given square, the automaton may stay in the same position on the tape and reread or rewrite the same square; alternatively, it can advance one square to the left or one square to the right and read and/or write the new square. We require that, after the completion of an elementary read-write operation, the next internal state of the Turing machine, together with its direction of motion, be determined entirely by its previous state and by the character just read. Similarly, the character which the Turing machine writes in a square must be a determinate function of the state of the Turing machine and of the contents of the square just read. We assume throughout that all but a finite number of the squares of a Turing machine tape initially contain some special one of the finite vocabulary of print characters, which we call the blank character; this assumption will play a special role in the following discussion.

According to the above, a Turing machine is described by three functions of two variables each:

$$(1) \qquad p(c, \sigma);\ s(c, \sigma);\ m(c, \sigma) \ .$$

In (1) c denotes an arbitrary character chosen from the finite vocabulary of print characters; σ denotes the state of the machine which must be one of a finite set of possible states. The function p, which is the print function of the Turing machine, defines the character which the machine will print in the square which it is scanning if the character originally in that square is c and if the machine is in state σ. The function s, called the state function, defines the state into which the machine will pass on scanning the character c while in the state σ. The function m, called the move function, has one of the three possible

values -1, 0, or +1, depending on whether the Turing machine on scanning the character c while in the state will move one square to the left, will not move, or will move one square to the right.

Each step, and thus inductively the whole action of the Turning machine, is defined by what may be called its configuration, i.e. by its momentary internal state , the set of characters present on its tape at a given moment, and by the particular character which the Turing machine momentarily scans. This machine configuration may be represented in a form appropriate for our purposes as a state word of the structure shown below:

$$(2) \qquad S = c_1 \cdots c_k \, \sigma \, c_{k+1} \cdots c_n \, .$$

In (2), S denotes a state, $c_1 \ldots c_k$ denotes a sequence of print characters written on the Turing machine's tape, σ denotes its internal state, and $c_{k+1} \ldots c_n$ are additional print characters written on the tape. We place the state character σ immediately after the tape character c_k to indicate that the machine is momentarily scanning the particular character c_k. That is, in writing the state word (2), we place the internal state character for a Turing machine immediately after the particular print character scanned by the automaton.

In order that a finite word of the sort shown in (2) property describe the whole configuration of the infinite tape attached to a Turing machine, we adopt a special convention according to which leading and trailing blanks are suppressed. More precisely, in writing the state-word (2), we omit any blank character preceding the first non-blank character on the Turing machine tape and any blank character following the last non-blank character on the Turing machine tape, with the exception that if any blank character lies between a non-blank character and the square currently scanned by the Turing machine, all blank squares up to and including the scanned square must be indicated explicitly. With these conventions, state words of the type

shown in (2) will give a description of the state of an arbitrary Turing machine **complete in all** necessary essentials.

The iterative motion of a Turing machine from state to state and square to square may then be represented as an iterative evolution of the state word describing the Turing machine. The three possible elementary evolution steps may be shown as follows:

$$(3)\quad c_1 \cdots c_k \quad c_{k+1} \cdots c_n \longrightarrow c_1 \cdots c_{k-1}\, c_k'\, \sigma'\, c_{k+1} \cdots c_n$$
$$\longrightarrow c_1 \cdots c_{k-1}\, c_k' c_{k+1}\, \sigma'\, c_{k+2} \cdots c_n$$
$$\longrightarrow c_1 \cdots c_{k-1}\, \sigma'\, c_k c_{k+1} \cdots c_n .$$

On the left hand of (3) a typical state word S (identical to that shown in (2)) is written. On the right hand side of (3) we indicate the three possible words into which S might evolve in a single step of the Turing machine. The first of the state words on the right of (3) depicts an evolution step involving rewriting of the character c_k and change of internal machine state, but not accompanied by any motion of the Turing machine along its tape. The second word depicts a step involving by motion one square to the right. The third state word shown in (3) depicts a step involving rewriting of the character c_k and change of interval state, accompanied by motion of the automation one square to the left.

Given an initial state word, a Turing machine iteratively produces a sequence of state words of the type shown in (2) by elementary steps of the form shown in (3). This sequence of state words may be written in the form

$$(4)\qquad S_1;\ S_2;\ \ldots;\ S_k\ .$$

A sequence of state words of this kind may be called an orbit or path for the Turing machine. The evolution problem for a Turing machine may be described as follows. Suppose that a Turing machine is given. Suppose also that this Turing machine possesses a finite state of print characters c , a finite collection of internal states $\{\sigma\}$, and that

the elementary steps of which the Turing machine is capable is described by a finite set of elementary productions of the form (3). Find an algorithm which, given an initial state word S of the form (2), is capable of predicting by some determinate algorithmic process whether or not the Turing machine will ever evolve into a state (5) in which σ' is required to be some fixed internal state character of the Turing machine.

(5) $$S' = c'_1 \ldots c'_{k'} \; \sigma' \; c'_{k'+1} \ldots c'_{n'} \; .$$

The evolution problem that we have just stated is recursively unsolvable. That is, no algorithm with the specified properties exists. The non-existence of such an algorithm is a basic statement in the theory of recursive functions. We shall not attempt to give a detailed proof of this principle here, but shall instead refer to the books of Davis (Computability and Unsolvability) and Minsky (Computation: Finite and Infinite Machines) in which extended and particularly lucid discussions of this point will be found. We may, however, sketch a basis for this principle rather briefly in the following way. It can be seen rather readily by explicit formal constructions that a certain particular Turing machine of the sort which we have described is capable of simulating an arbitrary computer. Such an automaton is called a universal Turing machine. Since almost by definition a formal algorithm is a process that can be be programmed for a computer we may reason by contradiction roughly as follows. Suppose that an algorithm of the kind described above did exist. Then it could be programmed for a computer. The action of the computer under the control of its program could in turn be simulated by a Turing machine. But using a version of the Cantor diagonal process, we can construct a Turing machine which deviates in its action, at least for one initial state, from the result predicted by any single algorithm. This contradiction establishes the nonexistence asserted above.

A stop state of a Turing machine is a state $\hat{\mathfrak{d}}$ of the machine with the following special property: once the machine passes into the internal state $\hat{\mathfrak{d}}$, it will remain through all succeeding steps in the same state and will cease to move along its tape. Moreover, once it has entered the state $\hat{\mathfrak{d}}$ it will never change the contents of the square on the tape which it is scanning. This description may, of course, be summarized more briefly by the single statement: once the machine enters the state $\hat{\mathfrak{d}}$, it stops. For this reason we call $\hat{\mathfrak{d}}$ a stop state. The basic result on the nonexistence of an algorithm stated above may be put more sharply as follows. There exists a Turing machine with a finite vocabulary of print characters and of internal states, possessing a stop state in the above sense, and for which there exists no algorithm capable, given an arbitrary initial state of the Turing machine and its tape, of deciding whether or not the Turing machine will ever enter the stop state. The unsolvable decision problem described by the preceeding sentence is sometimes called the halting problem for Turing machines. The halting problem is of course a special case of the equally unsolvable state transition problem described earlier. We will find it convenient in what follows to make use of the fact that not only the general state transition problem but also the halting problem is algorithmically unsolvable. We leave it to the reader to deduce the stronger statement from the weaker.

In order to relate Turing machines to languages and their grammars in the most convenient way, we shall find it convenient to rewrite the sequence (4) of state words of a Turing machine in a very slightly modified form. The modification we require is described simply by the rule: reverse the order of characters in every even numbered state word, but not in any odd numbered state words. If this elementary transformation is performed, the sequence of state words which we obtain may be called the alternating sequence of state words for the Turing machine. As an alternative term for "alternating sequence of state words"

we introduce the term <u>track</u>. To be more precise, we make the following definition. If (4') is any sequence of characters, then the sequence of characters shown on the right of (4") is called its reverse.

(4') $\qquad W = a_1 \ldots a_n$

(4") $\qquad W^r = a_n \ldots a_1$.

Writing W for the sequence (4') we indicate its reverse as in (4") by the character W^r. Given the sequence (4) of state words of a Turing machine, the track of the Turing machine is then represented by the sequence (6).

(6) $\qquad S_1; S_2^r; S_3; S_4^r; \ldots; S_{2k-1}; S_{2k}^r$.

As (6) indicates, it is convenient in what follows to restrict <u>tracks</u> to consist of even numbers of words; thus we always assume that the last word in a track has the form S_{2k}^r.

The successive words in a track are separated by semicolons in (6). It is convenient for us to make this a formal convention. That is, we assume formally that "semicolon" is a character not present in the vocabulary of print words of our Turing machine (and also not representing an internal state of the Turing machine); and we assume that a Turing machine track is always written formally as a sequence of reversed and nonreversed state words, each separated from the next by a semicolon. In this way, we represent an entire Turing machine track as a composite word. We shall now show that the class of words which can occur in this way may be described completely by a pair of Backus Normal Form grammars, and that these grammars may easily be written in terms of the scheme of productions defining the Turing machine.

Indeed, a track consists of a sequence of pairs of words (an odd numbered and even numbered word making up a pair) successive pairs being separated by semicolons, and the first word of a pair being separated from the second word of the same pair by a semicolon. Each word of such a pair is obtained from the preceding word by an elementary

production of the Turing machine and by a reversal. Thus, every pair has necessarily a form of the following kind:

$$(7)\quad \underbrace{c_1 \cdots c_{k-1}}_{f}\ \underbrace{c_k\ c_{k+1}}_{t}\ \underbrace{c_{k+2} \cdots c_n}_{b}\ ;\ \underbrace{c_n \cdots c_{k+2}}_{b^r}\ \underbrace{'c_{k+1} c_k'}_{t'}\ \underbrace{c_{k-1} \cdots c_1}_{f^r}$$

In (7) the substring of characters marked f and f^r represent respectively the front substring of the first word of the pair, which is not modified in the transition from a state to the next succeeding state, and the reverse of this front substring. The section of the strings (7) marked t and t' are respectively the group of three characters including the internal state character for the Turing machine which is subject to modification in elementary transition; and the transformed form of this same group of three characters. Finally, the section of the pair of state words (7) marked b and b^r are respectively the back substring of the Turing machine state word, which is a part the state word not subject to transformation in an elementary Turing machine production, and the reverse of this back substring.

The above description makes it plain that the structure of a track may be described by a rather simple grammar of Backus Normal Form. The necessary Backus grammar is shown below,

$$(8)\quad \begin{aligned} &\langle \text{track} \rangle = \langle \text{pair} \rangle \mid \langle \text{pair} \rangle\ ;\ \langle \text{track} \rangle \\ &\langle \text{pair} \rangle = \langle \text{middle} \rangle \mid c\ \langle \text{middle} \rangle\ c \mid d\ \langle \text{middle} \rangle\ d \mid \ldots \\ &\langle \text{middle} \rangle = t_1 \langle \text{end} \rangle t_1' \mid t_2 \langle \text{end} \rangle t_2' \quad \ldots \\ &\langle \text{end} \rangle = ; \mid c\ \langle \text{end} \rangle\ c \mid d\ \langle \text{end} \rangle\ d \mid \ldots \end{aligned}$$

In (8) the characters c, d, etc. occurring in the definer on the right hand side of the definition of the syntactic element <pair> are intended to be a listing of all possible print characters of the Turing machine c. The characters c, d, etc., occurring in the definer on the right hand side of the Backus definition for the syntactic type <end> are, in the same way, a comprehensive list of all the print characters for the Turing machine which the

grammars (8) are intended to model. The doublets t_1, t_1'; t_2, t_2';... occurring on the right hand side of the definition of the syntactic element <middle> are intended to be a listing of all the separate elementary production triples $c_k \sigma c_{k+1}$, that appear among the Turing machine productions (3), each taken together with the result triple $c_k' \sigma c_{k+1}$, etc. into which a single Turing step will transform $c_k c_{k+1}$.

Notice now that not every sequence of pairs separated by semicolons forms a valid Turing machine track. Indeed, the condition that the whole sequence (6) be built up out of pairs (in the sense of the grammar (8)) is precisely equivalent to the condition that the even element of every pair in (6) consisting of odd and a following even element be derived from the preceding odd element by a reversal and an elementary production of the Turing machine. But, in order that the sequence (6) be a valid Turing machine track, we require in addition that every odd element but the first in (6) be derived from the preceding even element by a reversal and an elementary production of the Turing machine. The structures (6) satisfying this latter condition may also be defined by a grammar in Backus normal form. Indeed, this latter condition amounts to nothing more than the condition that the structure (6) consist of a valid state word of the Turing machine followed by a semicolon, followed by a sequence of "reverse pairs", followed by a semicolon, followed by a final state word. In this context, we mean by a "reverse pair" a pair of words of which the first is the reverse of a valid Turing machine state word and the second is a valid Turing machine state word derived from the first by reversal and by an elementary Turing transformation. Such a "reverse pair" evidently has a structure closely corresponding to the structure shown in (5); the collection of all such reverse pairs may be defined by a grammar in Backus normal form in a manner very similar to that used above to define the collection of all possible

(nonreversed) pairs. It follows therefore that the collection of all those structures (6) in which every even numbered element and the next following odd numbered element forms a reverse pair may be defined by a Backus normal form grammar. A grammar accomplishing this and specifying also that the track in question must lead to a stopped state of the Turing machine is as follows,

(9) ⟨otrack⟩ = ⟨word⟩ ⟨mtrack⟩ ⟨eword⟩
⟨mtrack⟩ = ⟨opair⟩ | ⟨opair⟩ ⟨mtrack⟩
⟨opair⟩ = ⟨omiddle⟩ | c ⟨omiddle⟩ c | d ⟨omiddle⟩ d | ...
⟨omiddle⟩ = t_1' ⟨end⟩ t_1 | t_2' ⟨end⟩ t_2 | ...
⟨end⟩ = ; | c ⟨end⟩ c | d ⟨end⟩ d | ...
⟨word⟩ = ⟨printc⟩σ⟨state⟩ ⟨printc⟩ | ⟨printc⟩ ⟨state⟩⟨printc⟩⟨word⟩
⟨eword⟩ = ⟨printc⟩ $\hat{\sigma}$ ⟨printc⟩ | ⟨printc⟩ $\hat{\sigma}$ | ⟨printc⟩ ⟨eword⟩
⟨state⟩ = σ_1 | σ_2 | σ_3 | ...
⟨printc⟩ = ⟨c⟩ | ⟨d⟩ | ...

In (9) $\sigma_1, \sigma_2, \sigma_3, \ldots$ constitute a comprehensive listing of all the internal state characters for our Turing machine. The characters c, d, and so forth, constitute a comprehensive listing of all the print characters for our Turing machine. The character σ_0 is intended to be the unique stop character of the Turing machine. The pairs t_1, t_1', t_2, t_2' occuring in the definer for the syntactic type omiddle , are the same pairs as occur in (7) in the definition of the syntactic type middle .

The words which are grammatical according to both of the grammars (8) and (9) are then precisely those tracks of the Turing machine which lead to a stopped state of the machine. Note that our two grammars are constructed in an entirely elementary way using the print characters, internal state characters, and elementary productions of the Turing machine. We may therefore assert that the set of those tracks of a Turing machine which lead the machine to a stop state may be defined as the set of words which are grammatical according to both of two Backus normal form grammars.

A terminal character of a Backus grammer is a character

occuring in one or several definers but not occuring as a definition. Note that the two grammars described above share the same set of terminal characters.

Using the above observations, the basic Turing unsolvability theorem stated at the beginning of this chapter leads immediately to the following unsolvability result.

Theorem 1. There exist two Backus grammars, both involving the same set T of terminal characters, such that there exists no mechanical algorithm which, given a string w of terminal characters, will decide whether there exists an additional string v of terminal characters such that w v is a grammatical string of both languages.

The words w v occurring in Theorem 1 may be considered to be "puns" which have the unusual property of being grammatical in two different languages. For this reason it is reasonable to call the problem whose unsolvability as stated in Theorem 1 as the pun problem, and to regard Theorem 1 as asserting the algorithmic unsolvability of the pun problem.

2. Problems unsolvable for Backus grammars.

We now aim to apply the unsolvability of the pun problem stated in Theorem 1 at the end of the preceding section to obtain additional formal conclusions. It is therefore well for us to begin by stating certain basic definitions concerning Backus grammars somewhat more precisely than we have yet done. We first define a Backus language or a context free language as follows. The language is specified by a grammar; a grammar is an unordered collection P of productions of the form

$$c \longrightarrow c_1 \cdots c_k \tag{9}$$

A production written in this way specifies that a single character (representing a syntactic type of the grammar may be rewritten as a string of characters. We also require that precisely one nonterminal character c among all the characters occuring in the productions of P be specified as the base character of the grammar. A character occurring on the right hand side of one of the productions (9) but not occurring on the left hand side of any production is called a terminal character of the grammar. Let a string $c_1 \ldots c_k$ consisting of characters of the grammar be given. We say that a string obtainable from this string by replacing any of the characters of the string by a set of characters which may validly replace it according to one of the productions (9) of the grammar is obtained from the original string by immediate descent. If S_1 and S_2 denote two strings, and if S_1 is in this sense obtainable from S_2 by immediate descent, we write $S_1 \longrightarrow S_2$. A sequence of strings S_1, S_2,.. each of which is obtained from the previous string by immediate descent is called a grammatical evolution, and may be written in the manner shown in (10).

$$S_1 \longrightarrow S_2 \longrightarrow \cdots \longrightarrow S_k \cdot \tag{10}$$

In this case we say that S_k is obtained from the string S_1 by indirect descent; the relationship of indirect descent may be written in abbreviated form in the manner shown in (11).

$$S_1 \longrightarrow\!\!\!\!\longrightarrow S_k \cdot \tag{11}$$

The set of all strings which are obtained by indirect descent from the base character of a grammar and which consist exclusively of terminal characters of the grammar is known as the language, the context-free language, or the set of grammatical sentences derived from the given grammar.

In order to apply Theorem 1 of the preceding section, we must derive a number of set theoretical properties of the languages defined by Backus Normal Form grammars built up out of finite collections of basic grammatical productions of the type (9).

The first of the results which we require may be stated as follows. Lemma 2. The set-theoretical union of two languages involving the same set of terminal characters is a language.

The proof of Lemma 2 is easy. Let Γ_1 be a grammar generating the first language. Let Γ_2 be a grammar generating the second language. By renaming the nonterminal symbols occurring in the grammar Γ_2 as necessary, we may readily insure that all the nonterminal symbols of Γ_2 are distinct from each of the nonterminal symbols of the grammar Γ_1. Then, by replacing the base character of Γ_2 by the base character of Γ_1, we may insure that Γ_1 and Γ_2 have the same base character, but that, aside from this base character, Γ_1 and Γ_2 have no nonterminal character in common. Given this situation, we simply form the set theoretical union of the collection of productions (9) which constitute the grammar Γ_1 with the corresponding set of productions constituting the grammar Γ_2. This "union grammar" we call Γ.

Now we note that, in view of the manner in which Γ has been defined, every derivation (10) in which the string S_1 consists only of the base character common to the two grammars Γ_1 and Γ_2 is in fact either a derivation according to the grammar Γ_1 or a derivation according to the grammar Γ_2. Indeed, no string which contains intermixed nonterminal characters of both Γ_1 and Γ_2 can ever be generated from the case character by a production of Γ. Given this fact, it follows at once that every terminal string generated by the grammar Γ is a terminal string which would have been generated either by a derivation according to the grammar Γ_1 or by a derivation according to the grammar Γ_2. Thus, the language generated by Γ consists precisely of the set theoretical union of the languages generated by the languages Γ_1 and Γ_2, and our result is proved. Q.E.D.

Our next aim is to show that the analog, for a single language, of the pun problem whose algorithmic unsolvability is stated in Theorem 1 is in fact algorithmically solvable. This assertion forms the content of Lemma 3.

Lemma 3. Given any Backus grammar Γ there exists an algorithmic procedure which, given any string w of terminal symbols, will decide whether or not there exists an additional string v of terminal symbols such that wv is a valid string of the language defined by Γ.

Before giving the proof of Lemma 3, we shall derive some of the consequences. Note in the first place that since according to Lemma 3 a problem is solvable for a single language that according to Theorem 1 is unsolvable for the intersection of two languages it follws that the intersection of two languages is not necessarily a language. This result is stated as Theorem 4.

Theorem 4. There exist two languages defined by grammars Γ_1, Γ_2 involving the same set of non-terminal characters whose set-theoretical intersection is not a language definable by a grammar of Backus normal form.

Theorem 4 tells us that the class of languages defined by Backus grammars, which according to Lemma 2 is closed

under the set theoretic operation of union, is not closed. under the set theoretic operation of intersection. But a family of sets closed under union is also closed under intersection if it is closed under complementation. Therefore it follows that our set of languages is not closed under complementation. This result is stated as Theorem 5.

Theorem 5. There exists a language, defined by a grammar Γ of Backus normal form, the set-theoretic complement of which is not a language definable by any grammar of Backus normal form.

Theorem 4 and Theorem 5 are both based on Lemma 3, and it is now time for us to give the proof of Lemma 3.

Proof of Lemma 3: We shall show that, given the grammar and the string w, there exists an integer K, definable entirely in terms of Γ and of the length of w, such that there exists a sentence wv of the language defined by Γ if and only if there exists a second string v' for which wv' is a sentence of the same language whose parse tree consists of not more than K nodes. Suppose for the moment that this has been established. Then note that a) It is easy to enumerate all parse trees with a restricted number of nodes. Indeed, each parse tree consists of a collection of nodes which may be arranged in some serial order, and, associated with each node, either a terminal symbol (possibly null) or, a set of pointers to certain nodes earlier in sequence. The number of pointers associated with each node is bounded in length, the bound being of course the maximum number M of syntactic elements entering into a definer of any definition. Therefore any parse tree may be represented in some such form as

$$(12) \quad \ldots ; N_i, j_1^{(i)}, \ldots, j_{n_i}^{(i)} ; N_{i+1}, j_1^{(i+1)}, \ldots, j_{n+1}^{(i+1)}; \ldots$$

in which N_i is a node designator defining the type of the i-th node and $j^{(i)}$, ..., $j_{n_i}^{(i)}$ are integers subject to the condition. $j_m^{(i)} < i$ and representing pointers. As noted, we always have $n_i \leq M$. It is clear that, given a bound on the number of nodes allowed, the total number of possible patterns (12)

is bounded, and from this assertion (a) is evident.
b) Given the full set of parse-trees enumerated as in (a), we may plainly enumerate all the sentences consisting only of terminal characters and parsed by one of these trees. Then we can tell whether there exists a sentence wv' belonging to this set by serial examination.

This shows that to prove Lemma 3 we have only to prove the existence of the bound K. This we do as follows. Call a parse tree a minimal tree for w if it parses a sentence of the form wv consisting entirely of terminal characters and has no more nodes than any other parse tree with this same property. Call a string N_1, N_2, ..., N_k of nodes in a parse tree a pedigree of each N_j, $j > 1$, is an immediate descendant of N_{j-1}. Call the depth of a parse tree the maximum length of a pedigree in it.

Call the span of a node N in a parse tree the set of all terminal characters belonging to tree twigs which are immediate or indirect descendants of N. We first establish a bound k for the depth of a minimal parse tree for w. To do this, note that if $N_1, \ldots, N_k$ is a pedigree in a minimal parse tree, and if N_1 is a node of the same type as N_k, then the span of N_k must include at least one character of w not included in the span of N_1. Indeed, if this is false, we can obviously reduce the number of nodes of the parse tree by replacing the whole complex of branches depending from the node N_1 by the smaller set of branches descending from N_k, contradicting the assumed minimality of our parse tree. It follows at once that, if ℓ is the number of characters in w, no pedigree in a minimal parse tree contains more than 1 repetitions of a node-type. Hence, if d is the number of distinct node-types allowed by the grammar Γ, $d\ell$ is a bound for the depth of a minimal parse tree for w. Since each node in a parse tree can have at most M descendant nodes, it follows a minimal parse tree for w can contain at most $K=M^{d\ell}$ nodes, establishing the required bound K and completing the proof of our Lemma. Q.E.D.

Various additional interesting unsolvability results may be proved by the methods which we have used to prove Theorems 4 and 5. To obtain a first such result, we modify the grammar (9) so that the definer of the basic syntactic type ⟨otrack⟩ appears as shown in (10) rather than as shown in (9).

(13) ⟨otrack⟩ = S; ⟨etrack⟩ ; ⟨eword⟩ .

Here, we assume that S is some arbitrarily chosen state word for the Turing machine of Theorem 1. Then the collection of words (5) which are grammatical according to both of the grammars (7) and (8), as modified, are all those Turing machine tracks which start with the state S and which lead eventually to the stop state. Since, according to Theorem 1 there exists no algorithm capable of deciding whether there exists a track starting with an arbitrary S and leading to the stop state, it follows that there exists no algorithm which, given two arbitrary languages, is capable of deciding whether or not these languages have a non-zero intersection. This result is stated formally as Theorem 6.

Theorem 6. There exists no mechanizable algorithm which, given two arbitrary grammars Γ_1 and Γ_2 in Backus normal form and sharing the same finite set of terminal characters, will decide whether or not there exists any word v which is grammatical according to both grammars.

A slightly modified form of Theorem 6 gives another interesting result. We call a grammar ambiguous if, according to the grammar, there exist two structurally distinct parsing trees which generate the same string of terminal characters. A grammar which does not admit two such structurally distinct parsing trees is called unambiguous. Now note that the grammar (8) is unambiguous. Indeed, the division of a track into pairs is obviously unique. Moreover, within each pair, the ⟨middle⟩ according to the grammar of (8) is uniquely defined by the fact that the second character of the ⟨middle⟩ is an internal state

character for the Turing machine. Moreover, only one elementary Turing production can apply within a pair, since, given the state of a Turing machine and the character which it is scanning, the character which it will print and the subsequent motion and new state are uniquely determined. Thus in the syntactic analysis of a pair, the <end> is unique also. For the same reasons the grammar shown above as (9) is unambiguous as is the modified form of the grammar (9) shown in (13). It is of course this modified form which has been used to prove Theorem 6.

Suppose now that as in the proof of Lemma 2, we make all the nonterminal characters of the grammars Γ_1 and Γ_2 distinct by renaming the nonterminal characters of the grammar Γ_2, and that we then identify the base characters of Γ_1 and Γ_2, renaming the base character of Γ_2 if necessary. Let Γ be the union of the grammars Γ_1 and Γ_2 as in the proof of Lemma 2. Then, since Γ_1 and Γ_2 have been seen to be unambiguous, there will exist a string of terminal characters admitting two structurally distinct parsing trees if and only if there exists a string of terminal characters which admits both a parsing tree according to the grammar Γ_2. That is, the union grammar Γ will be ambiguous if and only if there exists a word which is grammatical according to both of the subgrammars Γ_1 and Γ_2. But Theorem 6 states that there exists no algorithm which, given two arbitrary grammars Γ_1 and Γ_2 of the form which we are considering, will decide whether or not there exists a word which is grammatical according to both of them. It follows immediately that there exists no algorithmic procedure which, given an arbitrary grammar Γ, will decide whether or not this grammar is ambiguous. We state the result just derived as Theorem 7.

Theorem 7. There exists no mechanizable algorithm which, given an arbitrary grammar written in Backus normal form, will decide whether or not this grammar is ambiguous.

3. Generalized languages and restricted languages.

Generalizations of the Backus normal form grammars which we have studied earlier in the present work have been proposed. In the present section, we will consider one such generalization and study the parsing problem for it. We shall also study a class of strings, the so-called regular strings, which constitute an interesting specialization of the class of languages considered in the two preceding sections.

The generalized languages which we wish to consider are those defined by the so-called context dependent grammars introduced by N. Chomsky. Such a grammar is specified by a finite vocabulary of characters c and by a finite unordered collection P of productions of the following form.

(1) P: AbC $\longrightarrow$ ABC

In (1), A, B and C all denote strings of characters formed from the available character vocabulary. The symbol b denotes some single character chosen from this vocabulary. We allow any of A, B or C to be the null string. Note that productions of the form (1) resemble the productions that we have used to define the context free languages in the preceding Section 2 (cf. display formula (9) of Section 2). However, in contrast to our understanding in the preceding sections, we intend in dealing with context-dependent grammars that the production (1) should be applicable only when the character b occurs in the context specified in (1), that is, with the particular string A to its immediate right and the particular string C to its immediate left. The special case in which A and C both are null is the case of the context independent productions studied earlier. As in the preceding section, we require that there exist precisely one character c among all the characters used in the productions P , is specified as the root or base character of the grammar. As before, we call any character occurring on the

right-hand side of one of the productions (1) but never occurring on the left-hand side of any production a terminal character of the grammar. If a character string $w = c_1 \ldots c_k$ is given, we say that a string obtainable from this string by replacing any of its characters c_j by a set of characters which may validly replace c_j according to one of the context dependent productions (1) is obtained from w by immediate descent. We use the notation for immediate descent introduced in Section 2 and define the notion of grammatical evolution as in Section 2. If there exists a grammatical evolution starting with the string S_1 and terminating with the string S_k we say, just as for context independent languages, that S_k is obtained from S_1 by indirect descent; we use the same notation as that introduced in Section 2 (cf. formula (11)) for this.

Note now that the evolution of an arbitrary Turing machine may be described by a single context dependent language of the type that we have just defined. Indeed, in the basic evolution steps shown in formula (3) of Section 1, all the characters in the state word of the Turing machine except the three central characters c_k, σ and c_{k+1} are seen to play a passive role. The transformation affecting these three central characters is almost precisely of the form (1) allowed for context dependent languages. The elementary Turing transformation differs from (1) only in that it may involve a shift of the character σ to the right or left of a character substitution. To force the Turing machine productions shown in (3) of Section 1 into the form strictly required for the Turing machine to be described formally by a context dependent grammar, we proceed as follows.

For each internal state character σ of the Turing machine, we introduce three additional linguistic characters, related to σ but playing slightly different roles in the grammar which we wish to define. These three characters may be called the "double" of σ, which we write σ_+, the "inverse" of σ, which we write σ^{-1}, and the inverse of the

double, which we write σ_+^{-1} . For each elementary production

(2i) $\quad c\sigma c' \longrightarrow c_1\sigma_1 c'$

of the Turing machine (not involving any rightward or leftward motion of the Turing machine) we introduce the linguistic production shown as (3i):

(3i) $\quad c\sigma \longrightarrow c_1\sigma_1\sigma^{-1}\sigma$.

Note that the linguistic production (3i) specifies a replacement for the character c in contexts wherein σ is the character immediately to the right of c. For each Turing machine production of the form

(2ii) $\quad c\ \sigma\ c' \longrightarrow \sigma_1 c_1 c'$

involving a leftward motion of the Turing machine we introduce a linguistic production of the following type:

(3ii) $\quad c\ \sigma \longrightarrow \sigma_1 c_1 \sigma^{-1}\sigma$

Formula (3ii) is, like (3i), a linguistic production involving a substitution for the character c. For each basic Turing machine production of the form

(2iii) $\quad c\ \sigma c' \longrightarrow c_1 c'\sigma$

involving rightward motion of the Turing automaton along its tape we introduce the linguistic production

(3iii) $\quad c\ \sigma\ c' \longrightarrow \bar{c}\ \sigma_+\sigma c'$,

which is a context dependent linguistic production, once again involving substitution for the character c. In addition to the context dependent grammatical productions shown above as (3i), (3ii) and (3iii), we add a number of additional productions to our grammar; these are intended to relate σ^{-1}, σ_+ and σ_+^{-1} to the character σ. In the first place we introduce context dependent "erasing" productions, involving the characters σ and σ^{-1} , and having the following form:

(4a) $\quad \sigma^{-1}\sigma \longrightarrow \sigma^{-1}$

(4b) $\quad \sigma^{-1}c \longrightarrow c$

Production (4a) allows substitution of a null string for the character σ. As indicated in (4a), this production is applicable whenever σ occurs in a context in which σ^{-1} is the next character on its left. Similarly, (4b) is an "erasing" production applicable to the character σ^{-1}, which is available provided only that the character immediately to the right of σ^{-1} is not the character σ. In addition to (4a) and (4b), we introduce the follwoing four productions

(5a) $$\sigma_+ \sigma \longrightarrow \sigma_+$$

(5b) $$\sigma_+ c \longrightarrow \sigma_+ \sigma_+^{-1} \; c \; \sigma$$

(5c) $$\sigma_+ \sigma_+^{-1} \longrightarrow \sigma_+^{-1}$$

(5d) $$c \; \sigma_+^{-1} \longrightarrow c$$

Production (5a) is an erasing production having much the same form as, and applicable in the same conditions as, the production (4a) described above. Formulae (5c) and (5d) show similar erasing productions. The production (5c) specifies that a character σ_+ may be replaced by a null character provided that the character immediately on its right is the character σ_+^{-1}, and (5d) specifies that σ_+^{-1} may be replaced by a null character provided only that the character immediately on its left is not σ_+. Finally, the production (5b), which involves substitution for the character c, states that c may be replaced by the sequence $\sigma_+^{-1} c \sigma$ if the character immediately preceding c to the left is the character σ_+.

On all of the productions (3i), (3ii) and (3iii) above, we impose the additional contextual restriction that they are only to apply if there exists no character of the form σ^{-1}, σ_+ or σ_+^{-1} within two characters of the character σ. With this restriction, the reader will easily see that the linguistic evolution beginning with a valid Turing machine word of the form (6)

(6) $$c_1 \ldots c_k \; \sigma \; c_{k+1} \ldots c_n$$

is at each step completely deterministic in the sense that at most one production in our grammar can apply, and that aside from "intermediate steps" involving cancellation of symbols σ, σ_+, σ^{-1}, σ_+^{-1} temporarily generated, this grammatical evolution exactly matches the evolution of the Turing machine which we have used to introduce our context dependent grammar.

To relate the Turing machine stopping problem equally directly to the parsing problem for the context dependent grammar introduced above, it is appropriate for us to introduce a simple standardization for Turing machines, as follows. Suppose that M is a universal Turing machine. We have seen the stopping problem for M is unsolvable. We define a slightly modified second Turing machine in terms of M' as follows. Into the vocabulary of print characters for M' we introduce all the print characters of M and, in addition, introduce two extra characters which we call respectively left and a right end marker characters. To any given initial tape for M we append these two end markers, the left end marker to the extreme left of the field indicated by the state word for M, the right end marker to the extreme right of the field indicated by the state word for M. The step by step operation of the machine M' is obtained by simple modification from the operation of M. In the first place, we require (except in the special "clean up" terminal states, defined below) that when M' encounters the left end marker character, it replaces the end marker by a blank, reprints the end-marker one space further to the left, and then returns to the blank square which it has just inserted. Similarly, when the Turing machine M' encounters the right hand end marker character, it replaces it by a blank, reprints a right-end marker one square to the right, and returns to the blank square which it has just inserted. In all other situations (except in the terminal situations to be described below) M' behaves in exactly the same way as M. It follows from the above that, except when manipulating the end-mark characters, M' has a state word which

is always the same as the state word of M, except for the presence of a left end marker character marking the extreme left hand of the modified section of tape and the right hand character marking the extreme right hand of the modified sections of tape.

We also provide the modified machine M' with two internal states, in addition to the states possessed by M. We call these two states a right erasing state and a left erasing state. We agree that any configuration which would bring M into its stop state brings M' into its right erasing state. Once in the right erasing state, M' has the following action. A square is scanned. If the square does not contain the right hand end marker its contents are replaced with a blank and M' moves one square to the right and remains in the right erasing state. In this way M' will successively erase every square to the right of the position at which it entered the right erasing state until it encounters the right end marker. When M' encounters the right end marker, it erases it and enters the left erasing state. In this state it proceeds iteratively to replace every scanned character with a blank and move one square to the left, except that when it encounters the left hand end marker it replaces it with a blank and stops. Thus, if starting with a given initial configuration, M would stop, then M' will also stop if started in the corresponding initial condition. But, before stopping, M' will have erased its entire tape. If c_0 denotes the blank character in the print vocabulary of M and M' and σ denotes the stopping state for M and M', then the final state word for M' in the configuration in which it is stopped is

(8) $\qquad c_0\sigma$.

If for any initial configuration (7) of M' we complete the definition of our context-dependent language by introducing a base character b which occurs in the single production

(9) $\qquad b \longrightarrow c_L c_1 \cdots c_k \; \sigma \, c_{k+1} \cdots c_n c_R,$

we obtain a language with the following property: the Turing machine M', started in the configuration (7), will stop if and only if the state word (8) is grammatical in our language. That is, the Turing machine M', started in the state shown on the right of (9), will stop if and only if the state word (8) can be derived according to the productions of the context dependent grammar by indirect descent from the character b. It follows that the parsing problem for context dependent languages includes the stopping problem for the universal Turing machine. Hence, the former problem is algorithmically unsolvable. We state this result as Theorem 7.

Theorem 7. There exists no programmable algorithm which, given an arbitrary context dependent grammar Γ consisting of finitely many productions of the form (1), and given a word w consisting of terminal characters of this grammar, will decide whether w belongs to the context-dependent language defined by Γ.

This result indicates that unrestricted context dependent grammars are too general to be directly useful for the specification of programming languages.

We now turn to consider the following question: to what extent can general context-free languages be parsed by algorithms restricted in various ways? In particular, can every context-free language be parsed by an advancing deterministic scheme of the sort considered in Chapter IV? We begin our discussion of this question by giving a very simple example of a context-free language -- the set of all symmetric strings on two letters, i.e., the set of all strings on two letters which return their form under left-right reversal. We shall now show that this set of strings admits no determining context-free grammar which can be parsed by a deterministic advancing scheme, essentially because no parsing scheme can find the middle of the string without scanning the string to its end and then backing up.

We proceed as follows. First note that the set of symmetric strings on two letters a,b is a context-free language with the simple grammar

(1) ⟨word⟩ = aa | bb | a | b | a ⟨word⟩ a | b ⟨word⟩ b.

To show that this simple language cannot be parsed by any deterministic advancing scheme, we must first give an abstract general model of such parsing mechanisms. We model advancing deterministic parses in terms of an abstractly defined class of pushdown automats, as follows. A deterministic pushdown automaton, or simply pushdown automation, is defined by specifying a finite alphabet of input characters $\{a\}$, a finite alphabet of pushdown stack characters $\{\alpha\}$, and a finite collection $\{\sigma\}$ of internal states, together with various functions of those characters described in detail below. The alphabet of input characters must include a specially defined end-string character $\dashv$. The alphabet of pushdown characters must include a special stack bottom character α_0. The set of internal states of the automaton must include a distinguished initial state σ_0, and a distinguished accepting state σ_+. The automaton acts in stepwise mode. At each moment, its full state is defined by the state of its pushdown stack, the state of its input string, and by its internal state σ. The pushdown stack state is defined by a word $\alpha_0\alpha_\beta\gamma\ldots\delta$ formed out of pushdown stack characters and containing exactly one (initial) occurrence of the character α_0. The input string state is defined by a remaining input word $a_1\ldots a_n\dashv$ formed out of input string characters and containing exactly one (terminal) occurrence of the special character $\dashv$. Each move of the automaton may involve both a state change and an elementary transformation both of the pushdown stack and of the input string; each such move is fully determined by the internal state of the automaton, by the topmost (i.e., rightmost) character on the pushdown stack, and by the next (i.e., leftmost) character of the input string. An elementary move may either add an extra character to the top of the pushdown stack, remove a single character from the top of the pushdown

stack, or leave the pushdown stack unchanged; such a move may also either delete the leftmost character from the input string, or leave the input string unchanged. Thus, the action of the pushdown automaton is fully defined by the following functions of δ (top character on pushdown stack), a (next character in input string), and σ (internal state).

(2) $S(\delta,a,\sigma)$, value in $\{\sigma\}$ = next internal state function;
$P(\delta,a,\sigma)$, values -1,0,+1 = pushdown stack move function;
$D(\delta,a,\sigma)$, defined only if $P(\delta,a,\sigma) = +1$, value in $\{\alpha\}$ = pushdown character placed on stack;
$I(\delta,a,\sigma) = 0,-1$ = input move function.

We restrict these functions by the following conditions:

(3) $P(\alpha_0,a,\sigma) \geq 0$ for all a and σ
(stack may not be pushed past its bottom);
$I(\alpha,\dashv,\sigma) = 0$ for all α and σ
(input scan may not pass end-string mark);
$S(\alpha,a,\sigma_+) = \sigma_+$
(acceptance state is invariant).

A string $a_1 \ldots a_n$ of input characters not containing the end-string character $\dashv$ is said to be <u>accepted</u> by the pushdown automaton if, when started in the internal state σ_0, the pushdown stack being in the "empty" state α_0, and the input string being in the state $a_1 \ldots a_n \dashv$, the pushdown automaton will eventually reach the invarient internal acceptance state σ_0. If A designates a pushdown automaton we write the set of all input words which the automaton accepts as OK(A). Note therefore that A may be regarded as a mechanism for performing a "syntax check" on certain classes of strings. It is clear that a language not permitting such a syntax check by a pushdown automaton cannot, in any reasonable sense, be parsed by any such automaton.

The theorem to which we shall devote all our efforts in the present section is as follows:

Theorem 8: Let Σ be the set of all symmetric strings on the two letters a and b. Then there exists no deterministic pushdown automaton A such that

$$\Sigma = OK(A).$$

We shall build a proof by contradiction out of a few lemmas. Thus suppose that our theorem is false, and let A (as described above) be a pushdown automaton such that $\Sigma = OK(A)$.

We first consider input strings of the special form $b^n a^m w$, w being an arbitrary string on the letters a and b (exponents indicating repetition) and examine the reaction of the automaton (started in its normal initial state) to such a string. Note first of all, that the automaton will, after a finite number of steps, delete every one of the m+n characters $b^n a^m$. Indeed, were this false, the reaction of the automaton to an input string $b^n a^m w$ would be independent of w. But since $b^n a^m a^m b^n$ is accepted by the automaton, while $b^n a^m a$ is not, this is impossible.

We now make the following auxiliary definition.

Definition 9: N(n,m) is the least number of symbols on the pushdown stack of the automaton A at the beginning of any move in the series of moves of the automaton during which the n+m-1-th of the characters $b^n a^m$ but not the n+m-th of these characters has been removed from the input string.

The following lemma shows that in scanning the class of input strings described above, the automaton A must develop an increasingly large stack.

Lemma 10: We have $\lim_{m \to \infty} N(n,m) = \infty$ for all n.

Proof: Suppose the contrary. Then there must exist some n_o and some sequence $m_1, m_2, \ldots$ tending to ∞ such that $N = N(n_o, m_1) = N(n_o, m_2) = \ldots$. The sequence $m_1, m_2, \ldots$ is infinite, while the total number of possible internal states of A and the total number of possible conditions of a pushdown stack word of length N are both finite. It therefore follows that there exist two distinct integers $m < \bar{m}$

such that both the internal and the stack state of the automaton A take on identical values at the beginning of two distinct moves of the automaton, namely, at the beginning of some move of A during which the n_o+m-1'th but not the n_o+m'th of the characters $b^{n_o}a^m$ has been removed from the input string, and at the beginning of some move of A during which the $n_o+\bar{m}-1$-th but not the $n_o+\bar{m}$'th of the characters $b^{n_o}a^{\bar{m}}$ has been removed from the input string. But then the automaton will accept a string $b^{n_o}a^m w$ if and only if it accepts the string $b^{n_o}a^{\bar{m}}w$. Putting $w = b^2a^m b^{n_o}$ we see that this is impossible, a contradiction proving our Lemma. Q.E.D.

We now make a second auxiliary definition.

Definition 11A: $N_1(n,m)$ is the number of the first move of the automaton A, as it scans an input string of the form $b^n a^m w$, with the following two properties

(a) at the beginning of the move in question, all the n characters b^n have already been removed from the input string;

(b) at the beginning of the move in question, the number of characters on the pushdown stack of A has attained a value which will increase and not recur again until a character equal either to b or to $\dashv$ is subsequently encountered in the input string.

Note that it follows at once from Lemma 1 that $N_1(n,m)$ is well defined for each n and all sufficiently large m, and that, if m is sufficiently large, $N_1(n,m)$ is independent of m. This allows us to make two more auxiliary definitions.

Definition 11B: For each $N \geq 1$, let $M(n)$ be an integer sufficiently large so that $N_1(n,m)$ is well defined for all $m \geq M(n)$, and so that $N_1(n,m) = N_1(n,\bar{m})$ for all $m,\bar{m} \geq M(n)$.

Definition 11C: For each n let $\sigma(n)$ be the internal state of the automaton A at the beginning of its $N_1(n,M(n))$ -th move, and let $\alpha(n)$ be the top character of the pushdown stack at the beginning of this same move.

We now note that since the total number of internal states of the automaton A and the total alphabet of pushdown state characters are both finite, there exists some infinite increasing sequence $n_1, n_2, \ldots$ of integers such that $\sigma(n_1) = \sigma(n_2 = \ldots$ and $\alpha(n_1) = \alpha(n_2) = \ldots$.

Next we observe that the following lemma may be established by an argument similar to that used to prove Lemma 1; we leave it to the reader to make the necessary adaptation of that argument.

Lemma 13: Let the automaton A be given an input string of the form $b^n a^m b^k w$ to scan, and consider the sequence of moves which it will make. Let $N_2(n,m,k)$ be the least number of symbols on the pushdown stack of A during any move during which the first of the latter group of k b's has been removed from the input string, but during which the first character of w has not yet been scanned. Then

$\lim_{m \to \infty} N_2(n_1 m_1 k) =$ for all fixed n and k.

If we combine Lemma 12 with the facts stated in the paragraph preceeding its statement, we can deduce certain interesting facts.

Let n_j be as in Lemma 12, and consider the action of A in scanning input strings of the form $b^{n_j} a^{m_j} b^k \dashv$. We assume here that the integer m_j, which will be defined more precisely below, is at any rate sufficiently large so that

i) m_j exceeds $M(n_j)$ and therefore exceeds the number i_j of characters a of the input string scanned by A at the beginning of its $N_1(n_j, M(n_j))$=th move;

ii) $N_2(n_j, m_j, k)$ exceeds $N(n_j, M(n_j))$.

It follows from i) and ii) that the moves of A subsequent to its $N_1(n_j,M(n_j))$-th move, and up to the first move at which the terminator $\dashv$ is scanned, depend only on the portion $a^{m_j-i_j}b^k\dashv$ of the input string remaining after this move. Hence, if we put $m_j = i_j+m$, where m is any sufficiently large integer, it follows that the moves of A subsequent to its $N_1(n_j,M(n_j))$-th and up to the first move at which the terminator $\dashv$ is scanned, are independent of j. It is also plain that A will not examine any one of the first $N(n_j,M(n_j))$ symbols on its pushdown stack during any of these moves. The symbols on the pushdown stack of A during any of these moves will therefore consist of a first group depending only on j and a second group depending only on the number of the terminating string: $a^mb^k\dashv$ of input signals which have been scanned. The following lemma summarizes these facts.

Lemma 13: Let K be any fixed large integer. Let $\{n_j\}$ be as in the paragraph immediately preceeding the statement of Lemma 1. There exists a sequence of integers m_j such that for all $1 \leq k \leq K$, and letting $\Sigma(n,m,k)$ denote the condition of the pushdown stack of the automaton A at the beginning of the first move during its scan of the input string $b^na^mb^k\dashv$ in which the string terminator symbol $\dashv$ is scanned, then

i) $\Sigma(n_j,m_j,k)$ may be written as a concatenation

$$\Sigma(n_j,m_j,k) = \Sigma_1(j)\ \ \Sigma_2(k),$$

where the string $\Sigma_1(j)$ of pushdown stack characters depends only on j, and the string $\Sigma_2(k)$ depends only on m and k.

ii) The internal state of A at the beginning of this same move depends only on k.

Using Lemma 13, it is quite easy to complete the proof of Theorem 8. Let J be one more than the number of internal states of the pushdown automaton A, and choose $K > n_J$ in

Lemma 4. Present each of the strings $b^{n_j}a^{m_j}b^{n_i}\dashv$ where $i,j=1\ldots,J$ to the automaton, and consider the resulting sequence of moves. In particular, consider the moves made by A after that first move, described in Lemma 13, in which it scans the string terminator symbol $\dashv$. Plainly there exists no $i \leq J$ for which A passes into its accepting state without removing all the characters of $\Sigma_2(n_i)$ from the pushdown stack. Indeed, if such an i existed, the response of A to the inputs $b^{n_j}a^{m_j}b^{n_i}\dashv$ would be independent of j, which is clearly impossible. Consider then the internal state σ_j of A at the beginning of the first of its moves which follows the removal of all the characters of $\Sigma_2(n_1)$ from the pushdown stack. Since J exceeds the number of internal states of A, there must exist two distinct integers i and i', neither exceeding J, such that $\sigma_{i'} = \sigma_i$. But then A reacts to each of the strings $b^{n_j}a^{m_j}b^{n_i}\dashv$ in exactly the same way as it reacts to $b^{n_j}a^{m_j}b^{n_{i'}}\dashv$. In particular, A accepts $b^{n_i}a^{m_i}b^{n_{i'}}\dashv$, a contradiction which completes the proof of Theorem 8.

CHAPTER 6. OPTIMIZATION METHODS FOR ALGEBRAIC LANGUAGES

1. Introduction.

We noted in Chapter I that, whereas a source language program consists ideally of a minimal set of indications in appropriate form defining an algorithm, the 'machine' or 'assembly language' program into which this program is translated either by a compiler or by hand may contain a great deal of additional information. In particular, we observed that such an assembly language program specifies particular machine registers and register types to be used in performing the individual machine operations to which a source language is reduced; specifies the particular order in which the necessary sequence of operations is to be stored within the machine; the particular order in which the sequence of operations is to be executed by the hardware; the detailed pattern in which intermediate information generated in the course of computation is to be stored and reloaded, etc. The substantial purpose of all these additional specifications is the transformation of the original source language algorithm into a form which is more or less 'optimized'; that is, into an equivalent form which will run as fast as possible and make minimal necessary demands on the various forms of storage available within a given computer. Now, the general problem of determining the equivalence or inequivalence of a pair of programs is certainly unsolvable. For this reason, it is impossible to develop a fully systematic and wholly adequate theory of program optimization. Any discussion of optimization will necessarily be a semi-systematic account of a variety of methods, each of partial applicability, each intended to optimize one more or less limited aspect of a program.

In spite of the difficulties which must be faced in its attainment, a reasonable level of optimization is essential if source language programs are regularly to replace programs

written in assembly language. To the extent that compiler-produced code is substantially slower and more voluminous than code produced by hand, it is economically inevitable that programs of major significance will be hand-coded. Fortunately, however, many of the hand coding techniques most significantly enhancing the quality of code are of a sufficiently simple logical structure as to be algorithmically formalizable. The resulting collection of algorithms can recapture many of the experienced programmer's "coding tricks." A really good optimization program can in many cases produce code comparing very favorably with hand code. In some cases, by making use of the unique ability of a computer to examine a great many items rapidly and accurately, an optimization program can even outdo the hand coder. A human programmer is, after all, limited by time in the number of optimizations which he can consider, and by accuracy and debugging considerations in the number of optimizations which, having considered, he will decide to use. It is to be anticipated that, as our knowledge of optimization algorithms grows more organized and sophisticated, and as growing computer speed and storage capacity permit the use of increasingly powerful optimization procedures, we will be able to produce mechanically optimized code whose quality substantially exceeds that of hand code in all but isolated special cases.

Optimization procedures may be divided into two recognisably distinct classes: machine independent optimizations and machine dependent optimizations. Machine independent optimizations make use of certain rather general transformations of algorithms which, for a large class of languages and machines, are apt to improve efficiency. Thus, for example, the elimination of instructions, redundant because the calculations they accomplish have already been performed, will in almost all cases and for almost all machines accomplish a saving both of computation time and of program size.

Such an optimization is therefore machine independent. Machine dependent optimization methods, on the other hand, depend rather strongly on particular features of a given machine. Thus, for example, if one particular register of a machine is, in virtue of the machine's structure, the locus of a particularly rich set of single-bit Boolean operations, then a program involving any considerable amount of Boolean manipulation will be most efficient if we can arrange to have most of the Boolean operations performed in the designated register. Moreover, in this situation, any preliminary operation preparing results which are needed as Boolean operands should, for the sake of efficiency, generate its results directly in this same register. Since in the present monograph we are not concerned with any particular machine, we intend to discuss the machine independent optimizations much more fully than optimizations depending on particular machine structures. On the other hand, since any line separating these two classes of optimizations is artificial at least in part, and since a number of important optimizations are in fact machine dependent, we cannot avoid all discussion of machine dependent optimization.

We begin with quick survey of some of the principal machine independent optimizations.

1. Redundant instructions may profitably be eliminated. Thus, for example, the FORTRAN statement

(1) $\quad \vdots \quad \vdots$
$\quad A(I,J) = B(I,J)$,

in which A and B are understood to be two-dimensional arrays (which for the purpose of the following illustration we assume to have the dimension (25,25)) will be expanded by a completely unoptimized translator into code of the form

(2)
```
...
K = I * 25
L = K + J
M = Address(B) + L
Load from (M)
```

```
N = I * 25
O = N + J
P = Address(A) + O
Store into (P)
...
```

It is apparent on inspection of the code (2) that the quantities M and O have exactly the same values as the quantities K and L, so that, if the value of L is saved and hence available in the latter part of the code sequence, statement(1) is expanded may be translated more efficiently as

(3)
```
...
K = I * 25
L = K + J
M = Address(B) + L
Load from (M)
P = Address(A) + L
Store into (P)
...
```

Note that our assertion that the code (3) is more efficient than the code (2) depends only on very mild assumptions concerning the computer on which the codes (2) and (3) are to run. We assume, in particular, that enough registers or core locations are available for the quantity L to be saved and, if necessary, reloaded without requiring a reloading process so elaborate as to be more expensive than the multiplication N = I * 25 and the addition O = N + J combined. Making this mild assumption, we regard the process that takes us from the "raw code" (2) to the "optimized code" (3) as a machine independent optimization.

Note that for almost all machines the code (3) will be better than the code (2) in two valuable senses. In the first place, (3) calls for the execution of fewer machine operations than does (2), so that (3) will run faster. Moreover, the code (3) is shorter than the code (2) so that less internal storage is required to store the sequence

(3) than is needed for (2). Many, but not all, of the machine independent optimizations which we shall consider will yield both these advantages. However, we shall also discuss optimizations (like the optimizations by "reduction in strength" described below) which make codes more efficient in regard to their execution time, but at the expense of a slight expansion in their length. Notice also that the optimization described above is an axiomatic aspect of "good handcoding procedure," and would be made, as a matter of course, by any experienced programmer expressing the algorithm (1) directly in assembly code.

2. A second class of optimizations aims at the optimization-time detection of expressions with determinate constant values, and the replacement of such expressions in the optimized code sequence by their known constant values. For example, it is not uncommon for a programmer to define a quantity PI at the beginning of its program by the FORTRAN statement PI = 3.14159, and then to use this quantity symbolically in subsequent portions of his program, where he might, for example, write

```
(4)     ...
        X = 2 * PI * Y
        ...   etc.
```

On the reasonable assumption that the quantity PI is never redefined, the expression (4) may be rather trivially rewritten as

```
(5)     ...
        X = 6.28318 * Y
        ...  ,
```

which in the expanded, assembly language version of the same code would both be shorter and save a multiplication. The saving of time which results from this replacement can of course be particularly important if the instructions (4) or (5) happen to lie in a frequently executed region of the program.

Evidently, other constants than π can be involved in such optimizations. A programmer striving for generality of code will, in many cases, refer to various constants symbolically throughout his program, intending to supply particular constant values for all necessary symbols in a block of statements forming a prefixed initialization section in his program. Any symbol used in this way may be the subject of an optimization of the kind that takes us from (4) above to (5). Optimizations of this sort have been called optimizations by constant propagation; this is the term by which we shall describe such optimizations in the remainder of the present chapter.

Optimization by constant propagation may be particularly important in connection with subroutines. Often, a symbolic argument to a subroutine will in fact be assigned only one single value every time the subroutine is called from a given location. The use of techniques allowing the joint optimization of separately pre-compiled subroutines enables one to take advantage of such circumstances to produce efficient subroutine code with which folding has been performed during a pre-run optimization pass.

3. Another useful machine independent optimization procedure is the elimination, within a program, of all calculations whose results are never used, and of all calculations which do nothing but prepare data required for such calculations. Of course the occurrence in an unoptimized program of such a calculation is normally a result of programmer error or carelessness. However, various of the optimization procedures discussed below may, as a side effect, introduce unused calculations of this sort into a program. For this reason, it is important for an optimizer to be able to remove such calculations. This situation will be illustrated below.

The operations required for the elimination of unused computations are also of use in performing another important optimization, namely, the elimination in generated machine code of stores of quantities subsequently unused. Often, for example, a quantity playing an important role within a given code loop is not used or needed once this particular section of code is left. A typical example of such a quantity is an integer used as a counter within FORTRAN DØ-loop. If the optimizer is able to detect the points within a code at which various quantities become useless, it can avoid storing such quantities in core locations on leaving the regions in which these quantities are required.

A symbolic quantity which when a program reaches a certain point is no longer needed in its execution is said to be dead at this point in the program. One of the most essential programming tricks used by the experienced assembly language programmer is the systematic detection of program points at which variables become dead, and the use of this information to eliminate unnecessary stores and for the economical assignment of the fast registers available within a given computer. As we shall see, the systematic process of flow tracing which an optimizer can employ in order to detect dead variables and eliminate stores of them is very close in its logical form to the process which must be employed to eliminate redundant calculations.

4. We noted above that the various subparts of a program will in general be executed with different frequencies. In many programs, this skewing of the distribution of execution time over the whole set of instructions constituting the programs is quite extreme. Certain, often very small, sections of the program may be executed with extreme frequency. Such sections of program, which commonly have the logical structure of a simple or only slightly branched loop, are commonly called the 'inner loops' of the program. It is often the case that a program spends the overwhelming bulk

of its execution time in these small inner loops, and that all the rest of the instructions making up the program, which may constitute a very much larger mass of code, are executed relatively infrequently. In such cases, the speed with which a computer will complete the execution of a given algorithm may be increased considerably if instructions can be moved out of frequently traversed program sections into less frequently traversed sections. Optimization which makes use of this fact is called optimization by code motion. Thus, for example, the FORTRAN DO-loop

```
(6)       ...
          X = 0.0
          DØ 1 I=1,100
        1 X = X + 2.0 * Q * FLØAT(I)
          ...
```

will run with distinctly enhanced efficiency if the multiplication of the constant Q by 2.0 is moved out of the loop, and the short code sequence (6) recast as

```
(7)       ...
          X = 0.
          QQ = 2.0 * Q
          DØ 1   I=1,100
        1 X = X + QQ * FLOAT(I)
```

Detection of cases in which an improvement of this sort is available is another important goal for a well constructed optimizer.

The optimizing transition from (6) to (7), that is, transport of an operation (or more generally of a sequence of operations) out of a program loop is only possible if none of the arguments involved in any of the instructions to be moved is redefined within the loop. This and other more general constraints on code motion will play an important part in our discussion. These constraints on code motion and related optimizations are most naturally expressed in terms of the flow graph of a program. For this reason a

systematic discussion of program flow graphs will preceed our study of optimization by code motion. We will be led through an analysis of program flow to the definition of certain important nested families of subregions of the program having the character of general abstractly defined "nests of loops."

5. There exists a generalization of the simple code motion optimization discussed above with a significantly increased range of applicability. This generalisation may be comprehended most readily by considering simple example. Suppose that the code

```
(8)      ...
         DØ 1  I=1,100
      1  A(I,J) = B(I,J)
```

occurs in a source program. Compilation of this code, followed by optimization by the elimination of redundant instructions as described above, would yield the following pattern of instructions.

```
(9)      ...
         I = 1
LØØP:    K = I * 25
         L = K + J
         M = Address(b) + L
         Load from (M)
         P = Address(A) + L
         Store into (P)
         I = I + 1
         IF(I .LE.100)  GØ TØ LØØP
         ...
```

Since the quantity I is redefined within the loop contained in the code (9), the multiplication K = I * 25 cannot be moved out of the loop. However, it is possible for us to replace this (relatively expensive) multiplication operation by a (less expensive) addition operation. We have only to make use of the fact that a multiplication is an iterated addition. This observation allows us to transform the

code (9) into the form shown immediately below.

```
(10)      ...
          I = 1
          K = 25
LØØP:     L = K + J
          M = Address(B) + L
          Load from (M)
          P = Address(A) + L
          Store into (P)
          I = I + 1
          K = K + 25
          IF(I.LE.100)GØ TØ LØØP .
          ...
```

Passage from (9) to (10) involves

(a) motion out of the loop of the instruction K = I * 25, followed by an optimization by constant propagation "folding" the known value I = 1 into the instruction K = I * 25, resulting in the simpler instruction K = 25.

(b) The insertion, at the point within the given loop at which I is incremented, of the "differential" form K = K + 25 of the original instruction K = I * 25.

An optimization of this sort, replacing, within a loop, the relatively expensive operation (e.g. multiplication) by a relatively inexpensive operation (e.g. addition) is called optimization by reduction in operator strength, or, more simply, reduction in strength. In a subsequent section of the present chapter, we shall discuss this method of optimization more systematically.

6. Elimination of unnecessary computations may often be combined with corresponding modification of the loop-closing conditional transfers normally generated from source statements having the general character of the FØRTRAN statement; doing this, we obtain an additional optimization. If, for example, we consider the code sequence (10), we notice that the loop closing statement 'If(I.le.100) go to LOOP' may be

transformed into a test of the quantity K, which quantity is calculated within the loop and available at the point in the code at which the IF statement is to be executed. Specifically, we may replace the termination test for the DO-loop of (10) by the quite equivalent test "IF(K.LT.2500) GO TO LOOP." Now, once this change has been made, the operation I = I + 1, which merely counts iterations, may become superfluous. This will certainly be the case if the quantity I is dead at the point at which exit from the indicated loop is made. In this case, by combining test replacement with instruction removal, we can rewrite the code sequence (10) in the more efficient form

(11)
```
            ...
            K = 25
LOOP:       L = K + J
            M = Address(B) + L
            Load from (M)
            P = Address(A) + L
            Store into (P)
            K = K + 25
            If (K.LE.2500) GØ TØ LØØP
            ...
```

The procedure we have just described is called <u>optimization by test replacement</u>. Note that, in the optimized code (11), no reference to the symbolic quantity I occurs; in particular, no register is required, even temporarily, for the storage of the quantity I. Generally speaking, removal of references to dead variables not only saves code but economizes the use of available fast registers.

Later in the present chapter we shall discuss miscellaneous optimization procedures related to the test replacement optimization that we have just described, useful in connection with iterative loops of the kind common in many algebraic languages.

7. On many high speed computers, tests, i.e., conditional transfers, are expensive operations to execute, in that, when a conditional transfer occurs, the even flow of instructions through the hardware is momentarily interrupted, forcing the hardware to use time to recover from this transient interruption. Later in the present chapter we shall discuss a class of machine "semi-independent" optimizations of marginal utility which are aimed at just this point.

8. There exists an interesting class of loop optimizations which increases the efficiency of a program, reducing the number of instructions which must be executed at the expense of expanding the number of instructions of which the program actually consists. The following example will illustrate the procedure, generically called optimization by unrolling, which we have in mind. Consider the elementary iterative loop

(12)
```
      ...
      DØ 1 J=1,100
1     A(J) = B(J)
      ...
```

This loop, expanded into a form closer to assembly language, would have the following form.

(13)
```
      ...
      J = 1
LOOP  K = Address(B) + J
      Load from (K)
      L = Address(A) + J
      Store into (L)
      J = J + 1
      If (J.LE.100) GØ TØ LØØP
      ...
```

A more efficient, albeit longer, code sequence equivalent to (13) appears below.

(14)

```
            ...
            J = 1
LOOP:       K = Address(B) + J
            KK = Address(B+1) + J
            Load from(K)
            L = Address(A) + J
            LL = Address(A+1) + J
            Store into (L)
            Load from (KK)
            Store into (LL)
            J = J + 2
            If (J.LE.100)GØ TØ LØØP
            ...
```

Both code sequences (13) and (14) are equivalent to the source language code (12). However, comparing (13) with (14), we see that, in the first place, the elementary loop in (13) is traversed 100 times, whereas the modified or "unrolled" loop in (14) is traversed only 50 times. The total number of address calculations, loads, and stores performed is equal in both cases. However, the bookkeeping operations (incrementation of the loop counter J, execution of the loop termination test) are performed 100 times in the code shown in (13) but only 50 times in the code shown in (14). Particularly if the test operation is an expensive one, this may yield a significant saving in execution time. In a subsequent section, we shall observe that application of the unrolling optimization to loops performing various common two dimensional matrix operations attains still further gains.

9. As we remarked above, we intend to describe optimizations of the machine independent class rather extensively, but will discuss machine dependent optimizations only in rather cursory fashion. Machine dependent optimizations must, in fact, often be adapted rather closely to the particular features of an individual computer. Certain optimizations

of this class may however exhibit a fair degree of trans-computer commonality, expecially in regard to computers designed to perform scientific numerical calculations of the most common sort efficiently. Such machines will typically be provided with a fair number of high speed "scratch" registers. Operations finding their arguments in such registers rather than in normal "memory" or "core location," will normally proceed faster than operations which must reference core. Various of the more important machines of this class have an order code structure in which many operations find both their operands in high-speed register and generate their results in a register; an important consideration for computers of this class is the optimization of register allocation. Symbolic quantities occurring in semi-expanded assembly code must be assigned to registers at any point in the actual machine code in which operations are to be performed on them. Once the available supply of fast registers has been completely used, and when a register is required for a new quantity, one or more of the available registers must be unloaded in order to free it for reassignment. This forces the interpolation into final machine code of auxilliary load and store operations. A good register allocation scheme, by carefully deciding on the pattern in which symbolic quantities will be assigned to registers, can reduce the number of such interpolated load and store operations very considerably. This optimization requires a systematic survey of the semi-final code, aiming at the estimation for each symbolic quantity of the imminence of its next use. Using these estimates, a good resource scheduling algorithm can evolve an effective pattern of register assignments.

10. High speed machines often attain their speed by the systematic application of 'microparallelism', that is, by the provision of independent or multiple arithmetic, Boolean, indexing, and other functional units, and by the execution of local sequences of instructions in parallel where this

is logically possible. For machines of this class, instruction scheduling can be a second important machine dependent optimization. If a program is to run with maximal speed it is important that the operations in it be ordered in such a way as to provide the greatest possible scope for parallel execution. Thus, for example, in evaluating the expression A * B + C * D on a machine either possessing two multiply units or able to execute two multiplies in parallel, it is good practice to begin the two multiplication operations A * B and C * D as soon as possible, and to schedule the addition after the multiplications have been started. Various extensions of this idea are possible. However, since this class of optimizations depends in a fairly sensitive way on the manner in which the computer hardware handles overlapped instructions we shall not discuss it in any detail.

11. A final interesting class of machine dependent optimizations involves the possibility of combining several instructions into a single machine instruction. Thus, for example, on one computer one may have to make a size test on the absolute value of a quantity by first forming the absolute value and then performing a separate test; a machine with a different order code structure might permit the combination of these two instructions into a single machine instruction. Code improvements of this sort depend quite sensitively on the order code structure of a given computer and hence tend to be highly machine dependent. We shall therefore have very little to say about them.

2. Basic Code Blocks and Their Optimization.

Optimization, which is a complex process of code transformation, finds the source of its greatest difficulties in the presence, within programs, of transfers. In this section, we shall treat a preliminary case of the optimization problem, assuming away this fundamental difficulty.

That is, we shall discuss the optimization of sections of code containing neither labelled points which may be the targets of control-transfers, nor transfer operations sending control out of the given section of code. Such a sequence of instructions, having the property that control always enters the sequence at its first instruction and always leaves the sequence at its last instruction, is called a basic block. A general program may be divided into a collection of basic blocks; we have only to introduce a point of division at each labelled instruction within the program and at each conditional transfer operation which it contains. In subsequent sections of the present chapter we shall go on to develop methods for the representation of the flow structure of a program containing conditional transfers and associated labelled destination points and to study the present consequences, for optimization, of program flow.

Optimization by the elimination of redundant instructions and by the propogation of constant values may usefully be performed even in the limited context of a single basic block. In this same context we may also optimize by the elimination of unnecessary variable names. E.g., the code sequence

(1)
```
D = X * Y
C = D
B = C
A = B
```

may be simplified, by elimination of the unnecessary intermediate variables B,C and D, and reduced to the form

(2)
```
A = X * Y .
```

On the reasonable assumption that information is available which separates the variables whose values are calculated in the given block into 'live' and 'dead' variables, we may further optimize a basic block by the elimination within it of all dead variables and all calculations which play no other role than to prepare values for such variables.

On the other hand, none of the optimisations involving the motion of code, as described in Section 1, make sense in the limited context of a single basic block. Thus, in the present section, we shall not discuss either optimisation by code motion, optimisation by strength reduction, or optimisation by test replacement. Our attention will be confined to optimisation by constant propagation and by the elimination of redundant computations. The relatively simple and straightforward algorithms described in the present section will, in the section following the present section, be sophisticated by the inclusion of methods for the treatment of code flow. The more powerful optimisation scheme thereby obtained will be called the linear nested region optimisation scheme. Thus, the present section may be regarded as preparatory to the section which follows it.

The first aim which a basic block optimiser must accomplish is the detection of formal identities. This end is attained as follows. When, in processing a block of code, the optimiser encounters an elementary computation not previously occurring, a serial or value number is assigned to this computation; this value number may be thought of as an identifier attached to the result quantity produced by the calculation. At the same time, an entry defining the operation to be performed, the value numbers of its two inputs, and the value number of its result is placed in a hash-addressed available computations stack. The form of an entry in this stack is as follows.

(opn)	(val1)	(val2)	(valnum)	constant bit (isconst)
op	first input value no.	second input value no.	result value no.	c

Fig 1. Form of item in available computations stack
(Parenthesised keywords are names of corresponding fields in Table I.)

Each operation subsequently encountered is checked against the available computations stack to determine whether or not an identical operation has previously been performed; if it has, the operation in question is redundant and may be replaced by a reference to a previously calculated value. On encountering an assignment operation whose target is a given variable, we associate the value number of the expression forming the right hand side of the assignment with the variable which is the target of the assignment. To appreciate the combined effect of these procedures, consider the sequence of statements

(2)
```
X = B * C * D + B * C * 2.0
B = B * C * 3.0
Y = B * C + 1.0
```

The multiplication operation B * C occurs four times in this code sequence. Three of these multiplications yield the same value, so that two of them are redundant and may be eliminated. The formally identical multiplication occurring in the last line of (2) is however not redundant, since the value of B is changed by the store operation which forms part of the second line of (2). Clearly, every time the value of a variable is reset by a store operation affecting it, subsequent computations involving this variable must be redone. Our optimisation algorithm takes account of this circumstance by attaching to the target value of an assignment statement, the variable number belonging to the right hand side of the assignment statement. Thus, for example, in optimising the code corresponding to the sequence (2), the variable B will receive a new value number during the optimisation of the second displayed statement. The final occurence of the multiplication B * C will then involve a pair of input value numbers distinct from the value numbers previously associated with the same multiplication.

The optimiser will thus be able to recognize that the final occurence of B * C in the code sequence (2) is irredundant. A rather similar procedure may be employed in connection with indexed variables and assignment operations having indexed variables as targets; additional details are given below. Note

that a field must be reserved in the symbol table entry describing a given variable to contain the value number currently associate with the variable. When, in processing a basic block, the optimiser encounters a variable for the first time, it will assign a previously unused value number to the variable. A value number counter, incremented each time a new value number is assigned, may be used for this purpose. The same counter is used to assign value numbers to previously unencountered expressions.

The procedures employed in connection with indexed loads and stores necessarily differ somewhat from those which apply to the corresponding unindexed operations. The occurrence of an indexed assignment

(3) $$a(I) = \text{expression}$$

makes the compile-time value of every subsequent reference to an entry a(j) taken from the array a indefinite, except that expression a(i) itself is known to have the value assigned to it in (3). For this reason, on encountering an indexed assignment (3), we give the variable (i.e., array) name a new value number, and construct a new available value item representing the indexed target expression a(i), setting the operation field of this new item to signal an indexed fetch operation, and setting its first input field to reference the value number newly assigned to the array a, its second input field to reference the value number of index expression j, and its result value field to contain the value number of the assigned expression on the right of (3). Once this is done, indexed load operations may be treated in much the same way as ordinary binary operations; each assignment (3) to a location in the array a will, in effect, invalidate every fetch expression a(j) in the available computations stack except for the single expression a(i).

The data structures used by the basic block optimizer are as follows. The compiled code, held in a suitable intermediate form, constitutes the code text. Variables and their relevant attributes are represented by entries in the symbol table. A list of the calculations already performed, whose values are available for the elimination of redundant operations, is con-

tained in an available values stack. For efficiency, entries in this stack are located using an auxiliary hash table. Constants generated by the optimiser's constant propagation process are held in a constants table.

A description of the entries in the foregoing data structures will aid understanding of the remaining details of the basic block optimiser. Figure 2 shows the symbol table item representing a variable name.

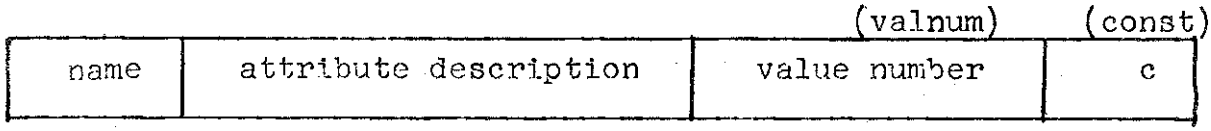


Fig. 2. Symbol table item.
(Parenthesised key words are names used in Table I)

In addition to the usual name-reference an attribute description fields, symbol table items include a value number field giving the number of the value a times the variable at any phase of the optimisation process, and a one-bit constant field, set to one if the value momentarily assigned to the variable is known to be a constant. Note again the value number field attatched to a variable or to an item in the code will be 0 if no value number has yet been assigned to the given variable or computation.

Figure 3 shows the form of the code text items associated with the various operations which may occur within a basic block.

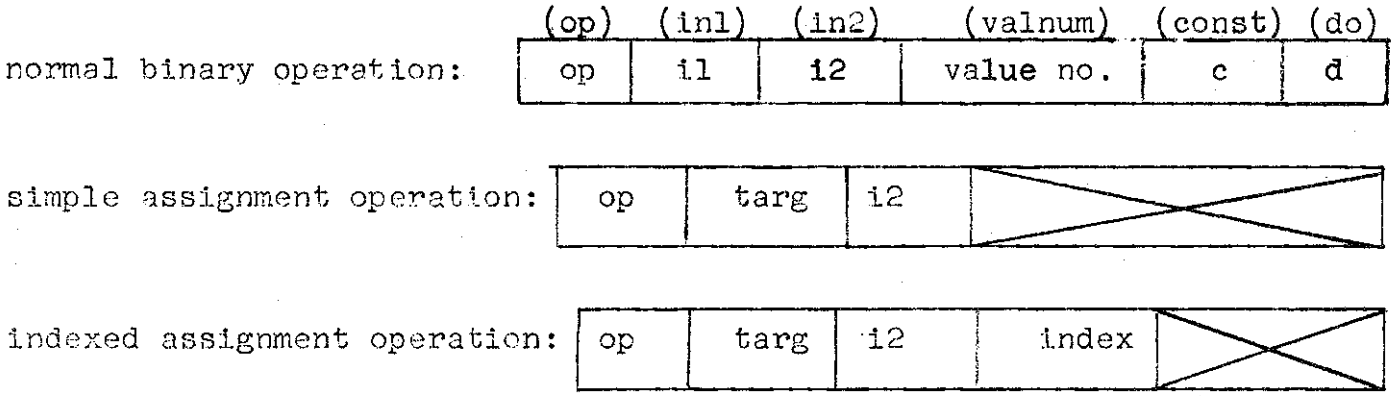


Figure 3. Form of items occuring in basic block of code text
(Parenthesised key words are names of corresponding fields in Table I)

Items representing normal binary operations have an operation code field and two input argument fields (i1 and i2) which reference either a variable entry in the symbol table or another calculation item in the code text. An additional field containing the value number of the result of an operation is maintained. This field is filled in during optimisation. Binary operation entries also contain two additional one-bit fields, c and d. The c or constant field is set to 1 if the result of an operation is known to be a compile time constant. The d or do bit is set to 0 if the binary operation is redundant and is to be suppressed during code generation, otherwise the d field is set to 1. An indexed assignment entry operation has the form explained previously, and shown in Figure 3; a simple assignment operation entry has a similar form, but does not involve any index field.

Table I below describes the basic block optimisation procedure in algorithmic detail. The procedure begins with a few initializations. A pointer, called now, which always indicates the code text item of current concern to the optimiser, is set to point to the first instruction in the basic block to be optimised. At the same time, a pointer called end is set to point to the last item in the block to be optimised; this pointer is used in the algorithm to determine the completion of optimisation of a given block. Pointers indicating the next available entry in the constant values and in the available computations stack are initialized, and the basic value counter curval used by the optimiser for generating new value numbers is also initialized to 1. After initialization, a main loop, starting at the label nextitem, is entered. The algorithm then examines the operation code contained in the operation code field of the code text item to be processed, and transfers, using a calculated go-to statement, to whatever subprocess is appropriate for treatment of an operation of each of the several possible types.

Ordinary binary operations are treated as follows. The two inputs to the operation are examined. If either of these input arguments has not yet been assigned a value number, a subroutine is called to make this assignment. Once this is done, value numbers are available for both input arguments. At this point, the input arguments are checked; if these input arguments are known constants a compile-time constant value for the result of the operation may be calculated, and transfer is made to a special procedure for carrying out this calculation. In the contrary case, the available computations stack is searched, using a subroutine find, to find an available computation item showing the same operation code and input value numbers as the computation item currently being processed. If no such item is found, the algorithm transfers to the procedure which must be employed in the non-redundant case. In the contrary, redundant case the current code text item is marked as redundant and non constant, and the value number determined from the corresponding item in the available computations stack assigned to it; then the algorithm loops back to its beginning to process the next code item in turn. Note that, in this redundant case, the code generation process which eventually follows will replace the operation occurring in the original code text by a direct reference to its result value, calculated by a previous operation.

In the nonredundant case, the operation indicated in the code text must be performed, and a corresponding new entry must be made in the available items stack. In this case, we build up the necessary available computations stack item by setting its operation field and its first and second input value fields, and by updating the hash table through which reference to the available calculations stack is made. The corresponding code text item is then marked as irredundant and non-constant, and a new value number is entered into the value number field of the code text item and of its corresponding available value stack item; the value number counter is then incremented and the algorithm loops back to process the next code item in turn.

i1, i2 are v1, v2

Table I. Algorithm for optimising a basic block.

now=pointer to first item in basic block;	initialise current item pointer and set up basic block end pointer;
end=pointer to last item in basic block;	
constptr=1;	initialise pointers to constant values
aptr=1;	stack and available items stack;
curval = 1;	initialise value number counter;
go to optype;	go to process first instruction;
nextitem:	
now = now + 1;	test for completion of block
if (now. gt. end) go to finish;	optimisation;
optype:	
item = code(now);	determine operation code of item to
op = op(item);	be processed;
goby kind (op) (computation,	transfer to subprocess appropriate to
simpassign, indxassign, indxload);	treatment of operations of various kinds;
computation:	
i1 = in1(item); i2 = in2(item);	determine two input argument code items;
if (valnum(code(i1)).eq. 0)	if the first input argument of the
call newval (i1);	operation has not yet been assigned a value number, call a subroutine to assign it a value number hitherto unused;
if (valnum(code(i2)).eq.0)	carry out corresponding procedure for
call newval (i2);	second input argument of current operation;
v1 = valnum(code(i1));	determine the value numbers of
v2 = valnum(code(i2));	input arguments;
if (const(code(i1)).and.const(code	if both input arguments are constants,
(i2))) go to fold;	compile time calculations can be done;
oldval = find(op,v1,v2);	else try to find available prior value number with identical operation and inputs;
if(oldval.eq.0) go to notavail;	returned value of 0 means no such
	calculation available;
else do(code(now)) = no;	else mark computation as redundant,
const(code(now)) = no;	non-constant;
valnum(code(now)) = oldval;	and assign value number found;

```
go to nextitem;
                notavail:


opn(avail(aptr)) = op;
val1(avail(aptr)) = v1
val2(avail(aptr)) = v2;
call hashin(aptr)

do(code(now)) = yes;
const(code(now)) = no;
valnum(code(now)) = curval;

valnum(avail(aptr)) = curval;

curval = curval+1;
go to nextitem;
                fold;

c1=constval(v1);c2=constval(v2);

c=calculate(op,c1,c2);

v=findconst(c)

if(v.eq.0) go to newconst;

                constop:
do(code(now))=no;
const(code(now))=yes;
valnum(code(now))=v;

go to nextitem;
```

```
loop back to process next code item;
in nonredundant case, an operation
 will be performed and a corresponding
 new entry must be made in the available
 items stack;
set operation field,
 first and second input value fields
for newly performed calculation;
update available calculations
 hashtable to reference new item;
mark computation  as irredundant,
 non-constant;
enter new value number into code item
 field;
assign same number to item in available
 values array;
increment value number counter;
loop back to process next code item;
beginning of sequence to handle
 compile-time constant calculations;
find constant values for both input
 arguments;
apply specified operation to argument
 values;
attempt to find calculated value in
 constants table;
if not available, transfer to "new
 constant" sequence;

flag current item as redundant
 and show that constant value is known;
enter value number refencing constant
 in current code item;
loop back to process next code item;
```

```
                newconst:
v=curval;                                  assign new value number for new constant;
curval=curval+1;                           increment value number counter;
val(const(constptr))=v;                    assign value number and calculated
cval(const(constptr))=c;                    constant value to new constant
                                            value table entry;
constptr=constptr+1;                       increment top pointer of constant
                                            value table;
go to constop;                             loop back to finish treatment of
                                            current code item;
                simpassign:                beginning of sequence for treatment
                                            of simple assignment operation;
variable=in1(item);                        determine variable to which assignment
                                            is made;
quantity=in2(item);                        determine quantity being assigned to
                                            variable;
if(valnum(code(quantity).eq.0)             if the right-hand side of the assignment
 call newval(quantity);                     operation has not yet been given a
                                            value number, call a subroutine to
                                            assign it a value number hitherto
                                            unused;
valnum(code(variable))=                    assign value number of quantity as
                valnum(code(quantity));     value number of variable;
const (code(variable))=                    variable also inherits constant bit
                const(code(quantity));      from quantity;

go to nextitem;                            loop back to process next code item;
                indxassign:
variable=in1(item);                        determine indexed variable to which
                                            assignment is made;
index=in2(item);                           determine index expression;
quantity=in3(item);                        determine quantity being assigned
                                            to variable;
if(valnum(code(quantity).eq.0)             if the right-hand side of the assignment
 call newval(quantity);                     operation has not yet been given a
                                            value number, call a subroutine to
                                            give it a value number hitherto unused;
if valnum (code(index).eq.0)               and do the same for the index quantity
 call newval(index);                        appearing in the assignment statement;
```

```
valnum(code(variable))=curval;

curval=curval+1;
opn(avail(aptr))=indexed fetch;
val1(avail(aptr))=variable;
val2(avail(aptr))=index;

valnum(avail(aptr))=
 valnum(code(quantity));
isconst(avail(aptr))=
 const(code(quantity));
call hashin(aptr);

go to nextitem;
                indxload:

i1=ini(item);i2=in2(item);
if(valnum(code(i1)).eq.0)
  call newval(i1);



if(valnum(code(i2).eq.0)
 call newval(i2);

v1=valnum(code(i1));
v2=valnum(code(i2));
oldval=find(op,v1,v2);


if (oldval.eq.0) go to notavail;

isconst=findc(op,v1,v2);

do(code(now))=no;
const(code(now))=isconst;
```

```
assign new value number to indexed
 variable;
increment value number counter;
construct new available value
 item representing indexed expression
 by setting operation field, first
 and second input value fields;
assign value number of quantity
 as value number of indexed expression;
indexed expression also inherits
 constnat bit from quantity;
update available calculations hashtable
 to reference new item,
loop back to process next code item;
sequence for handling indexed load
 operation;

if the first input argument of the
 indexed load operation has not yet
 been assigned a value number, call
 a subroutine to assign it a value
 number hitherto unused;
carry out corresponding procedure for
 second input argument (index) of
 load operation;
determine the value numbers of
 variable and index;
try to find available prior value
 number with identical operation and
 inputs;
returned value of 0 means no such
 calculation available;
determine constant attribute from bit
 of available entry found;
mark indexed load as redundant;
set constant attribute appropriately;
```

```
valnum(code(now))=oldval;
go to nextitem;
                finish;
return to main routine
 eventually to process another
 basic block of code.
```

```
assign value number found;
loop back to process next code item;
```

The procedure for handling a computation for which a compile time constant value is available is as follows. We first find the constant values of each of the input arguments of the computation in question. Next we apply the specified operation to these two argument values, obtaining a constant value for the result of the calculation. We then attempt to find this calculated constant in the existing constants table; if no such value is available in the constants table we enter such a value into the constants table, at the same time assigning this value a new value number and incrementing the value number counter. In either case, a value number representing the constant value with which we are dealing will be available. This value number is entered into the appropriate field of the current code item; this item is flagged as redundant and as having a known constant value, and the algorithm loops back to process the next code item in the basic block being optimized. So concludes the treatment of binary calculation items of ordinary form.

Simple assignment statements are treated as follows. The variable to which assignment is being made and the quantity being assigned to this variable are first determined. The quantity being assigned is checked to verify that it has been given a value number, and, if this is false, a value number is given to the quantity at once. Then the value number of the quantity is assigned to the target variable; the target variable also inherits a constant bit from the quantity being assigned, so that, if the quantity is known to be a constant, the variable inherits the attribute of having a known constant value. The

algorithm then loops back to process the next code item.

Indexed assignments are treated by a somewhat more complex procedure. The indexed array which is the target of the assignment, the index expresseion itself, and the quantity being assigned are all determined. Next, both the quantity and the index expression are checked to make sure that they have been given value numbers and, in the contrary case, new value numbers are at once specified. A new value number is also given to the indexed target variable of the indexed assignment being processed. At this point, a new item, corresponding to the indexed expression forming the left hand side of the assignment operation being processed, is set up in the available computations stack; this item represents the load of an indexed expression formally identical with the indexed target expression of the assignment being processed. The newly established available calculation item inherits the value number available originally attaching to the right hand side of the assignment operation being processed. When, in the subsequent functioning of the optimiser, an indexed load of a logically identical indexed expression is encountered, it is possible to reference a known value directly, no indexing operation being required. The available computation stack item established in connection with an indexed assignment operation also inherits a constant bit from the quantity forming the right hand side of the assignment operation; we note in this connection that indexed load operations, as distinct from normal binary operations, are taken to have known constant values if and only if the constant bit in a matching available computations stack entry is on.

After updating the available calculations hash table to reference the available calculation item added in the manner indicated, the optimiser loops back to treat the next code item in turn.

Indexed load operations are treated by a process which resembles the treatment of ordinary binary operations rather closely, except that, as remarked above, an indexed load operation is

taken to have a constant value not when both its inputs have constant values, but rather when there is a matching available computations entry showing the attribute "constant" in the appropriate one-bit field.

When every code item occurring in a basic block to be optimised has been processed, the basic block optimiser returns to the main routine from which it has been called; eventually it will be called again to process another basic block of code.

The procedures described in Table 1 may be sophisticated to obtain a somewhat improved optimisation level. Note to begin with that if, as a standard matter, the compiler producing the intermediate language code to be optimised always arranges the arguments of commutative operations like multiplication and addition in some standard order, the mathematical identity between the two terms of A * B + B * A, as well as other similar identities, will be detected. This will allow the elimination of instructions whose redundancy might otherwise not be seen. A similar, more sophisticated procedure would examine chains of arithmetic operations of the form A * B * C, and reassociate multiplications and additions in some standard order. This would lead to the detection of additional eliminable instructions.

Similar remarks may be made concerning the more sophisticated optimisations required in the presence of program flow; we now turn our attention to this more interesting class of optimisation algorithms.

3. Optimization by the Linear Nested Region Scheme.

The basic block optimizer described in the preceding section can be extended straightforwardly and without overwhelming difficulty to take program flow into account. The linear nested region optimization scheme to be described in the present section is such an extension. We may derive the necessary modifications of our prior algorithm by the following considerations. In optimizing a basic block, we take advantage of the serial structure of the code which it contains to deduce two fundamental facts:

i. Every calculation preceding an instruction I of the block will necessarily have been performed before the instruction I is encountered;

ii. The current value of any variable to which a value is assigned within the block is uniquely defined as the value established by the last preceding assignment.

As its name indicates, the linear nested region optimizer treats code to be optimized on the basis of the particular linear order in which it is presented (though, as we shall see, algorithms for the useful rearrangement of code order before optimization are available). In the context of this linear code order, the general linear nested region algorithm makes straightforward allowance for the partial failure, occasioned by the occurrence of forward and backward branches within the code, of the above basic principles i) and ii). In regard to ii) note that in the flow configuration shown in Figure 1, and on the assumption that no assignment to the variable b and occurring in the block B_3 precedes the use of this variable in B_3, but that such assignments are made in predecessors of both of the blocks B_1 and B_2, the value which b has in its indicated use is indefinite at compile time.

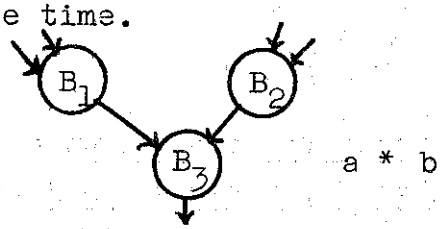


Figure 1. Use of a variable b after convergence of code flow.

The linear nested region algorithm, like the basic block optimizer described previously, will operate by assigning value numbers to each variable and expression occurring within a code. Values indefinite at compile time must be assigned value numbers never used before. Accordingly, the action of the algorithm on encountering the terminating label of either a forward or a backward branch is as follows:

1) Backward branch: Assign a new value number to every variable set in a block of code which precedes the label in question along the given backward branch. (See Figure 2.)

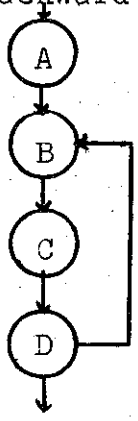


Figure 2. Elementary case of backward branch procedure. On entering block B, a new value number must be established for every variable to which an assignment is made in blocks B, C, or D.

2) Forward branch: Assign a new value number to every variable set in a block of code preceding the label in question and following the block from which the given forward branch originates. (See Fig.3.)

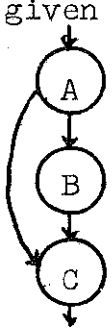


Figure 3. Elementary case of forward branch procedure. On entering block C, a new value number must be established for every variable to which an assignment is made in block B.

Rules 1) and 2) above ensure that, when the optimizer begins to process a block in which the particular preceding assignment which has set the value of a variable cannot be determined at compile time, the variable is assigned a value number never used before. This prevents any expression contained within the block and involving this variable from being eliminated as redundant relative to a formally identical expression occurring in a preceding block.

Assertion i) above will also fail in the presence of program flow. Note, to illustrate this point, that block C of Figure 3 can be entered without any of the instructions in block B having been executed. The linear nested region optimizer, like the basic block optimizer on which it is modeled, maintains a table of operations previously performed, whose values are available (available computations stack). The remark above makes it plain that on encountering the terminating label of a forward branch, every calculation occurring in a code block which lies between the origin and the terminus of the forward branch must be removed from this table. We may assert more generally that, unless we take explicit account of identical calculations occurring in several different positions within a code, only those calculations which lie in a predominating block A of a given code block B are available for the elimination of calculations redundant within B. Here, we define a code block A to be a predominator of B if and only if it is a logical consequence of the flow structure of the program in which these code blocks occur that the instructions in block A must be executed before control enters block B. The sophisticated optimizers to be described in the later sections of the present chapter will, in fact, detect the occurrence of identical calculations occurring at several different positions within a code, and use this information to eliminate redundancies; the simpler linear scheme to be described in the present section will only eliminate a calculation α as redundant if a single logically identical calculation occurs prior to α either in the same block as α or

in a block predominating the block containing α.

For example, the linear scheme described in the present section will not eliminate the third occurrence of a * b as redundant in the following example:

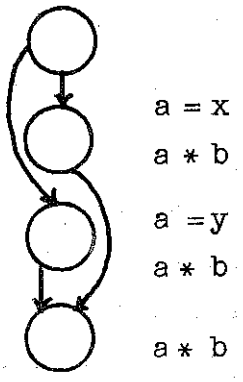


Figure 4. A redundant instruction not detected by the linear nested region optimizer.

The linear nested region optimizer to which the present section is devoted does in fact not even incorporate a precise criterion for predominance, but uses a simple approximate predominance notion derived from the actual sequential order in which blocks occur within a code as presented. The approximate predominance notion would actually be precise if an assumption implicit in the linear nested region optimization scheme, namely that a block B following a block A in serial order is always reachable directly from A via either a conditional transfer or normal control flow, were satisfied. Were this the case, we could calculate the predominators of every block B simply by maintaining a "predominator stack" of code blocks and causing it to vary appropriately as we process the set of call code blocks in serial order. On completing the optimization of a block A and beginning that of its immediate serial successor B, we would then have to add A to the predominator stack, but to remove from this predominator stack every block which is bypassed by a forward branch terminating at a point from which the block B may be reached without passing through any block preceding B.

Note that, as its name suggests, the predominator stack would then behave like a push-down stack, in that blocks would always be added at the top of the stack, and in that, when blocks are removed, the blocks removed would always form a contiguous interval of blocks at the very top of the stack. The algorithms to be described in the present section actually work not with a stack of code blocks, but with a stack of available calculations. When the optimizer processes a new calculation, this calculation is added to the top of an available calculations stack. When the optimizer enters a new block B of code, it must remove from this stack of available calculations every calculation which is bypassed by a forward branch terminating at a point from which the entry label of the block B may be reached without passing through any block preceding this label. Note, for example, that in Figure 5 every instruction lying in the indicated range α must be removed from the available computations stack when the optimizer encounters the label L, since the code in this range can be bypassed via the forward branch B and the two interlocking backward branches shown in that figure.

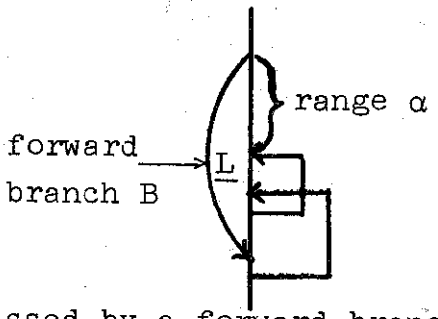
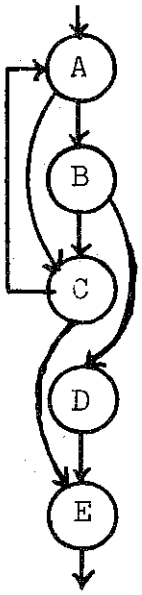


Figure 5. Code bypassed by a forward branch via an interlocking set of backward branches.

Note also that for this to be possible, the optimizer must be supplied with information "looking ahead" in the code, as, for example, the fact that the label L in Figure 5 is the target of a backward branch (occurring further on in the code) which interlocks a second backward branch whose scope contains the target label of the indicated forward branch B. In the present section, we shall describe detailed procedures by which information of this sort can be collected.

Later in the present section we shall also describe "code straightening" algorithms, which, by reordering the blocks constituting a code, bring the code into an order in which the serial approximation to predominance used by the linear nested region optimizer is not too badly inaccurate.[1/]

1/ If the code to be optimized has been "straightened" by the use of the code re-ordering algorithms described later in the present section, the approximate predominance notion used within the linear nested region optimizer will differ from true predominance only rather slightly in practical terms. For example, in code having the following flow



which is in an order acceptable the code-straightener to be described, the fact that B is not a predominator of block C will cause the optimizer to miss the fact that B is a predominator of block D.

The general linear nested region optimizer to be described not only performs the redundant expression elimination and constant propagation optimizations familiar from the preceding section, but also attempts to move loop-independent operations out of loops which contain them. The method used to determine the possibility of code motion may be described as follows. First of all note that, like the basic block optimizer, the linear nested region optimizer makes use of a value number counter to generate a sequence of unique value numbers. On encountering a label which terminates a backward branch (and thus begins a loop), the current value of this counter is recorded; then, as indicated above, we assign new values to all variables set within the loop just entered. As operations are subsequently encountered within the loop, the value numbers of the arguments of these operations are examined. Any operation both of whose arguments have value numbers which lie below the minimum value number associated with variables set within the loop necessarily has loop-independent arguments, and may be moved out of the loop. This general assertion, however, must bear a small caveat. Consider a loop which contains an embedded forward branch, as shown in Figure 6.

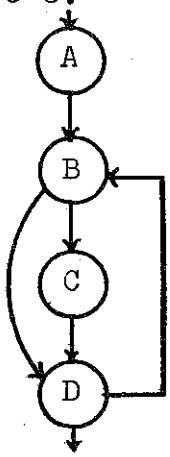


Figure 6. Loop containing embedded forward branch.

If the execution frequencies applying to the various blocks in the section of code shown in Figure 6 are not known a priori, it may be that the frequency with which the instructions in block C will be executed is smaller than the execution frequency

of the instructions in block C. If this is the case, motion of an instruction from block C into block A is undesirable. Moreover, in the situation shown in Figure 6, the motion of certain types of instructions from block C to block A may be logically impossible, in that such motion may give rise to effects which the programmer meant to rule out. Suppose, for example, that a floating-point multiplication a * b occurs in block C of Figure 6. Such a multiplication can, on many machines, generate a floating-point overflow interruption; the intent of the conditional forward branch shown in Figure 6 may be to avoid multiplication in certain cases which would generate interrupts. Thus it is possible that movving a * b from block C to block A might cause an undesired interrupt to occur, thereby frustrating the programmer's intent and violating the basic requirement that an optimized algorithm express an algorithm logically equivalent to the algorithm as originally given. Such a mishap is called a violation of the safety constraints applying to the optimization process. Note also that a predictive test of the kind contemplated above may occur in highly disguised form. Thus safety, i.e., the requirement that an optimizer transform a program into a second program which generates no more interrupts, sets no more indicator bits, etc., than the original forbids us to move certain classes of conditionally executed instructions out of a loop. Note, however, that the optimized program is allowed to generate fewer interrupts, etc., than the original program.

The situation becomes still more difficult in the case of a loop entered by a forward branch as shown in Figure 7.

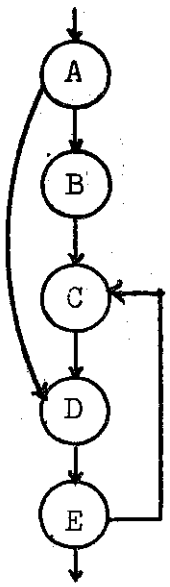


Figure 7. Loop entered by a forward branch.

A loop-independent instruction can only be moved out of a loop if a copy of it is placed at every entrance to the loop. To move a loop-independent instruction out of the block D or E, we would have to create additional copies of the instruction and place them appropriately, thus lengthening our code and complicating it with additional transfers. While such procedures are worth considering in connection with more complex optimizers than the linear nested region algorithm to which the present section is devoted, we shall, in the present section, not attempt to face the problems that arise in connection with the use of such procedures. (In fact, no entirely satisfactory procedure for handling code motion out of multi-entry loops of this kind is known.) The relatively simple algorithm to be described in the present section will never attempt to move code out of loops entered by a forward branch, unless these instructions follow the terminating label of the forward branch, and unless the instructions can be moved to a point preceding the origin of the forward branch.

Some of the force of this restriction can be seen in Figure 8A below.

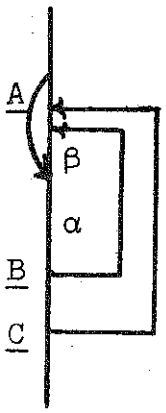


Figure 8A. Code motion out of loops entered by forward branches.

As has been indicated above, we shall not attempt to move any instruction at the position marked β out of loop B. An instruction which occurs at the position marked α can be moved out of the loop B, but only if it can be moved at least as far as the forward branch A, i.e., past the whole span of instructions bypassed by this forward branch. In this case, it is plainly necessary that instruction α move out of loop C also. Figure 8B below shows a somewhat more complex situation in which much the same observation can be made.

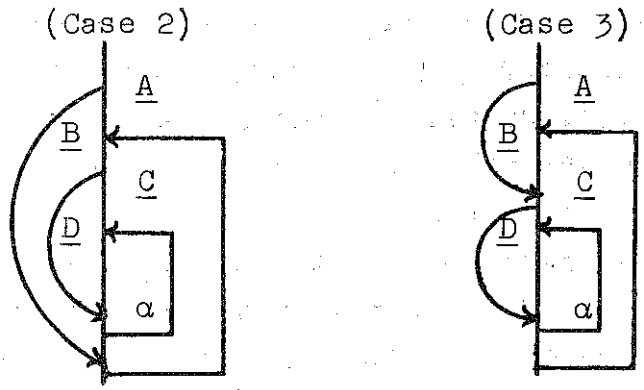


Figure 8B. A second and third case of loops entered by forward branches.

In either of the two cases shown in Figure 8B it may be possible to move the instruction α out of loop D by placing it immediately previous to the forward branch C; it may also be possible to move α out of the larger loop B by placing it immediately previous to the forward branch A. Note, in this connection, that points immediately preceding forward branches,

as well as points immediately preceding loops, may be destinations at which moved code is placed.

A circumstance connected with the motion of code out of loops makes it advantageous to use a modified intermediate code representation based rather directly on optimizer-assigned value numbers. Consider, for example, the simple loop shown in Figure 9A below.

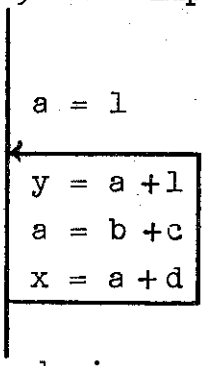


Figure 9A. Expression renaming during motion of code out of a loop.

Since the expression b + c occurring in Figure 9A is loop-independent, the optimizer would move it out of the loop shown in that figure, leading to the first situation shown in Figure 9B.

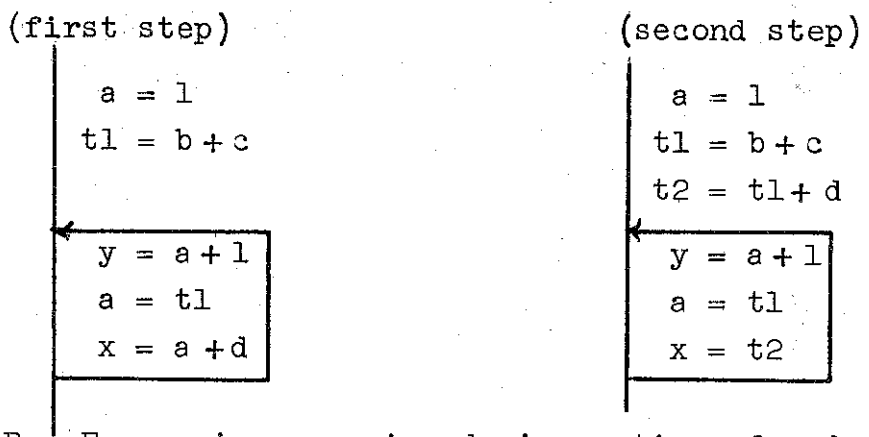


Figure 9B. Expression renaming during motion of code out of a loop, continued.

The pattern of value numbers which would be encountered in Figure 9B would then reveal that the sum a + d is loop-independent, so that this sum can be moved out of the loop. In doing so,

however, a + d is actually moved past a formal redefinition (a = t1 in Figure 9B) of the variable a; so that the moved expression to be placed at the head of the loop is properly t1 + d rather than a + d. Note that if this modification is not made, the resulting code is in error. The necessary modification becomes obvious if, as a standard matter, we identify specific operations by the value numbers, rather than the programmer-assigned variable names, of their operands. The association between value numbers and named variables changes with each assignment statement occurring in a code; nevertheless, in final code generation, we must know the source in which a given value, identified by its value number, is to be found. This information can be made available by associating, with each value number, both its original source (defined just below) and a list of all locations momentarily known to contain the value. By the original source of a given value number N, we mean either the first operation whose result has value number N, or if there is none such, the named variable, used as an operand in some other operation, first identified as having N as its value.

The following example will illustrate the general points made just above. Consider the simple code sequence

```
y = x* c
a = y
y = x + y
b = x
d = b * c
a = x + a
b = b + d   .
```

The following table shows the assignment of value numbers to variables and to expressions during the optimization of this code sequence, and also represents the modified code produced during this same optimization. Note that the detection of redundancies reduces the size of the modified code relative to that of the original code.

original code	Assigned Value Numbers a	b	c	d	x	y	x*c	x+y	b*c	b+d	x+a	Modified Intermediate Code
y = x * c			2		1	3	3					V1*V2(=V3);y=V3
a = y	3											a = V3
y = x+ y					1	4		4				V1+V3(=V4);y=V4
b = x		1										b = V1
d = b * c				3					3			d = V3
a = x + a	4										4	a = V4
b = b+ d		4								4		b = V4

Figure 10A. Pattern of value number assignments and modified intermediate code produced by linear nested region optimizer in a special case.

Figure 10B below represents the information concerning the

Op. No.	Modified Code	Value number sources, original and auxiliary 1	2	3	4	Final Code
1.	V1* V2 (=V3)	x;	c;	Op1;		
2.	y = V3	x;	c;	Op1;y		y = x * c
3.	a = V3	x;	c;	Op1;y,a		a = y
4.	V1+V3 (=V4)	x;	c;	Op1;y,a		
5.	y = V4	x;	c;	Op1;a	Op4;	y = x+ y
6.	b = V1	x;b	c;	Op1;a	Op4;y	b = x
7.	d = V3	x;b	c;	Op1;a,d	Op4;y	d = a
8.	a = V4	x;b	c;	Op1;d	Op4;y,a	a = y
9.	b = V4	x;	c;	Op1;d	Op4;y,a,b	b = a

Figure 10B. Available sources for values and final code produced by linear nested region otpimizer in a special case.

original source of a value, as well as the list of locations momentarily known to contain the corresponding value, as this information would be maintained by the optimizer during its

processing of the code sequence shown above. Final code generation would in actual practice incorporate a register-allocation algorithm and consequently be somewhat machine-dependent. This final code generation process would, in the manner indicated above, scan through the modified code, maintaining a list of all registers and locations momentarily containing a given value. Whenever a particular value was required, it would be obtained in whatever manner was easiest, i.e., preferably from a register, otherwise from a core location. In any case in which a value required at a subsequent stage of code generation was found not to be available in any register or core location, the code generator would allocate a cell for the storage of the value, interpolate a store into this cell immediately after the operation in terms of which the value is originally calculated, and fetch the value, when necessary, from this cell.

Note again that our optimizer begins with an initial code text representing an unoptimized instruction sequence and transforms this into an optimized modified code text in which instruction operands are generally described by their value numbers. The details of this process will be set forth in what follows. Code generation, given a particular computer, will take the modified text to a form much closer to actual machine code.

The overall structure of our linear nested region optimizer is as follows. A first forward pass over the code to be optimized collects information concerning transfers, labels, as well as assignment information, all of which is required during a following forward pass. This optimizer "prepass" is described algorithmically in Table III below. During a second pass, summarized in Table I and described in Table II below, the bulk of optimization, including constant propagation, elimination of redundant calculations, and motion of code out of loops is performed. Additional processing, described later

in the present section, can be added if optimization by reduction in strength is to be carried out.

The principal data structures used by the optimizer are as follows. The compiled code, held in its initial intermediate form, constitutes the code text. The optimized compiled code in modified form constitutes the modified code text. Variables and their relevant attributes are represented by entries inaa standard symbol table. A list of those calculations already performed, and whose values are available for the elimination of redundant operations, is contained in an available computations stack. For efficiency, entries in this stack are located through an auxiliary hash table. Values of constants, as required by the constant-propagation process within the optimizer, are held in a constants table. A stores array, consisting of an ordered sequence of entries, the n-th of which references the particular variable which is the target of the n-th store, is also used. The optimizer uses a special data structure, called a loop and branch stack, to record information concerning the pattern of forward and backward branches in the local section of code currently being optimized. A branch item list, consisting of an ordered set of references to the successive branch items occurring in the code text, serves for the rapid location of all branches and labels within the text.

Table I below describes, in broad outline, the structure of the main pass of the linear nested region optimizer. The reader should note the way in which this algorithm incorporates the main goals and constraints described earlier in the present section.

Table I. Main pass of the linear nested region optimizer. Overall structure of algorithm.

	Initialize data structures and pointers.
Repeat:	Advance to next code text item, and test for termination.
	Determine operation type, which may be computation, simple assignment, indexed assignment, indexed load, subroutine call, function call, forward branch, label, or backward branch. Transfer accordingly to appropriate subprocess.
Computation:	Eliminate redundant computations using available computations stack.
	Detect compile-time constants and perform corresponding calculations.
	If calculation is independent of a containing loop, move it out of loop and out of largest possible surrounding loop.
	Post new entries in available computations stack as appropriate.
	Go back to repeat.
Simple assignment:	Enter value number of right-hand expression of assignment in descriptor of "target" variable occurring on left hand side of assignment.
	Go back to repeat.
Indexed assignment:	Post available calculation item representing indexed load corresponding to indexed assignment being processed.
	Give new value number to target array of assignment.
	Go back to Repeat.

Indexed load:	Eliminate redundant indexing operations using available computations stack. Post new entries in available computations stack as appropriate. If index quantity is independent of a containing loop, and no array entry is set within the loop, move the current indexed load out of inner loop and out of largest possible surrounding loop. Go back to Repeat.
Subroutine call:	Assign a new value number to every variable whose value may be changed by the subroutine. Go back to Repeat.
Function call:	Assign a new value number to every variable whose value may be changed by the subroutine. Assign a value number to represent the value of the function. Go back to Repeat.
Forward branch:	Post forward branch on branch and loop stack to indicate that the code which follows lies under a branch. Go back to Repeat.
Label:	Remove from the available computations stack all items which represent computations performed under a forward branch from which the current label may be reached by a series of backward branches; Using the stores array, give a new value number to every variable whose value is set within a forward branch from which the current label may be reached by a series of backward branches; Remove all forward branches terminating at the current label from the loop and branch stack.

Post a new loop and branch stack item for each backward loop terminating at the current label, to indicate that code which follows lies in a loop.

Give a new value number to every variable which is the target of an assignment either lying within the loop or occurring at a point from which a place within the loop can be reached without passing through the first instruction in the loop.

Go back to Repeat.

Backward branch: Redefine the starting point of each loop represented in the loop and branch stack which interlocks with the backward branch B being processed; cause each such loop to show the start of B as its own start.

Remove the loop B from the loop and branch stack.

Go back to Repeat.

[End of Algorithm]

A closer description of the entries in the various data structures which it uses will aid in more detailed understanding of the linear nested region optimizer. Figure 11 shows the symbol table item representing a variable name.

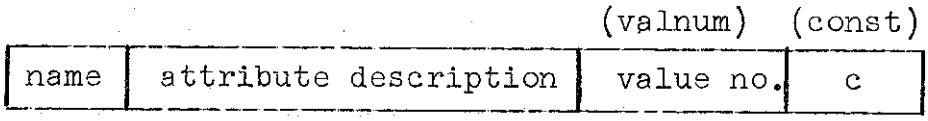


Figure 11. Symbol table item.
(Parenthesized names define modes of field reference in Tables II, III, and IV.)

In addition to the usual name-reference and attribute description fields, we append a value number field giving the number of the value assigned to the variable at any given phase of the optimiza-

tion process, and a one-bit constant field, set to 1 if the value momentarily assigned to a variable is known to be a constant. Note that the value number field attached to a variable or to an item in the code text will be zero if no value number has yet been assigned to the given variable or computation.

Note also that, as a matter of convenience in the algorithmic descriptions to follow, we assume the "symbol table" as ordinarily conceived to be incorporated into the code text. This allows us to assume that each operation item in the code text contains argument pointers which are of standard form, irrespective of whether the argument's operations are variable names or the results of earlier operations. On the other hand, the detailed algorithms described in what follows do not expect to encounter symbol table items among the items they scan. It may be assumed therefore that the compiler preparing the code text segregates all symbol table items in an initial section of code text; alternately, trivial modifications may be incorporated into the algorithms which follow, to allow them to bypass symbol table items.

Figure 12 shows the form of the original code text items associated with various non-transfer operations.

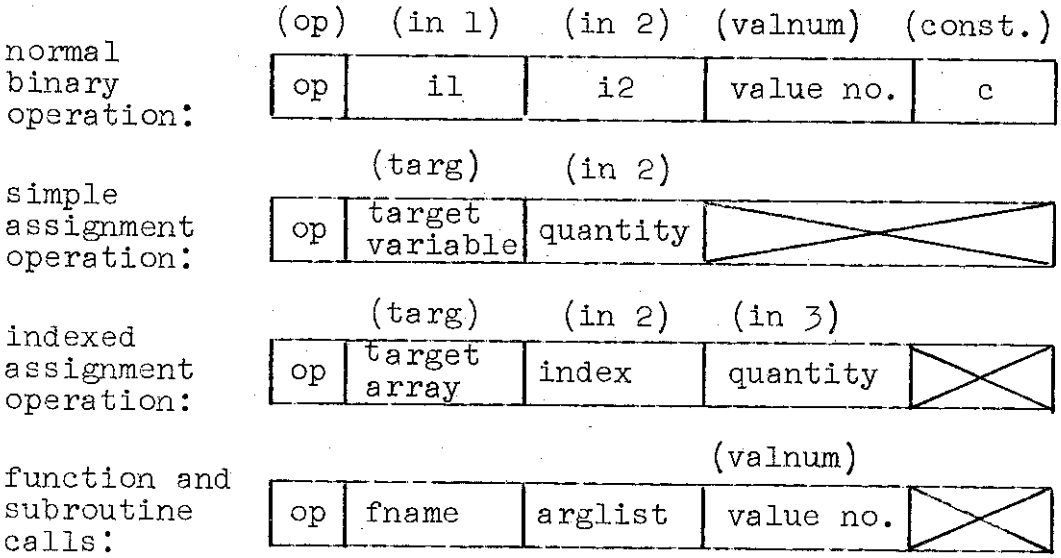


Figure 12. Non-transfer item entries in code text. (Parenthesized names define modes of field reference in Tables II, III, and IV.)

These entries have a relatively standard form. Items representing normal binary operations have an operation code field and two input argument fields (i1 and i2) which reference either a variable entry in the symbol table or another calculation item in the code text. An additional field containing the value number of the result of an operation is present in the entry representing a binary operation. This field is filled in during optimization. Binary operation entries also contain an additional one-bit field c. This c or constant field is set to 1 if the result of an operation is known to be a compile time constant.

The remaining code text items designating non-transfer operations have similar but appropriately modified forms. The text item representing an indexed assignment operation contains, besides its operation code field, three fields which respectively designate the target variable of the assignment operation, the variable or the computation item providing the value to be stored, and the variable or the computation item providing the index governing the store. No value number field is provided in assignment entries, since our algorithms associate a value number with the target variable of an assignment rather than with the store operation itself.

Figure 13 shows the form of the modified code text items corresponding to the above original code text items.

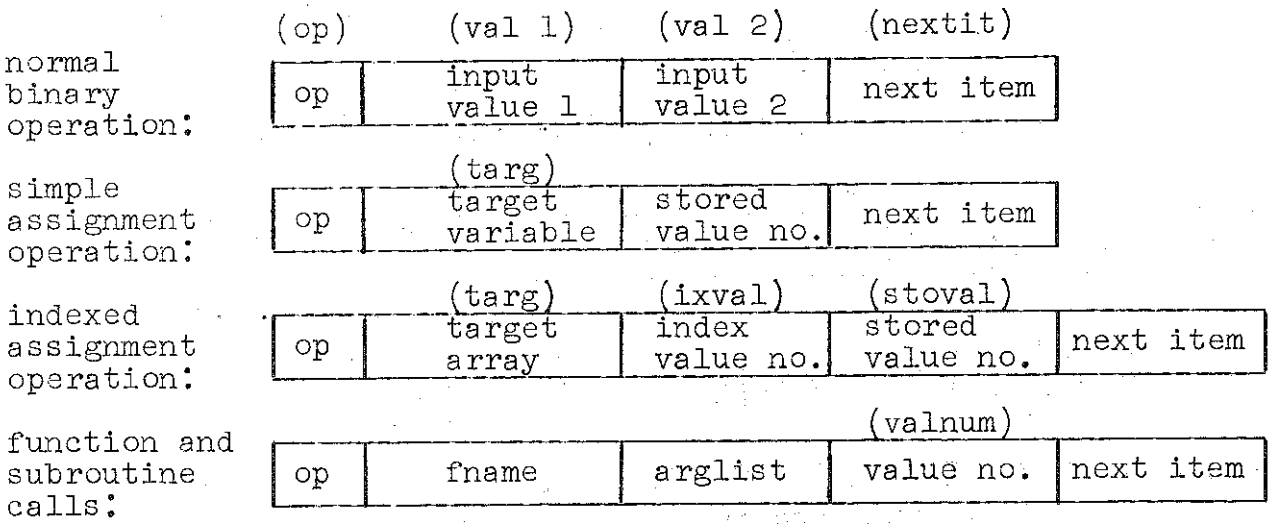


Figure 13. Non-transfer items in modified code text. (Parenthesized names define modes of field reference in Tables II, III, and IV.)

Each of these items is furnished with a next item pointer field, not present in the original code text; since our algorithm must occasionally insert items in the modified code text, we include these fields in order to be able to treat the modified code text as a list.

We allow one additional type of pseudo-operation entry to appear in the modified code text, for use in cases in which a previously unused value number is assigned to a variable (as, for example, on entering a loop, cf. the detailed algorithms below). Such a "pseudo-assignment" operation will have the same general format as an actual (simple) assignment operation in the modified code text, but will have a distinguishing operation code value. Items of this kind are needed in order that it be possible to reconstruct all details of the changing associations between value numbers and programmer variable names.

Three types of transfer items can occur in the code text; forward branch items, backward branch items, and label items. The first pass of the optimizer numbers these items in order of their occurrence, thus assigning to each a branch item number by which it can be referenced. Branch item numbers are used by our algorithms at various times to determine the relative position of pairs of branch items. Both forward and backward branch items contain target, condition, and (in the modified code text) next item fields. The target field of a branch item references the label item which is the target of the transfer represented by the branch item; the condition field references the variable or computation controlling a conditional transfer; in the modified code text, this field contains a value number. If a transfer is unconditional, the condition field is left blank. The next item field of a transfer item (in the modified code text) like that of a non-transfer item, points to the next entry in the code text and structures the code text as a list in which insertions and deletions may conveniently be made.

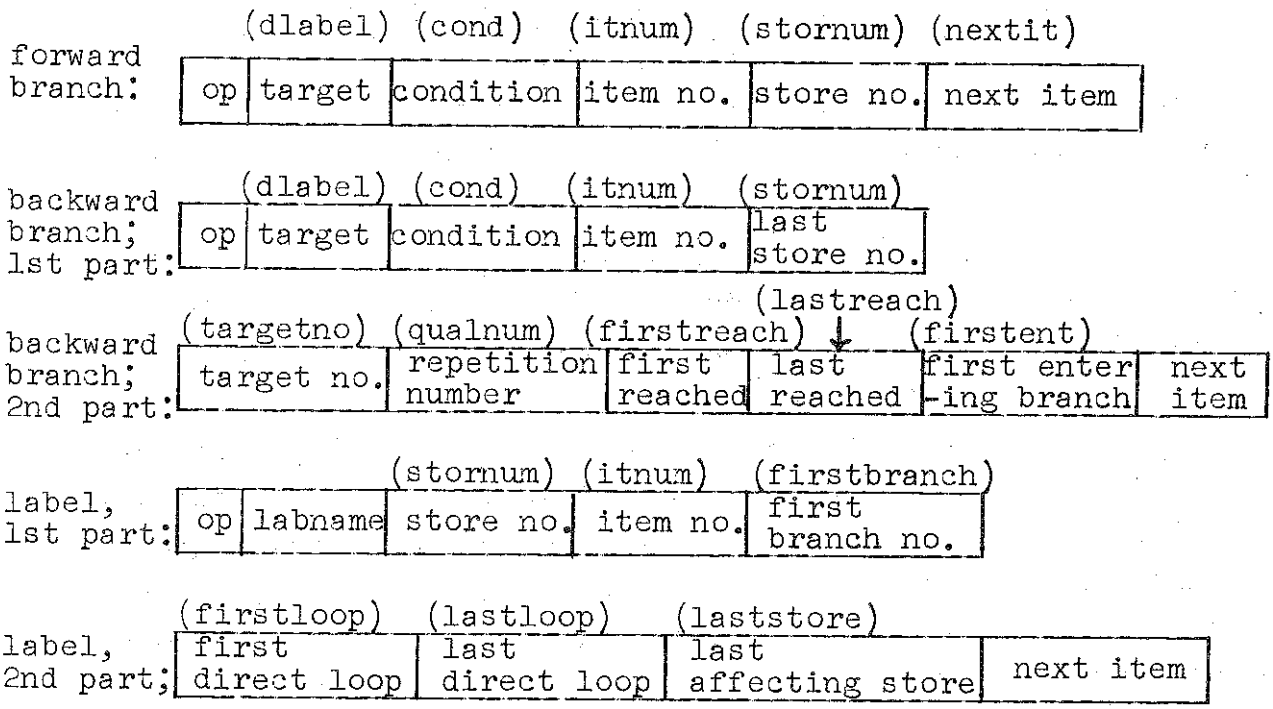


Figure 14. Transfer items in original and modified code text. (Parenthesized names define modes of field reference in Tables II, III, and IV; note that the "next item" field is present only in the modified code text, and that the "condition" field contains an operation pointer in the original code text, but a value number in the modified code text.)

A forward branch item contains a store number field in which is recorded the serial number of the first assignment lying within the scope of the forward branch. A backward branch item contains a (roughly similar) last store number field. However, the procedure for determining the value to be entered into this field is more complex than that employed in connection with forward branches; the value to be entered in this field is defined as the serial number of the last store operation from which a point within the scope of the backward branch in question may be reached via a series of backward branches (see Figure 15).

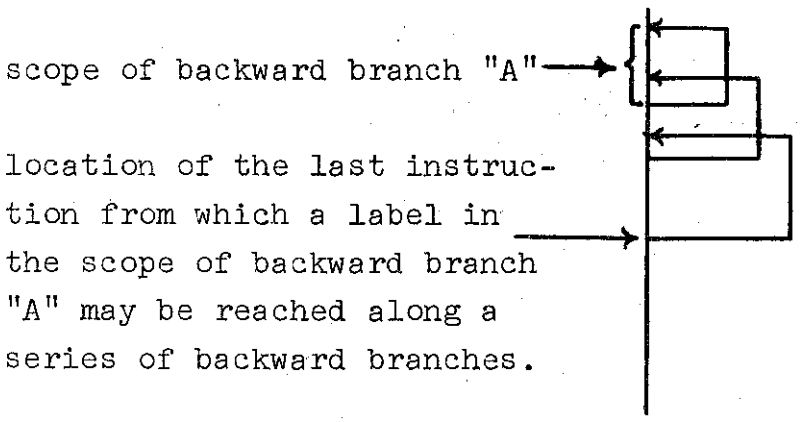


Figure 15. Interlocking backward branches and their effect on the last store field of a backward branch code item.

A backward branch item contains a number of additional fields, and all in all contains a larger number of fields than either a forward branch or a non-transfer item in the code text; for this reason, each backward branch item is broken up into two parts. The first of these two parts contains the fields already described. The second part of the two-section item representing a backward branch contains four fields. The first of these gives the branch item number of the label which is the target of the backward branch. The second gives the number of preceding backward branches having the same target label as the backward branch represented by the item in question. The third and fourth fields are respectively pointers to the beginning and end of a list of labels, called the "backtarget list", associated with each backward branch during the data-collecting first pass that precedes the main pass of our optimizer. (Cf. Table IV below for a detailed account of this initialization pass.) Both of these fields are pointers to items in an auxiliary transfer and label list area. In general, the items on this latter list have the form shown in Figure 16.



Figure 16. Data fields of auxiliary transfer list item. (Names in parentheses give field reference modes used in Tables II, III, and IV.)

The first two fields in Figure 16 contain the chaining information necessary to structure a circularly chained list of transfer items; the second field references a particular label or loop item, i.e., a particular code text entry representing a label or a backward branch. (The purpose of listing backward branches is explained in the next paragraphs.)

Since the fields which must be associated with a label item L in the code text are numerous, such an item is also represented in two parts. The first part contains an operation code field designating the item as a label and contains a pointer to the symbol table entry wherein a specification of the label name and of its attributes is to be found. The next field gives the serial number of the first assignment operation following L. A fourth field gives the branch item number of L. The fifth and last field in the first of the two code text entries representing a given label gives the branch item number of the first forward branch terminating at a point from which L may be reached by a series of backward branches (cf. Figure 5 above and the paragraphs of discussion associated with that figure); if no such branch exists, this field is left blank. The second of the two entries representing a given label contains two pointers to a set of auxiliary transfer list entries. Associated with each label by means of these transfer list entries is a list of all these backward branches terminating at the given label. The necessary list entries are maintained in the auxiliary transfer and label list area described above, and have the form shown in Figure 16. The first field in the second part of a label entry references the first item on this list; the second field references the last item on this list. The next field in the second part of a label entry references the number of the last assignment operation in the code text from which the label may be reached by a succession of backward branches; cf. Figure 15 and the explanations above directly related to this figure.

Figure 17 shows the form of a typical item in the available computations stack.

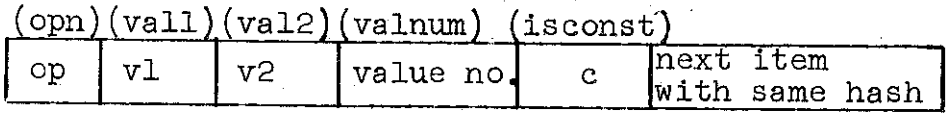


Figure 17. Item in available computations stack. (Names in parentheses in mode of references to corresponding fields in Tables II, III, IV.)

The first three fields contain an operation code and the value numbers of the two inputs to this operation. The next field gives the value number of the result of the operation. An additional one-bit constant field is used only for indexed load operations, and is set to 1 if this operation is known to yield a compile-time constant (cf. the relevant section of the algorithm of Table II). To determine whether or not a given ordinary computation is redundant, the optimizer algorithm searches the available computations stack for an item with a given operation code and pair of value numbers as inputs; if an item with the desired characteristics is found, the value number of the operation result is at once available. To facilitate the necessary search, and, at the same time, to allow the removal of items which is necessary on encountering a label terminating a forward branch, a combined hash-table-stack technique is appropriate. This technique is as follows. With the available computations stack, which, like all stacks, is provided with a top pointer, we associate a hash-table of a suitable size. This hash-table is entered using a hash computed from the first three fields (search key) of the item shown in Figure 17. The hash table entry found contains the index, within the available computations stack, of the first item with the given hash. Each item in the available computations stack stack then points to the next item with the same hash. The available computation search routine chains along these pointers until either a zero pointer is

encountered or until the stack top pointer index value is exceeded. Whenever the overall optimizer algorithm encounters a label item terminating a forward branch, it removes, from the available computations stack, a certain contiguous set of computation items, extending in every case from the top of the stack to some point within the stack. (Cf. the discussion of predominator blocks in the earlier paragraphs of the present section.) Giving the available computations stack the data structure described above allows us to remove these items simply by reducing the stack top pointer to whatever value is appropriate. If a hash table item subsequently is found to point to an entry in the available computations stack not having the correct hash, this always indicates that no entries in the available computation stack have the correct hash, so that the hash table item in question is to be regarded as being blank and may be reset to a new value.

Figure 18 shows the form of entries in the loop and branch stack which the optimizer algorithm uses to keep track of the local configuration of forward and backward branches with which it is momentarily concerned.

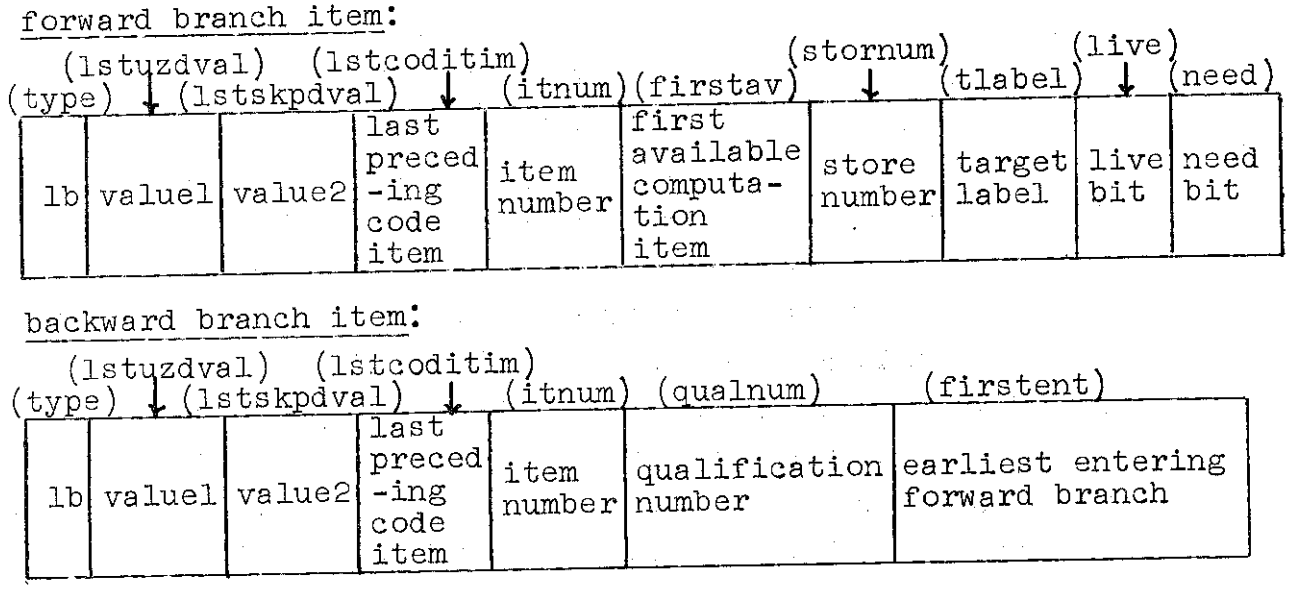


Figure 18. Entries in loop and branch stack.
(Names in parentheses give field reference terms used in Tables II, III and IV.)

The first field, lb, is a one-bit field which flags the item as being either a forward or a backward branch item. Three additional fields are reserved in both forward and backward branch items, corresponding to the fact that either loop-heads or backward branches can be destination points for moved code. The first of these records the last value which the value number counter has attained before entry into the conditional scope defined by a forward branch or the loop represented by a given backward branch item, as the case may be; the second of these fields records the first value of the value number counter associated with calculations lying within the loop or conditional scope. When, on entry into a given loop or conditional scope, the optimizer establishes a new entry on the loop and branch stack, the value number counter is incremented by some standard value (for example, 100), and these two fields of the new loop and branch stack item are filled in. This procedure reserves a range of value numbers for assignment to computations subsequently moved out of loops, and makes it easy to test instructions for loop independence. The last of the set of four fields common to loop and branch stack entries records the position, in the modified code text, of the last code item immediately preceding the beginning of a forward branch or loop. This defines the position at which code items being moved out of loops may be inserted. As explained in more detail below, the location shown in this field of a loop entry E must be adjusted whenever a loop begun before E happens to terminate before E is terminated (case of interlocking backward branches).

A forward branch item records the number of the first available computation stack item referencing a computation following the forward branch. The value inserted in this field when the forward branch item is established is simply the momentary value of the available computation stack top pointer at the moment when the occurrence of a forward branch code item

causes a new loop and branch stack entry to be made. A forward branch item also records the branch item number of the forward branch and the number of the first store operation following the forward branch. A forward branch item contains a 1-bit live bit field, set to 1 initially, and reset to zero when the target label of a forward branch is encountered. On encountering a label during our optimization scan, we will (cf. Table II below and the discussion associated with this table) remove all but the first forward branch item terminating at the given label from the loop and branch stack. This one branch item may, however, be flagged as terminated and kept, to allow the motion of code to the point just preceding the corresponding branch, when appropriate. Finally, a forward branch item contains a 1-bit need field, set to 1 if it is ever discovered that the forward branch is the first forward branch entering a loop; if this bit is set, it prevents removal of the forward branch item from the loop and branch stack, which would normally take place when the optimizer encountered the target label of the corresponding forward branch.

Backward branch items are entered into the loop and branch stack whenever a label item terminating one or more backward branches is encountered in the code text. As many separate loop and branch stack entries are made as there are backward branches terminating at the given label. Information allowing the determination of this number is available in the code text item representing the label in question; cf. Figure 14. In addition to the fields already described, a backward branch item in the loop and branch stack also records the branch item number of the label generating it, and the branch item number of the earliest forward branch entering the given loop.

Table II describes in detail most of the procedure employed by the principal (second) pass of the linear nested region optimizer. This algorithm makes use of eight subroutines: maymove, which is called during the processing of a computation

item to determine whether or not the item can be moved out of one or more loops; find, which is a simple search routine for locating items in the available computations stack; findc, a related search routine which returns the constant bit of an item which it has found; hashin, which inserts a new entry into the available computations stack; newval, which assigns a value number to a variable lacking one; newvalno, which assigns a new value to all variables addressed by stores in a given range of code; calculate, which applies a specified operation to a pair of constant inputs to achieve the calculation of a compile-time constant; findconst, which searches the constants table to see if a given constant has already been entered into this table. A detailed algorithm for maymove, the only nontrivial subroutine among these eight, is given in Table III.

Note also that

a) newval assigns a reserved value number and increments the reserved value number counter preval as long as reserved value numbers are still available; otherwise newval assigns a standard value number and increments the standard counter curval. (Cf. the short section of Table II below following the label newconst.)

b) Both newval and newvalno insert one pseudo-assignment operation into the modified code text for each variable to which they assign a new value number. As already explained, the information contained in these pseudo-operations is needed if one is to be able to reconstruct the correspondence between value numbers and programmer names using only information contained in the modified code text.

The algorithm of Table II begins with a few initializations. In particular, a pointer, called now, which always indicates the code text item of current concern to the optimizer, is set to point to the first instruction in the code text; by convention, this instruction is a dummy no-operation immediately preceding all executable code text items. Both a main value number counter curval and a reserved value counter preval are initialized; the

latter to 1, the former to 1000. The curval counter is used to assign value numbers in the normal case; the preval counter is used to assign value numbers either to new constants calculated at compile time, or to variables encountered before any value number has been assigned to them; such values are therefore independent of all loops. Our procedure thus reserves a range of value numbers signifying values which may be calculated outside of all loops; if the program being optimized is so large that all value numbers in this range are used, we switch to use of curval rather than preval. After initialization a main loop starting at the label nextitem is entered; this loop begins with a termination test; in the non-termination case, the algorithm advances the now pointer to reference the next following item of code text, determines the operation code contained in the operation code field of the item to be processed, and transfers, using a calculated go-to statement, to whatever subprocess is appropriate for treatment of an operation of each of several possible types. The necessary transfer is governed by the values held in an array kind(op) which assigns an operation type to every operation code which can occur in the code text.

The first additional optimization to follow redundant expression elmination and constant detection is code motion. A subroutine maymove is employed to determine whether the operation currently under examination may be moved out of one or more loops. This subroutine, described in more detail in Table III, compares the maximum of the value numbers of the two input arguments of the current calculation with appropriate value number fields in relevant backward branch items momentarily contained in the loop and branch stack. Maymove returns both an indication of the place in the modified code text to which a movable instruction is to be moved and the value number to be assigned to a moved instruction. Whenever an instruction is moved, the loop and branch stack entry representing the

largest loop out of which it is possible to move the given instruction is updated to reflect the fact that one additional operation has been moved out of the loop.

In those rare cases in which so many operations have been moved out of a loop or to the head of a forward branch as to make it impossible to assign distinct value numbers in the reserved range associated with the loop or forward branch to every moved operation, we assign unreserved value numbers and increment the value number counter. In such unusual cases our algorithm may fail to detect the movability of logically movable instructions subsequently encountered.

Indexed load operations are handled in much the same way as ordinary computations, except in regard to the treatment of compile-time constants. The treatment of these operations, as well as the treatment of indexed store operations, is described in detail in Table II.

Subroutine call items in the code text are handled by assigning new value numbers to every argument of the subroutine (if additional information concerning the subroutine were available, it would be sufficient to assign new values only to those subroutine arguments which the subroutine might change), as well as to every "common" variable or array which can possibly be modified before return from the subroutine occurs. Whenever a new value number is assigned to a variable or array name, a pseudo-assignment operation is added to the modified code text, making it possible to reconstruct the correspondence between value numbers and programmer names. Function call items occurring in the code text are handled like subroutine call items, except that the name of a function must be treated as an additional argument of the function. Note that, if the optimizer is informed that no call to some certain function F can have "side effects", i.e., that no such call causes re-usable information to be stored anywhere but in the cell containing the returned function value, a call to F may be treated simply as a multi-argument operation.

In this case, redundant calls to the function can be eliminated, and function calls occurring within loops can be moved, under restrictions which are evident generalizations of those applying to ordinary binary operations. This optimization may be quite valuable; the algorithms which follow, however, do not include it.

The remaining parts of the algorithm shown in Table II describe the processes necessary for the treatment of forward branches, backward branches, and label items occurring in the code text. On the occurrence of a forward branch item, the algorithm sets up a new entry at the top of the loop and branch stack, posting its destination label field, its branch item number, the serial number of the last preceding store operation, the number of the first item in the available computations stack which lies within the forward branch, the last code item preceding the forward branch, and the current state of the value number counter. A range of some 100 value numbers is skipped, thereby reserving these value numbers for such code items as may subsequently be moved to the head of the branch, and the limit of this skipped range of value numbers is posted. A slightly changed copy of the branch item, in which the condition governing a conditional branch is represented by a value number, is also added to the modified code text.

Forward branches are terminated and loops initiated by the occurrence of a label item L in the code text. The procedure necessary for treating such items is shown in Table II, following the point marked label in the algorithm of that table. All forward branches terminating at L are handled before any loops beginning at L are considered. The forward branch terminating procedure is as follows. The first forward branch item representing a forward transfer which terminates at a point from which L may be reached by a series of backward branches is determined; the section of code lying between this forward transfer item and the terminating label defines a maximal

"forward scope" terminating at the label in question. (Cf. Figure 5 above and the discussion associated with this figure.) Every calculation belonging to this maximal scope is removed from the available values stack. Every variable which is the target of a store operation lying in this scope is assigned a new value number. After this is done, the items on the loop and branch stack lying above the forward branch item corresponding to the first forward branch terminating at the current label are processed. Every forward-branch-representing loop and branch stack item which terminates at the label being processed and which is not the first forward branch entering any loop is suppressed. Backward branch items are merely moved to keep the loop and branch stack in a packed position. Each item which either represents a

Table II. Detailed algorithm for main process of linear nested region optimizer.

now = dummy leading no-op in code text;	initialize current item pointer used by main process;
item = code(now);	
constptr = 1;	initialize pointers to constant stack, available items stack, loop and branch stack;
aptr = 1;	
lbptr = 1;	
curval = 1000;	initialize value number counter and reserved value number counter;
preval = 1	
nextitem:	
if now counter references last code item go to finish	test for completion of optimization pass;
now = now + 1;	if not finished, advance to next item of code;
item = code(now);	
op = op(item);	determine operation code of item to be processed;

```
goby  kind(op) (computation,      transfer to subprocess appropriate
     label, simpassign,            for treatment of operations of
     indxassign, indxload, call,   various kinds;
     function, branch, loop,label)

     computation:                  beginning of process for treating
                                    normal binary operations;
i1 = in1(item);i2= in2(item);      determine two input argument
                                    code items;
if(valnum(code(i1)).eq.0)          if the first input argument of
     call newval(i1);               the current operation has not
                                    yet been assigned a value number,
                                    call a subroutine to assign it a
                                    value number hitherto unused;
if(valnum(code(i2)).eq.0)          carry out corresponding procedure
     call newval(i2);               for second input argument of
                                    current operation;
v1 = valnum(code(i1));             determine the value number of
v2 = valnum(code(i2));              input arguments;
if(const(code(i1)).and.            if both input arguments are constants,
 const(code(i2))) go to fold;       compile time calculations can be done;
oldval = find(op,v1,v2);           try to find available prior value
                                    number with identical operation
                                    and inputs;
if(oldval.eq.0) go to notavail;    returned value of 0 means no such
                                    calculation available;
else const(code(now)) = no;        else mark non-constant;
valnum(code(now)) = oldval;        and assign value number found;
go to nextitem;                    loop back to process next code item,
                                    placing no item in modified code text;

     notavail;                     in nonredundant case, an operation
                                    will be performed and a corres-
                                    ponding new entry must be made
                                    in the available items stack;
```

```
opn(avail(aptr)) = op;                  set operation field,
val1(avail(aptr)) = v1;                  first and second input value
val2(avail(aptr)) = v2;                  fields for newly performed calcu-
                                         lation;
call hashin(aptr);                      update available calculations
                                         hashtable to reference new item;
const(code(now)) = no;                  mark computation as non-constant;
call maymove(max(v1,v2),yesno,          call major subroutine to determine
     place, valnumber)                   if calculation is movable and new
                                         location and value number of
                                         calculation;
if(yesno.eq.yes)go to move;             if movable, go to code motion sequence;
else valnum(code(now))=curval;          else enter value number into
                                         code item field;
valnum(avail(aptr)) = curval;           assign same numerr to item in
                                         available values array;
curval = curval + 1;                    increment value number counter;
using information from
 appropriate fields of
 avail(aptr), add operation
 item to end of modified
 code text;
go to nextitem;                         loop back to process next code item;
     move:
valnum(code(now))=valnumber;            enter subroutine-supplied value
                                         number into code item field
valnum(avail(aptr))=valnumber;          assign same number to item in
                                         available values array;
using information from appropri-
 ate fields of avail(aptr), set
 up modified code text item in
 place supplied by maymove
 subroutine;
go to nextitem;                         loop back to process next code item;
```

```
     fold;

c1 = constval(v1);
     c2 = constval(v2);
c = calculate(op,c1,c2);

v = findconst(c);

if(v.eq.0) go to newconst;

     constop:
const(code(now)) = yes;
valnum(code(now)) = v;

go to nextitem;

     newconst:
if(preval.ge.1000) go to alt;


v = preval;
preval = preval + 1;

     altback:
val(constable(constptr)) = v;
cval(constable(constptr))= c;

constptr = constptr + 1;

go to constop;

     alt:
v = curval;
```

```
beginning of sequence to handle
 compile-time constant calculations;
find constant values for both
 input argumnnts;
apply specified operation to
 argument values;
attempt to find calculated value
 in constants table;
if not available, transfer to
 "new constant" sequence;

flag current code item as constant;
enter value number referencing
 constant in current code item;
loop back to process next code item;

use alternate procedure if
 reserved range of value numbers
 is exhausted;
assign new value number for new
 constant increment reserved
 value number counter;

assign value number and calculated
 constant value to new constant
 value table entry;
increment top pointer of constant
 value table;
loop back to finish treatment
 of current item;

if reserved range of value numbers
 is exhausted, use ordinary rather
 than reserved value number;
```

```
curval = curval + 1;                    increment ordinary value number
                                         counter;
        goto attback;
        simpassign:                     beginning of sequence for
                                         treatment of simple assignment
                                         operation;

variable = in1(item);                   determine variable to which
                                         assignment is made;

quantity = in2(item);                   determine quantity being assigned
                                         to variable;

if(valnum(code(quantity)).eq.0)         if the right-hand side of the
  call newval(quantity);                 assignment operation has not yet
                                         been given a value number, call
                                         a subroutine to assign it a value
                                         number higherto unused;

valnum(code(variable)) =                assign value number of quantity
 valnum(code(quantity));                 as value number of variable;
const(code(variable)) =                 variable also inherits constant
  const(code(quantity));                 bit from quantity;
using valnum(code(variable)),
 set up a modified code text item
 representing a simple assignment
 operation, and add it to the end
 of the modified code text;
go to nextitem;                         loop back to process next code item;

        indxassign:
variable = in1(item);                   determine indexed variable to which
                                         assignment is made;

index = in2(item);                      determine index expression;
quantity = in3(item);                   determine quantity being assigned
                                         to variable;

if(valnum(code(quantity).eq.0)          if the right-hand side of the assign-
  call newval(quantity);                 ment operation has not yet been given
                                         a value number, call a subroutine
                                         to give it a value number hitherto
                                         unused;
```

```
if(valnum(code(index).eq.0)
  call newval(index);
```
and do the same for the index quantity appearing in the assignment statement;

```
valnum(code(variable))= curval;
```
assign new value number to variable;

```
using the array name variable
 and the new value number curval,
 set up a modified code text
 item representing a pseudo-
 assignment of curval to variable,
 and add it to the end of the
 modified code text;
```

```
curval = curval + 1;
```
increment value number counter;

```
opn(avail(aptr))=indexed fetch;
vall(avail(aptr))= variable;
val2(avail(aptr)) = index;
```
construct new available value item representing indexed expression by setting operation field, first and second input value fields;

```
valnum(avail(aptr)) =
  valnum(code(quantity));
```
assign value number of quantity as value number of indexed expression;

```
isconst(avail(aptr)) =
  const(code(quantity));
```
indexed expression also inherits constant bit from quantity;

```
call hashin(aptr);
```
update available calculations hash-table to reference new item;

```
using information from
 appropriate fields of
 avail(aptr), add indexed
 store operation item to
 end of modified code text;
```

```
go to nextitem;
```
loop back to process next code item;

```
        indxload:
```
sequence for handling indexed load operation;

```
il = in1(item); i2=in2(item)
```
determine variable and index of load;

```
if(valnum(code(il).eq.0)
  call newval(il);
```
if the first input argument of the indexed load operation has not yet been assigned a value number, call a subroutine to assign it a value number hitherto unused;

```
if(valnum(code(i2)).eq.0)                carry out corresponding procedure for
  call newval(i2);                        second input argument (index) of
                                          load operation;
v1 = valnum(code(i1));                   determine the value
v2 = valnum(code(i2));                    numbers of variable and index;
else oldval = find(op,v1,v2);            else try to find available prior
                                          value number with identical
                                          operation and inputs;
if(oldval.eq.0) go to notavail;          returned value of 0 means no such
                                          calculation available;
isconst = findc(op,v1,v2);               determine constant attribute from
                                          bit of available item entry found;
const(code(now)) = isconst;              set constant attribute appropriately;
valnum(code(now)) = oldval;              assign value number found;
go to nextitem;                          loop back to process next code item;

        call:                            sequence to handle subroutine calls;
examine all the arguments of
 the subroutine call;
if an argument is a complex
 expression rather than a simple
 variable or an indexed variable,
 go on to examine the next argu-
 ment;
if the argument j under examination
 is an indexed quantity, go to
 indxarg;
else execute valnum(code(j))             assign new value number to
  = curval;                               subroutine argument;
curval = curval + 1;                     increment value number counter;
add pseudo-store into argument
 to end of modified code text;
go on to examine the next
 argument;
```

```
when all arguments have been
 examined, assign a new value
 number to every additional
 indexed or unindexed variable
 whose value the subroutine
 may affect and add corres-
 ponding pseudo-store operations
 to modified code text;
go to nextitem;                    loop back to process next code item;

        indxarg:
set jl = variable occurring in
 the indexed quantity
valnum(code(jl)) = curval;         assign new value number to
                                    subroutine argument;
curval = curval + 1;               increment value number counter;
add pseudo-store into argument
 to end of modified code text;
add copy of code text item to
 end of modified code text;
go on to examine next subroutine
 argument;

        function:
valnum(code(now)) = curval;        assign new value number to
                                    current code item;
curval = curval + 1;               increment value number counter;
const(code(now)) = no;             flag function operation as nonconstant;
go to call;                        go back to treat arguments by
                                    normal subroutine call procedure;

        branch:                    sequence for treating forward branch
if branch is unconditional,         operation;
 put vl = 0 and go to uncbranch;
else il = cond(code(now));         if the branch is conditional,
if valnum(code(il).eq.0)            check to see that the condition
  call newval(il);                  argument has been assigned a value
                                    number; if not, assign it;
vl = valnum(code(il));             determine value number of condition;
```

```
        uncbranch:
using information in fields of
 code(now), together with value
 vl, set up forward branch item
 and add it to end of modified
 code text;
type(lupstak(lbptr)) = branch;          enter a new item at the top of
                                         the loop and branch stack,
                                         flagging it as a (forward) branch;
tlabel(lupstak(lbptr)) =                post the destination label of the
  dlabel(code(now));                     branch, its branch item number, and
itnum(lupstk(lbptr))=                    the number of the last preceding
  itnum(code(now))                       store operation;
stornum(lupstak(lbptr)) =
  stornum(code(now))
firstav(lupstak(lbptr)) = aptr;         post the number of the first item
                                         in the available values array
                                         which lies within the forward branch;

let loc be the location of the
 last modified code text item
 preceding the current branch;
lstcoditm(lupstak(lbptr))= loc;         post the last instruction immediately
lstuzdval(lupstak(lbptr))=curval;        preceding the forward branch and the
                                         current state of the value number
                                         counter;
curval = curval + 100;                  skip 100 value numbers to use for
                                         possible calculations  move to
                                         point preceding branch in later
                                         processing
lstskpdval(lupstak(lbptr))=curval;      post last skipped value number in
                                         forward loop and branch stack item;
lbptr = lbptr + 1;                      increment top pointer for loop
                                         and branch stack;
```

```
need(lupstak(lbptr)) = 0;
live(lupstak(lbptr)) = 1;


go to nextitem;

        loop:

if backward branch is
 unconditional, put vl = 0 and
 go to uncloop;
else il = cond(code(now));
if valnum(code(il).eq.0)
  call newval(il);

vl = valnum(code(il));

        uncloop;
using information in fields of
 code(now), together with value
 vl, set up backward branch item
 and add it to end of modified
 code text;
itmno = targetno(code(now));
qualnum = qualno(code(now));


find the entry on lupstak whose
 item number is itmno and whose
 qualification number is qualnm;
let place = location of this
 entry on lupstak;
lstcode = lstcoditm(lupstak(place));
lstuzd = lstuzdval(lupstak(place));
lstkpd=lstskpdval(lupstak(place));
```

```
note that the branch is not yet
 known to enter any loop, and
 that its terminating label has
 not yet been encountered;
loop back to process next code item;

sequence for treating loop
 (backward branch) operation;


if the backward branch is conditional,
 check to see if the condition
 argument has been assigned a value
 number; if not, assign it;
determine value number of condition;


determine the branch item number
 and the qualification number of
 the lupstak item terminated by
 this backward branch;


determine the last code item
 preceding the loop being terminated,
 as well as the last used value
 number and the last skipped value
 number attaching to this loop;
```

```
firstin=firstent(lupstak(place));   determine the branch item number
                                     of the first branch entering the
                                     loop being terminated or any
                                     loop interlocked with it;
lbpstr = lbptr - 1;                 decrement top pointerof loop stack;
for all j from place+1 to lbptr,
 do all instructions till lupitem;
if(type(lupstak(j)).eq.loop)        branch items on the branch and
 go to isloop;                       loop stack are merely moved to
else lupstak(j-1) = lupstak(j);      fill empty slot;
go to lupitem;

        isloop:                     sequence for treating loop item
                                     on loop and branch stack;
lupstak(j-1) = lupstak(j);          relocate item;
lstcoditm(lupstak(j-1))=lstcode;    redefine all loop items higher on
lstuzdval(lupstak(j-1))=lstuzd;      the branch and loop stack than
lstskpdval(lupstak(j-1))=lstskpd;    the loop item being terminated to
                                     have the same code-motion control
                                     information as the loop item
                                     being terminated;

if((firstin.ne.0).and.
  (firstent(j-1).ne.0)) then
firstent(lupstak(j-1))              redefine the first entering branch
 = min(firstent(lupstak(j-1)),       for each loop item higher on the
   firstin);                         branch and loop stack than the
else if(firstin.ne.0)                loop item being terminated to be
  firstent(lupstak(j-1))=firstin;    the earliest branch entering
                                     either the former or the latter loop;

        lupitem:
continue;
go to nextitem;                     when all relevant loop and branch
                                     stack items have been processed,
                                     transfer back to treat the next
                                     item in the code text;
```

```
        label:          beginning of sequence for treatment
                         of label item appearing in code;
                        (first all operations related to
                         forward branches terminating at
                         this label must be carried out);
itmno = firstbranch(code(now));  determine branch number of first
                         forward branch which terminates
                         at a point from which the current
                         label may be reached by a series
                         of backward branches;
if(itmno.eq.0) go to nobranch;   if no such branch exists, skip
                         first part of procedure;

find branch entry on lupstak
 whose branch item number is itmno;
let place = location of this
 entry on lupstak;
aptr = firstav(lupstak(place));  reset the top pointer for the
                         available values stack to the
                         value associated with the longest
                         branch item terminating at a
                         point from which the current label
                         may be reached by a series of
                         backward branches;
lowstor=stornum(lupstak(place)); determine the number of the first
                         store item within this same branch;
hystor=stornum(code(now));       determine the number of the last
                         store operation preceding the
                         current label;
call newvalno(lowstor,hystor);   call subroutine to assign new
                         value number to every variable
                         referenced by store in forward
                         branch;
remove = 0;                      initialize number of branch items
                         to be removed;
```

```
labl = labname(code(now));
for all j from place to lbptr-1
 do all instructions to britem;
if(kind(lupstak(j)).eq.loop)
  go to shift;
if(tlabel(lupstak(j)).eq.labl)
  go to examine;

firstav(lupstak(j)) = aptr;




        shift:
lupstak(j-remove) = lupstak(j);

go to britem;

        examine:
if(need(lupstak(j)).eq.0)
  go to suppress;

live(lupstak(j)) = 0;

go to britem;


        suppress:
remove = remove + 1;


        britem:  continue;
lbptr = lbptr - remove;
```

```
note name of current label;


loop items on loop and branch
 stack are merely to be moved;
other than first branch items
 terminating at current label are
 to be examined for suppression;
reset first included computation
 reference of all relevant forward
 branch items on loop and branch
 stack to reference current position
 of available computation top pointer;


loop items and unsuppressed branch
 items are moved to new location;
and continue iteration;


if forward branch item does not
 define a possible target for code
 motion, suppress it;
otherwise merely flag it as a
 terminated forward branch item;
go on to process next item on loop
 and branch stack;


removed item counter is incremented
 by 1 for each suppressed branch item;


adjust loop stack top pointer value;
```

```
        nobranch:

lastloop = lastlup(code(now));

if(lastloop.eq.0) go to nextitem;

lowstor = stornum(code(now));

thispt = lastloop;

firstpt=firstlup(code(now));

lupcount = 1;

        repeat:

insert no-op code item at end of
 modified code text, and let loc
 be its location;
this =codeptr(list(thispt));

type(lupstak(lbptr)) = loop;
lstcoditm(lupstak(lbptr))= loc;
lstuzdval(lupstak(lbptr))=curval;

curval = curval + 100;
```

beginning of sequence for setting up loop items representing loops beginning at current label;

determine pointer to last loop (reverse branch) item terminating in the present label;

if there is no such loop, transfer back to process next code item;

determine number of first store operation immediately following label;

initialize pointer to item in loop-list belonging to given label;

set up pointer to first item in loop-list belonging to given label;

initialize loop number count;

head of iterative loop for treatment of successive loops referencing given label;

determine transfer item in code stream corresponding to loop-list entry currently being processed;

build up new loop item at top of loop and branch stack, flagging it as being of type loop, and noting the last instruction immediately preceding the loop and the current state of the value number counter;

skip 100 value numbers to use for possible calculations moved out of loop in later processing;

```
lstskpdval(lupstak(lbptr))=curval;   post last skipped value number
                                      in new loop item;
firstent(lupstak(lbptr)) =           post branch item number of first
 firstent(code(this));                branch entering loop, taking
                                      value from corresponding field
                                      in loop-terminating backward
                                      branch;
if(firstent(code(this)).eq.0)        check for loop entered by
  go to poster;                       forward branch;
else find the entry on the loop
 and branch stack whose branch
 item number is
 firstent(code(this));
let j be the location of this
 item on the loop and branch
 stack;
set need(lupstak(j)) = 1;            flag forward branch item as
                                      defining possible code motion
                                      destination;

         poster:
itnum(lupstak(lbptr)) = itmno;       post branch item number of label
qualnum(lupstak(lbptr))=lupcount;     and loopcount of distinguishing
                                      modifier in new lupstak item;
qualnum(code(this)) = lupcount;      enter loopcount as distinguishing
                                      modifier into loop-closing
                                      backward branch code item;
hystor=stornum(code(this));          determine last store operation from
                                      which direct return to a point
                                      within loop being processed is
                                      possible;
call newvalno(lowstor,hystor);       call subroutine to assign new
                                      value number to every variable
                                      to which assignment is made
                                      within extended loop;
```

```
lupcount = lupcount + 1;          | increment qualification counter;
if(thispt.eq.firstpt)             | check for completion of processing,
  go to nextitem                  |  all loops to given label; on
                                  |  completion transfer back to
                                  |  take up next following code item;
thispt=prev(list(thispt));        | else pass to preceding loop-list
go to repeat;                     |  entry and go to process this
                                  |  entry;
        finish:                   | end of main process of linear
                                  |  nested region optimizer;
                                  | go on to any additional steps
                                  |  of code transformation and/or
                                  |  generation
```

forward branch not terminating at the label being processed or which is the first forward branch entering some loop is moved to preserve the packed condition of the loop and branch stack, and, at the same time, the field in such an item referencing the first computation in the available computations stack included within the forward branch is reset to the value that this field has in the longest branch terminating at the label currently being processed. The necessity for such changes may be seen by consulting Figure 19.

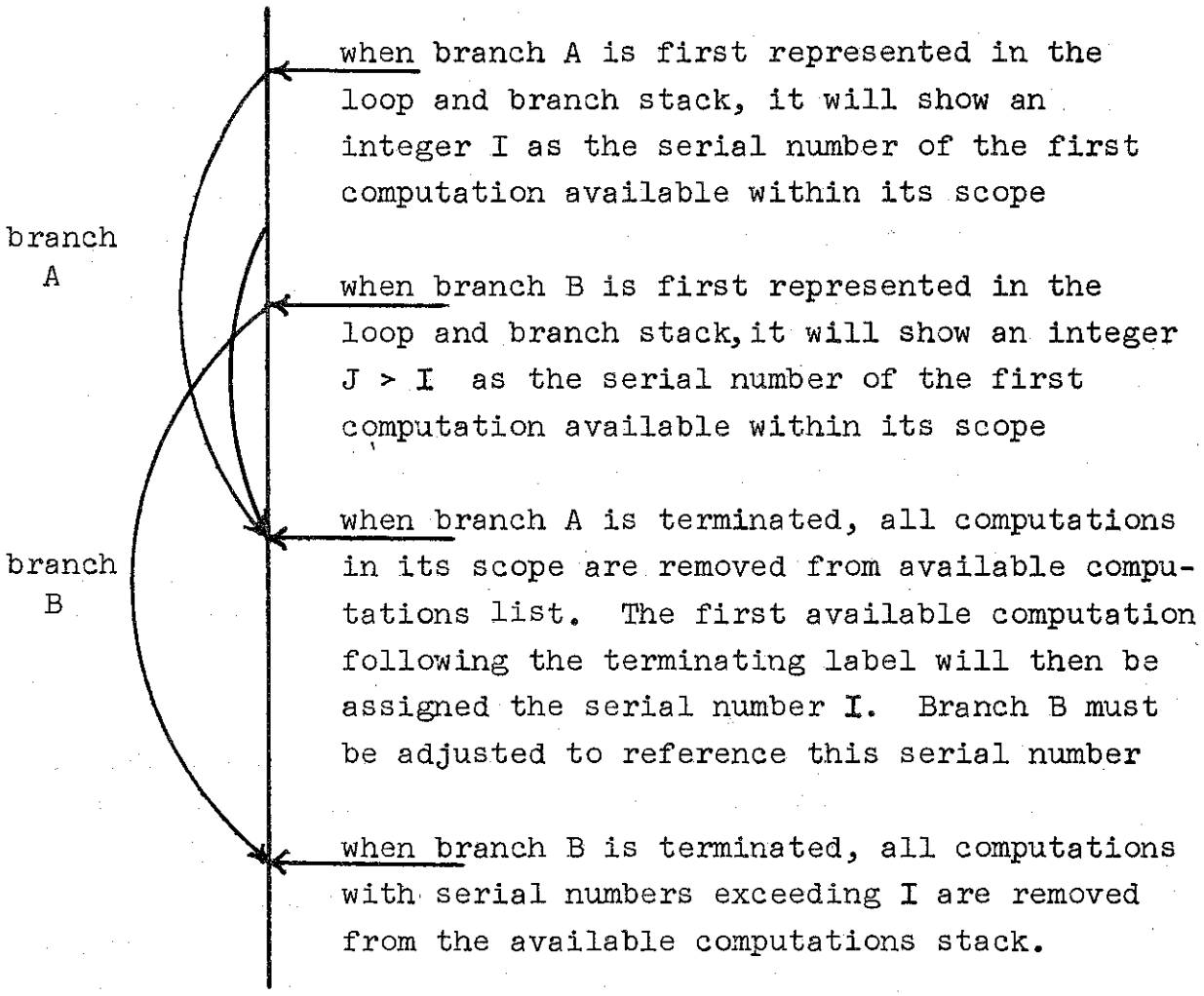


Figure 19. Processing of interlocing forward branches.

In the case in which a loop and branch stack item representing a forward branch is not suppressed on encountering its terminating label, the live bit in the stack item is dropped to indicate that the label has been encountered.

Once all the forward branches terminating at a given label have been processed, a set of new backward branch items, one such item corresponding to each backward branch terminating at the given label, is introduced at the top of the loop and branch stack. Using a counter initialized to one, each of these items is assigned a distinguishing "qualification number". An auxiliary no-operation is introduced into the code string for

each loop and branch stack item introduced; the location of this no-operation is entered in the newly established loop and branch stack item. A range of value numbers is reserved for code to be moved out of the loop being established; and the extent of this range is recorded in the loop and branch stack item being set up. This range is taken to extend from the current value of the value number counter and over a range of some hundred skipped values. The first and the last values of this reserved range are entered into the branch item being established; each time that a calculation is subsequently moved out of the loop represented by the loop and branch stack entry being set up, the first of these fields, representing the first value in the skipped range, will be incremented by one and the corresponding value number assigned to the moved operation (compare Table III, algorithm for subroutine maymove). Note that, by assigning each moved expression a value number consistent with its new location, we ensure that if the moved expression is a subexpression of a larger expression, movement of the larger expression will be handled correctly. This is the reason why reservation of a range of value numbers at the beginning of each loop is appropriate.

If any forward branch enters the newly established loop, the branch item number of the first such branch is posted in the appropriate field of the loop and branch stack entry being established; the necessary value is determined from the corresponding field in the loop-terminating backward branch. The previously posted loop and branch stack item corresponding to this forward branch is flagged by setting its need bit; this will prevent suppression of the stack item when the optimizer encounters its target label.

As a loop-representing loop and branch item is set up, the qualification number of the loop and branch stack entry currently being defined is entered into the associated loop-closing backward branch code text item. This same code text item is then consulted to determine the last store operation in the code sequence from

which return along the corresponding backward branch to the head of the loop being set up is possible, and a subroutine is called to give a new value number to each variable and array to which an assignment is made within this extended loop. This procedure assures proper treatment of interlocking loops (i.e., interlocking backward branches, cf. Figure 15). When every backward branch terminating at a given label has been processed by the steps described, the treatment of the label itself is complete and the algorithm advances to treat the next successive code text item.

We come finally to consider the algorithmic procedure employed by the linear nested region analyzer on encountering a backward branch. Each such backward branch terminates a loop. By consulting the appropriate fields in the code text item to be processed, the branch item number and the qualification number of the loop stack item terminated by the given backward branch are determined. Using these two descriptors, a unique entry on the loop and branch stack with these same descriptors is found. The location L of this entry on the loop stack is noted; in the subsequent steps of processing, all the items on the loop and branch stack which lie above this location will be processed.

Since items are added to the top of the loop and branch stack in the order in which corresponding label and forward branch items are encountered in the code text, every loop and branch stack item lying above a backward branch item B being terminated represents either

a) a forward branch whose origin follows the target label of B and whose terminus follows the origin of B (cf. Fig. 20), or

b) a backward branch defining a loop interlocked with B, i.e., a backward branch whose terminus follows that of B and whose origin follows that of B.

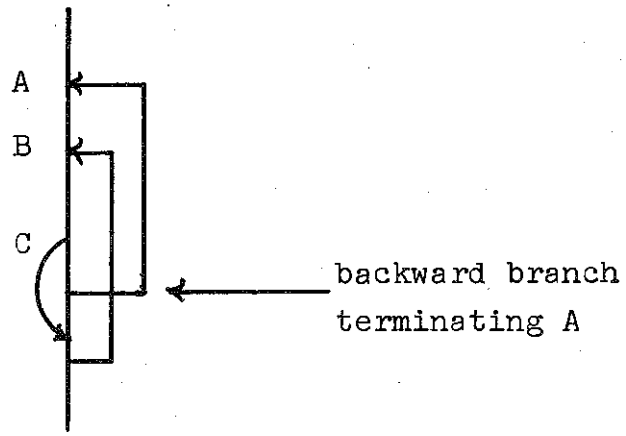


Figure 20. Correspondence between position of items on loop and branch stack and position of branch items in code. Stack entries corresponding to the two loops and the one forward branch shown are made in the order A,B,C. When, at the point indicated, A is terminated, B and C lie above it on the loop and branch stack.

The reader must bear these relationships in mind while considering the backward branch processing sections of the optimization algorithm.

Extracting appropriate fields of the loop stack item at location L, we determine the position of the last code text item preceding the loop being terminated, as well as the last value number attaching to code text items preceding the loop, and the last skipped value number attaching to the loop. In the same way, we determine the branch item number of the first branch entering the loop being terminated or any loop interlocked with it.

At this point, we begin to process the items on the loop and branch stack lying above the item that has been located. At the end of this processing, the loop entry originally at location L will have been deleted from the loop and branch stack, and all items lying above it moved down one place to maintain the loop and branch stack in a densely packed condition. Branch items processed are merely moved down one position in the loop and branch stack. Loop items are treated in a more complex fashion. In the processing of a loop item we begin

by moving it down one position in the loop and branch stack, just as if it were a branch item. However, the last used value and last skipped value fields of each loop item moved are reset to the values which these fields has in the loop item originally at location L. The field in each such loop item referencing the last code text item immediately preceding the loop is similarly reset, and the nominal "first entering branch" field in each such loop item is defined to be the earliest branch entering any loop interlocked with either the loop represented by the item just moved or the loop represented by the branch and loop item originally at location L.

These operations effect a kind of redefinition of the loops interlocked with the loop being terminated, at least insofar as the information used by the maymove subroutine to control code motion are concerned. When our optimizer passes the backward branch terminating a given loop A, the effective scope of any interlocked loop B is redefined so as to give B, as its starting point, that originally belonging to A. The loop and branch stack item representing B also inherits its "last skipped value" field from A; thus, no code item occurring after the terminating branch of A will be moved out of B unless none of its input arguments are redefined in the combined scope of A and B, allowing the item to be moved to the last code position preceding A. Figure 21 shows the successive "structural views" (determined by the set of variables which are assigned new value numbers and the items present on the loop and branch stack at any given time) which the optimizer applies to an interlocked pair of loops in the course of processing the section of code text having the flow structure shown in the figure.

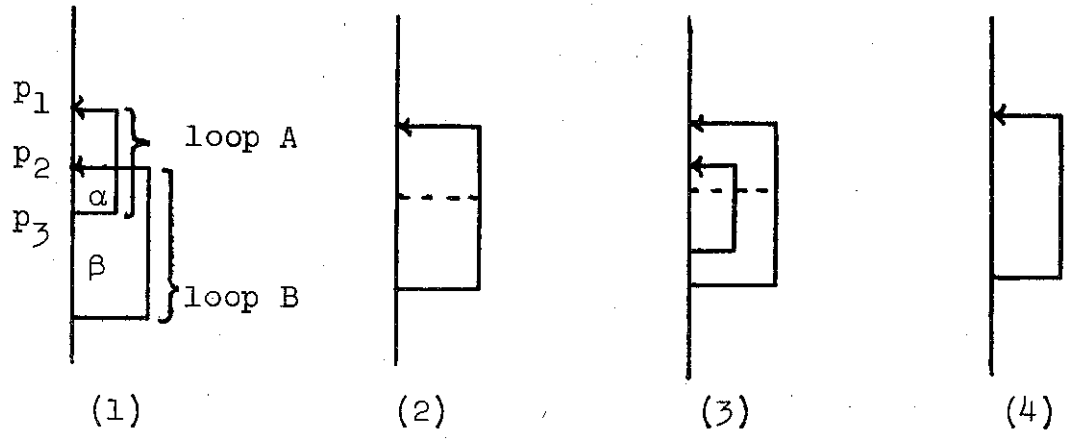


Figure 21. Processing of interlocked loops.
(1) Actual pair of interlocked loops.
(2) Nominal loop after label p_1 is processed.
(3) Nominal pair of loops after label p_2 is processed.
(4) Nominally remaining loop after backward branch p_3 is processed.

The effect of the procedures which we have set forth is to allow an instruction occurring at the position α shown in Figure 21 to be moved either to the first position immediately preceding p_2 or to the first position immediately preceding p_1, while an instruction occurring at the position marked β in Figure 21 will only be moved if it can correctly be moved all the way to the first position immediately preceding p_1. Note, in particular, that we will never move an instruction occurring at a position such as β from the place outside a loop such as A to a place inside the loop A. Note that, if loop A is executed more frequently than loop B, such motion would be undesirable. Motion of code from the range β into the loop A might also violate safety constraints: calculating a * b before a reached its "final" value (value on exit from A) might generate unwanted interrupts.

Figure 22 shows a set of backward branches interlocked in a more complex fashion.

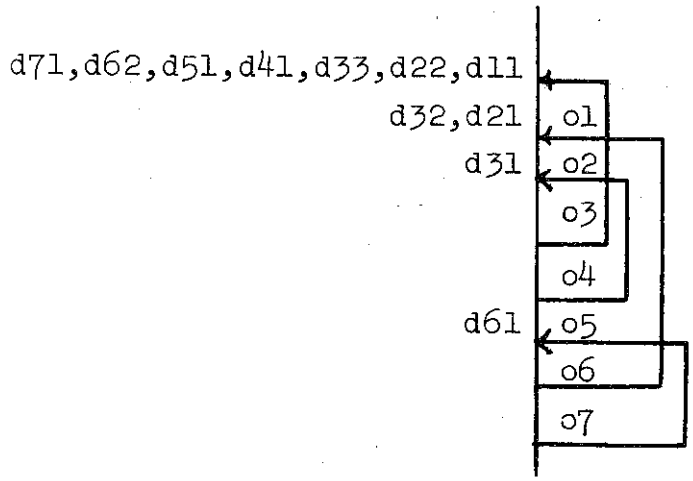


Figure 22. Complex interlocked set of backward branches showing points at which movable instructions may be placed.

Note in Figure 22 that a movable instruction originating in the range marked o1 may be moved to the point d11, that a movable instruction originating in the range marked o2 may be moved either to the destination d21 or to the destination d22, that a movable instruction originating in the range o4 may only be moved to the destination d41, etc.

We observe also that, in processing a backward branch, a slightly changed copy of the branch item, in which the condition governing a conditional branch is represented by a value number, is added to the modified code text.

This completes our description of the principal optimizer process described by Table II. After processing a code by use of this algorithm, one goes on of course to any additional steps of code transformation and/or generation which are required.

As already noted, the main subroutine used by the algorithm shown in Table II is the subroutine maymove. An algorithmic description of this subroutine is given in Table III below.

The maymove algorithm processes items on the loop and branch stack beginning with the topmost item and proceeding toward the bottom of the stack. These items represent: (a) backward branches whose target labels have already been encountered, but which have themselves not been encountered; (b) "unterminated" forward branches, i.e., forward branches whose target labels

have not yet been encountered; (c) "terminated" forward branches, i.e., forward branches whose target labels have been encountered.

Note that it is the responsibility of maymove to find the largest loop, as defined by items (a), out of which one can move the calculation α which it is given. As noted in the discussion of Figure 8 above, each item of type (b) represents a barrier to code motion. We never move a "conditional" instruction α, i.e., an instruction α bypassed by a forward branch, into the "unconditional" range preceding the forward branch. Moreover, and again as explained in connection with Figure 8, we never move an instruction α lying within a loop entered by a forward branch out of the loop unless the instruction follows the terminating label of the forward branch. Even in this case, α is only moved if it can be moved to a point preceding the occurrence of the forward branch. The maymove algorithm shown in Table III conforms to this requirement as follows. The entries on the loop and branch stack are successively examined, starting at the top of the stack, that is, starting with the last occurring (innermost) loop or forward branch, and working down into the stack. During this process, two pointers to loop and branch stack locations, and one branch item number, are constantly maintained. The first of these is a pointer j defining the stack location currently under examination, and hence defining a point to which α could be moved if not for the special restrictions applying to loops entered by forward branches. Two remaining fields, called valid and mustmove in the algorithm of Table III, play an equally important role. The first of these is a pointer defining the earliest code location to which maymove is, at any given moment, sure that α can be moved, even taking all restrictions concerning loops entered by forward branches into account; the second is a branch item number defining the last code position at which α can be placed if it is to be moved beyond the point currently specified by the valid pointer and in a manner consistent with all forward branches encountered by maymove up to

a given moment in its operation. Consider, for example, the action of maymove when called to process the instruction α shown in the following figure, 23.

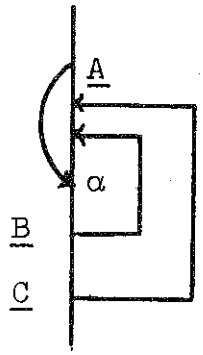


Figure 23. Moving an instruction out of a loop entered by a forward branch.

When maymove is called to process α, it first attempts to move it out of the loop B. If, ignoring the entering forward branch A, it finds this motion to be possible, the j pointer is advanced to reference the loop C, and a reference to the entering forward branch A (actually, the branch item number of A) is entered into mustmove. If maymove then finds that α cannot be moved out of the loop C, it will also be the case that no fully validated move has been entered into valid. If, on the other hand, α can move out of the loop C, it also follows that α can be moved back past the forward branch A. On discovering this fact, maymove will make a first non-zero entry into valid, and, then go on to try to find still larger loops out of which α can be moved. Even if motion out of larger loops turns out to be impossible, valid will still contain a reference to the first code location preceding the forward branch A, so that the earliest code position to which α can be moved will still have been successfully recorded.

The detailed structure of the algorithm of Table III is as follows. The algorithm first checks that the loop and branch stack is not empty. In the contrary case, code motion is impossible and a negative return from maymove is made immediately.

If the loop and branch stack is not empty, the valid and mustmove pointers whose significance is explained above are both initialized to zero. The algorithm then successively examines all items on the loop and branch stack starting at the top of the stack. An item is selected, its type (forward or backward) determined, and transfer to the separate processes appropriate in these two cases is made. Each "live" forward branch item, that is, forward branch whose terminal label has not yet been encountered, constitutes a block to code motion. Occurrence on the loop and branch stack of an item representing such a forward branch causes the algorithm to transfer to its pre-return procedure. Items representing "terminated" forward branches are only examined if the mustmove pointer is non-zero, i.e., only if a backward loop entered by a forward branch has previously been encountered, and a corresponding restriction on the placement of α noted. While this is the case, the item number of each terminated-forward-branch entry on the loop and branch stack is examined, if this item number is seen to represent a branch item lying previous to the item defined by the mustmove field, and if a check of value numbers shows that no argument of the code item to be moved is defined within the scope of the forward branch, the location of the forward branch item in question is recorded in the valid pointer, the mustmove pointer is reset to zero, indicating that no unsatisfied entering forward branch constraint currently remains, and the algorithm goes on to process additional loop and branch stack items, representing larger loops out of which it may be possible to move α.

Table III. Algorithm for the maymove subroutine

Subroutine maymove(maxinval,yesno,place,valnumber)

```
if(lbptr.eq.1) go to not;          code motion not relevant if not
mustmove = 0;                       within loop; initialize latest point
                                    at which code item can be placed;
valid = 0;                         initialize valid code motion
                                    destination;
for all j = 1 to lbptr-1,          look successively at all items in
  do all instructions till          the loop and branch stack,
  endloop;                          starting at the top of the stack;
entry = loopstack(lbptr-j);
if(type(entry).eq.loop)            separate forward and backward
  go to backward;                   branch cases;
if(live(entry).eq.yes)             an unterminated forward branch
  go to blocked;                    constitutes a block to code motion;
if(mustmove.eq.0) go to endloop;   if no entering forward branch is
                                    currently posted, ignore
                                    terminated forward branch items;
if((maxinval.ge.                   if an input argument of the calcula-
 lstskpdval(entry)) go to blocked;  tion to be moved is defined within
                                    a forward branch, motion past the
                                    branch is impossible;
if(itnum(entry).gt.mustmove)       if terminated forward branch follows
  go to endloop;                    latest point at which code item
                                    can be placed, it may be ignored;
else valid = j;                    else post start of forward branch as
                                    earliest possible code destination;
mustmove = 0;                      drop entering forward branch flag;
go to endloop;                     go on to process additional loop
                                    and branch stack items;
       blocked:                    when motion of code to earlier
                                    point is impossible;
if(valid.ne.0) go to move;         if valid code destination has been
                                    found, go to set up code motion;
```

```
        not:
yesno = no;  return;
```
negative return from subroutine;

```
        backward:
if((maxinval.ge.
 lstskpdval(entry))goto blocked;
```
procedure for backward branch item;
if an input argument of the calculation to be moved is defined within a loop, motion out of the loop is impossible;

```
if(firstent(entry).eq.0)
  go to notentered;
else mustmove =
 min(mustmove,firstent(entry));

go to endloop;
```
go to separate procedure for loop not entered by forward branch;
else if code is to be validly moved, it must move to point preceding all entering forward branches;
proceed to next loop and branch stack item;

```
        notentered:
valid = j;

mustmove = 0;
```
post backward branch as defining earliest possible code destination;
drop entering forward branch flag;

```
        endloop:  continue;
valid = lbptr;
```
fallout case: set variable j to reference loopstack bottom in subsequent procedure;

```
        move:
```
procedure to set up code motion descriptors:

```
entry=lupstack(lbptr-valid+1);
```
post lupstack entry for point to which motion is indicated;

```
valnumber=lstuzdval(entry)+1;
```
determine value number assigned to moved calculation;

```
if(valnumber.ge.
 lstskpdval(entry)) go to over;
```
check for conflict with value numbers assigned following destination point of motion;

```
lstuzdval(lupstak(lbptr-valid+1))
 = valnumber;
```
else update record of last value number used at destination point of motion;

```
        insert:                          beginning of procedure to insert
                                          code item at moved position;
location = lstcoditm(entry);             find location of last modified
                                          code text item preceding destina-
                                          tion point of motion;
insert blank code item
 immediately following code
 text position defined by
 location;
let place = position of this
 blank code text item;
lstcoditm(lupstak(lbptr-j+1))            update record of position of last
  = place;                                code item at destination point
                                          of motion;
yesno = yes;  return;                    positive return from subroutine;
        over:                            special procedure in value
                                          number overflow case;
valnumber = curval;                      use normal value number counter
curval = curval + 1;                      and increment it;
go to insert;                            loop back to use prior code
                                          insertion procedure;
```

[end of algorithm]

Backward branch items occurring on the loop and branch stack are treated as follows. If the code item α is found to be redefined within the loop described by a given backward branch item on the loop and branch stack, the algorithm transfers at once to its pre-return procedure. In the contrary case, the "first entering branch" field of the backward branch item is examined, and transfer is made to one of two slightly different subprocesses, depending on whether or not this field is zero, i.e., on whether or not the loop in question is entered by a forward branch. If the loop is entered by a forward branch,

the mustmove field is reset to the minimum of its former value and the branch item number of the first forward branch entering the current loop; this reflects the fact that if the code item α is to be validly moved, it must be moved to a point preceding all forward branches entering loops out of which it is moved. Once the mustmove value has been redefined the algorithm loops back to process the next loop and branch stack item in turn. If, on the other hand, the loop item being processed has not been entered by a forward branch, a reference to this loop item is entered into the valid pointer, indicating that the maymove algorithm is now certain that the instruction α can be moved in the code at least as far as the label defining this loop, and the mustmove field is reset to zero, indciating that no currently unsatisfied entering forward branch constraints remain. In this case also maymove loops back to process the next loop and branch stack item in turn.

The terminal procedure employed by maymove is as follows. If the valid field is found on termination to be set to zero, a negative return is made. In the contrary case, the loop and branch stack entry referenced by the valid pointer is found, the value number to be assigned to α at its new location is determined from the last used value field of this loop and branch stack entry, and the last used value field updated to reflect the fact that one additional instruction has been moved. A blank code item is inserted immediately after the last code item preceding the branch item represented by the loop and branch stack entry, and the subroutine maymove makes a positive return. A few details of the action of the maymove subroutine omitted in the above summary description will be found in the detailed algorithm shown in Table III.

Table IV below describes the first pass of the linear nested region optimizer. It is the responsibility of this pass to fill in those fields of code text transfer items (cf. Figure 12 above) which refer to aspects of the code determined by items lying

ahead of a given item, and hence which cannot easily be set up by the compiler itself. Such information is, of course, required in the algorithms of Tables II and III. The procedure described in Table IV uses a current item pointer, called now, as well as two counters called actr and bctr, which respectively count assignments and branch items. At the beginning of the algorithm, these counters are initialized, and the current item pointer is set to point to the first item of the code text. Then the algorithm enters a main processing loop, which inspects every code item in turn, bypasses code items representing ordinary computations, and, using an indexed transfer determined by an auxiliary array called kinda, proceeds to separate sub-processes for the treatment of assignment, subroutine call, and branch items. To process an assignment item, one merely adds, to an array reserved for this purpose, and at the location referenced by an assignment counter, a reference to the target variable of the assignment. The assignment counter is then incremented, and the algorithm loops back to process the next code item in turn. In this way, one builds up the complete list of assignment operations required by the algorithm in Table II. Subroutine and function calls are handled in much the same way as assignments. Every argument of such a call, if it is a simple or indexed variable rather than a complex expression and may therefore be an output argument of the subroutine, is added to the assignment list, and the assignment counter correspondingly incremented. All "common" variables accessible to and hence modifiable by the subroutine must also be added to the assignment list. Once this is done, the algorithm loops back to process the next code item in turn.

The treatment of branch items, and especially backward branches, is considerably more complex. The branch item processing sub-section of the first forward pass must attain several goals. Each branch item must be assigned a branch item number. With each label, we must associate a list of all backward

branches terminating at the given label; this list is required by the algorithm of Table II. Moreover, we must flag each backward branch on this list with its serial position on the list. With each label, we must also associate the branch item number of the earliest forward branch, preceding the given label, which terminates either at the given label or in the scope of a linked set of backward branches terminating at the label (cf. Figure 24). Our final goal is to associate, with each backward branch, the number of the last assignment operation from which a label within the scope of the backward branch can be reached without passing through any code preceding the target label of the backward branch; this information is also required by the algorithm of Table II. We attain our aims as follows. With each label we associate a store number, similar to that to be associated with backward branches, and called lastore in Table IV; namely, the number of the last assignment operation from which the label in question can be reached without passing through any of the code preceding it. In order to do this, we must associate, with each backward branch, a list of all the labels which can be reached via the branch and via linking backward branches, in each case without passing through code preceding such a label. This list is called the backtarget list of the given backward branch, and may be built up iteratively for any given backward branch β by adding, to the direct target of the backward branch, the backtarget lists of all linking backward branches lying within the scope of β. (Cf. Figure 24.)

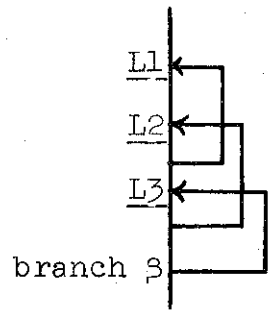


Figure 24. Linked backward branches and backtarget lists. Labels L1, L2, and L3 all belong to the backtarget list of branch β; any assignment operation preceding β is relevant for all these labels.

If a label L occurs on the backtarget list of a given backward branch β, we set lastore field of the code item representing L to reference the last store included in the scope of β. Carrying this operation out sequentially through the whole code insures that each label L will ultimately reference the last assignment operation, in order, from which L can be reached without passing through any code preceding L. Once this is done, we reprocess all backward branches, entering into the stornum field of each backward branch the maximum of the lastore field taken over all labels lying in the scope of the backward branch.

The algorithmic details of branch item processing during the first pass of the linear nested region optimizer are as follows. The stornum field of the current code item is set to the present value of the assignment counter; thus each branch item initially references the first assignment operation preceding it. As explained, this field will, in the case of backward branch items, ultimately be changed. The current value of the branch item counter is entered into the appropriate field of the current code item, and is also saved. A reference to the current code item is placed on a comprehensive list of branch items; this list is used during some subsequent processing phases to provide rapid reference to branch items. The branch item counter is then incremented, concluding the few algorithmic steps common to the processing of all branch items. At this point, indexed transfer is made, using an auxiliary data array kindb, to separate procedures appropriate for the treatment of the three different kinds of branch items. Label items require no further processing at this point, since all additional processing of label items is associated with backward branches which address them. Forward branch items are processed in a rather simple way. The target label of a forward branch is determined, and, if no earlier forward branch has the same target label, the given branch is the earliest branch terminating at its target label. In this case, a reference to the forward branch is posted in the target label of the branch. In either case, the algorithm loops back to process the next code item in turn.

Table IV. Algorithm for first pass of linear nested region optimizer.

```
now = dummy leading no-op in
  code text;                          initialize current item pointer;
actr = 1;
bctr = 1;                             initialize assignment counter
                                       and branch item counter;
        nextitem:
now = nextit(now);                    advance to next code item;
if(now.eq.0) go to loopfill;          check for completion of first phase;
item=code(now);op=op(item);           determine operation code of
                                       item to be processed;
goby kinda(op) (ordinary,             take appropriate transfer to
  assignment, branch, call)            processes for treatment of
                                       assignment, subroutine call,
                                       and branch items;
        ordinary:
go to nextitem;                       bypass all items not of assignment,
                                       call, or branch type;
        assignment:
astack(actr) = inl(now);              add target variable of assignment
                                       to assignment list;
actr = actr + 1;                      increment assignment counter;
go to nextitem;                       loop back to process next code item;
        call:
examine all the arguments of
 the subroutine call;
if an argument is a complex
 expression rather than a
 simple or indexed variable
 go on to examine the next
 argument;
```

```
else add a reference to the
 argument under examination to
 the assignment stack, increment
 the assignment counter, and go
 on to examine the next argument;
when all arguments have been
 examined, add to the assignment
 stack a reference to each indexed
 or unindexed variable whose value
 the subroutine may affect,
 incrementing the assignment
 counter as often as necessary;
go to nextitem;                      loop back to process next code item;

        branch:                      common initial procedure for
                                      three types of branch items;
stornum(code(now)) = actr;           post current state of assignment
                                      counter;
thisnum = bctr;                      record and post current branch
itnum(code(now)) = bctr;              item number;
branchlist(bctr) = now;              place reference to current code
                                      item on branch item list;
bctr = bctr + 1;                     increment branch item counter;
goby kindb(op) (forward,             transfer to procedure appropriate
  backward, label);                   for treatment of particular
                                      branch item;

        forward:                     forward branch procedure;
dest = dlabel(item);                 determine target label of branch;
if(firstbranch(code(dest)).eq.0)     if no earlier branch has same
 firstbranch(code(dest))=thisnum;     target label, this is earliest
                                      terminating at target label;
go to nextitem;                      loop back to process next
                                      code item;
```

```
        label:
go to nextitem;                          all additional processing of label
                                          items is associated with backward
                                          branches which address them;

        backward:                        beginning of sequence for
                                          processing backward branch
                                          code items;
dest = dlabel(item);                     determine target label of branch;
target=itnum(code(dest));                save and post item number of
targetno(code(now)) = target;             target label in current code item;
oldlast = lastloop(code(dest));          determine last preceding branch
                                          with same target label;

add this backward branch to the
 end of the list of backward
 branches targeting at and
 referenced by the target label;
if(oldlast.eq.0) go to firstloop;        test if this is first backward
                                          branch with given target label;
qualif = qualnum(code(codeptr            else determine qualification
 (list(oldlast))));                       number of last preceding backward
                                          branch to given label;
qualnum(code(now)) = qualif+ 1;          present item has next successive
                                          qualification number;
go to linking;                           and skip to begin processing of
                                          loops linked by current
                                          backward branch;

        firstloop:                       special sequence for processing
                                          first backward branch to given
                                          label;
qualnum(code(now)) = 1;                  present item has qualification
                                          number 1;
```

```
        linking:                          begin processing loops linked
                                           by current backward branch;
first = firstbranch(code(dest));          initialize calculation of first
                                           forward branch either targeting
                                           at termination label or entering
                                           scope of backward branch being
                                           processed;
initialize backtarget list of
 code(now) to consist of
 code(dest) as its single member;
for each j from target+1 to               iterative loop to find first
 thisnum-1, do all instructions            forward branch entering scope of
 thru linked;                              backward branch being processed,
                                           and to carry this information
                                           and store information back to
                                           all labels on backtarget list;
branchitem = code(branchlist(j));         examine branch item in scope of
                                           backward branch;
if(kindb(op(branchitem).eq.               bypass forward branch items;
  forward)) go to linked;
if(kindb(op(branchitem)).eq.              included backward branches
   backward)) go to backloop;              require different processing;
else if((first.ne.0)and                   in case of included label,
  (firstbranch(branchitem)).ne.0)          redetermine earliest forward
first = min(first,firstbranch              branch entering scope of
  (branchitem));                           backward branch being processed;
else first=firstbranch(branchitem);
go to linked;                             and go to next item in scope;
        backloop:
if(itnum(dlabel(branchitem)).ge.          if an imbedded backward branch
  target) go to linked;                    links the backward branch being
else add all elements of                   processed, the backtarget list
 backtarget list of branchitem             elements of the former must be
 to backtarget list of code(now);          be added to the backtarget list
                                             of the latter;
```

```
        linked:  continue;          end of iterative processing
                                     of backward branch scope;
lastlab = lastreach(code(now));     initialization for additional
labloc = firstreach(code(now));      processing of linked loops;
                                     post first and last elements
                                     of backtarget list for current
                                     code item;
        labels:                     beginning of loop to reset
                                     lastore and firstbranch fields
                                     of all linked loops;
label = codeptr(list(labloc));      calculate code item reference
                                     of current backtarget list item;
lastore(code(label)) = actr;        current assignment affects all
                                     loops linked with current loop;
firstb = firstbranch(code(label));  if label is not target of forward
if(firstb.eq.0) go to noearly;       branch, use special procedure;
else if (first.ne.0)                else first entering branch number
  firstbranch(code(label)) =         is minimum branch number targeting
  min(first,firstb);                 at label or entering any loop
                                     linked to label;
go to checkloop;                    and skip over initial branch
                                     procedure;
        noearly:                    procedure for loop which is not
                                     forward branch target;
if(first.lt.itnum(code(label))      if entering branch originates
 firstbranch(code(label))=first      before target label, post it
                                     as entering forward branch;
        checkloop:
if(labloc.eq.lastlab)go to          check for terminating of processing
  nextitem;                          of all loops linking current loop,
                                     in which case advance to process
                                     next code item;
labloc = nex(labloc);               else advance to next linked loop
go to labels;                        and continue linked-loop process;
```

```
        loopfill:

                                    beginning of pass over all branch
                                     items to set lastore and firstent
                                     fields of backward branch items;
for all j from 1 to bctr-1,         all branch items in code are
 do all instructions till            processed;
 loopsfull;
branchitem =code(branchlist(j));    extract j-th branch item;
if((kindb(op(branchitem)).eq.       forward branches and labels are
 forward).or.(kindb(op(              bypassed;
 branchitem)).eq.label))
 go to loopsfull;
else start=targetno(branchitem);    all labels in scope of backward
final = itnum(branchitem);           branch must be processed; iterative
firstin = 0;                         loop is correspondingly initialized;
lastasin =stornum(branchitem);      post number of assignment immediately
                                     following backward branch;
for all k from start+1 to           iterative loop to find all labels
 itnum-1, do all instructions        in scope of backward branch;
 till labs;
imbitem = code(branchlist(k));      post imeedded branch item;
if(kindb(op(imbitem)).eq.           only labels are to be processed
 forward.or.kindb(op(imbitem))       within interior iteration;
 .eq.backward) go to labs;
lastasin=max(lastasin,              last store applicable to loop is
  lastore(imbitem));                 last store applicable to any
                                     label in its scope;
if((firstbranch(imbitem).eq.0)      ignore labels which are not targets
 .or.(firstbranch(imbitem)           of forward branches originating
 .gt.start)) go to labs;             outside loop;
if(firstin.ne.0)firstin =           if label is a forward branch target,
 min(firstin,firstbranch             redetermine first entering forward
 (imbitem));                         branch either as unique entering
else firstin=firstbranch(imbitem);   forward branch or entering forward
                                     branch with smallest branch item
                                     number;
```

```
        labs: continue;
stornum(code(branchlist(j))) =
  lastasin;                          post number of last relevant
                                      assignment operation in
                                      backward branch code item;
firstent(code(branchlist(j)))
      = firstin;                     first entering branch applicable
                                      to loop is first branch reaching
                                      any label in its scope;

        loopsfull: continue;
end of first forward pass of
 linear nested region optimizer;
go on to main forward pass of
 optimizer;
```

[end of algorithm]

The algorithmic procedure for the processing of backward branch items bears most of the burden of "lookahead" involved in the first pass of the optimizer and is thus considerably more complicated than the procedures required for the treatment of other kinds of branch items. On encountering a backward branch code item β, one first determines the target label of the branch, posts the item number of this label in the current code item, and also saves the item number for later use. The last preceding branch with the same target label L is determined, β is added to the end of the list of backward branches referenced by L, and a test is made to determine whether or not β is the first backward branch with the target label L. This allows determination of the qualification number of the current backward branch either as 1 or as 1 + the qualification number of the last preceding backward branch having the target label L. The algorithm now finds all loops linked by the backward branch β, developing the backtarget list of β, and determining the first forward branch entering the scope of β. The necessary procedure has the form of an iteration, and is as follows. Calculation of the first forward branch entering the scope of the backward branch being processed is accomplished by a simple minimization carried out

over the first forward branch references of all labels included in the scope of β. For each backward branch α belonging to the scope of β and having a target label with a smaller item number than the target label of β, all backtarget list elements of α must be added to the backtarget list of β.

Once the backtarget list of β has been built up in the manner indicated, the lastore field of every label on the backtarget list of β is reset to reference the last assignment lying within the scope of the current backward branch. At the same time, the "first forward branch" reference attached to each label on the target list of the backward branch β is set to reference either the minimum of its former value and the number of the first forward branch entering the scope of β, or, if the label in question has not previously been recognized as the target of a forward branch, is set to reference the first forward branch entering the scope of the current backward branch, provided that this forward branch precedes the label in question. When all the labels on the backtarget list of a backward branch have been processed in this way, processing of the backward branch is complete and the algorithm loops back to process the next code item in turn.

Once a forward pass over all code items has been completed, a termination procedure affecting backward branch items only is carried out. All labels lying in the scope of each such backward branch β are examined, and the maximum of the lastore field for all these labels is calculated. This maximum quantity, representing the last assignment operation in the code from which one may reach a point in the scope of β without ever entering code preceding the target label of β, is then inserted into the stornum field associated with the backward branch.

When all backward branches have been processed in this way, the first forward pass of the linear nested region optimizer ends, and the optimization process flows on to begin the main optimization pass described in Table II.

6.4 Operator Strength Reduction in the LNRA algorithm.

The "value number" based algorithm described by Table II of Section 3, with which we have been concerned almost exclusively in that section, accomplishes the elimination of redundant expressions, the propagation of constants, and the motion of code out of loops. We shall now begin to consider the extensions of that algorithm necessary to incorporate the important reduction in strength optimization described briefly in Section 1 (cf. subheading (5) of Section 1) and more fully in the preceding section. Section 9 below gives a general analysis of this type of optimization, and discusses the implementation of strength reduction in connection with more powerful optimizers than that considered in the present section. Here we shall discuss the reduction in strength optimization only in a simpler form, but in a form which is nevertheless sufficient to cover the commonest and most important cases in which this optimization is of benefit. This common case is that in which a product formed with an index quantity under control of a DO iteration is reduced, thereby replacing an integer multiplication by an incrementation operation. Figure 25 below shows the typical features of code associated with such a situation.

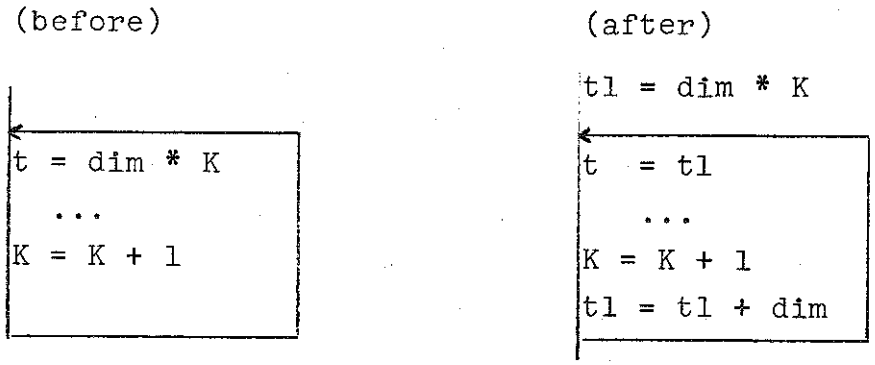


Figure 25. Common case of reduction in strength.

In code not generated in so very simple a way by an iterative control statement in a source language, more elaborate situations which may be treated by similar but more complex strength

reduction processes will occur. Figure 26 illustrates such a case.

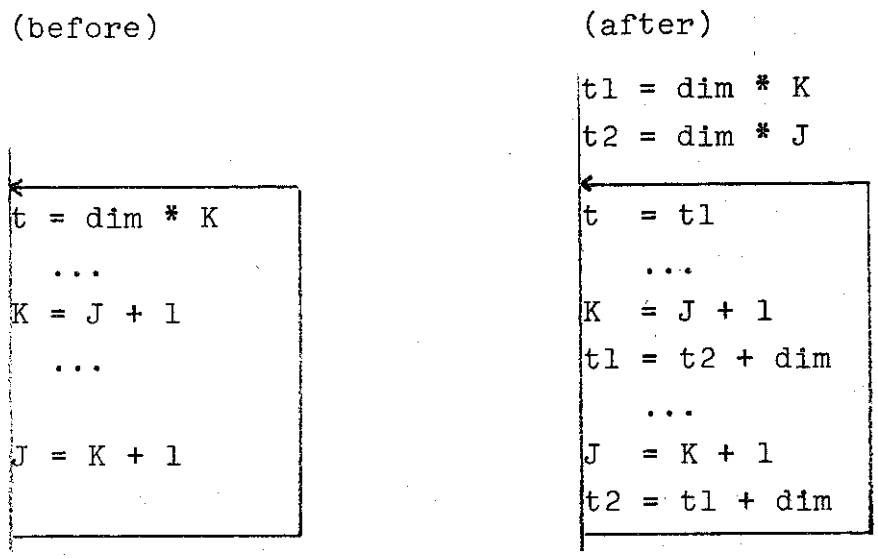


Figure 26. A more elaborate case of reduction in strength.

Discussion of these more complex cases is reserved for Section 9.

In the present section, it is convenient for us to regard strength reduction as a generalization of code motion. That is, in the situation shown in Figure 25, we regard the essence of the strength reduction optimization as being the motion of the multiplication operation dim * K out of the loop shown in that figure; strength reduction is then distinguished from simple code motion by the fact that the expression E being moved actually involves variables redefined (though only in a relatively simple iterative fashion) within the loop out of which E is moved. The existence within a loop of iterative redefinitions of variables occurring within moved expressions makes the process of code motion involved in strength reduction more complicated than the simple code motion process which we have already examined. Reductive code motion requires us to modify the code left within the loop by the insertion of compensating increment opeations appropriately attached to every iterative redefinition of a variable occurring in an expression reduced in strength.

We will, in accordance with constraints described more fully in the preceding section, only apply the strength reduction

process to integer expressions, and more specifically to polynomials in integer variables; in order to guarantee the applicability of strength reduction we shall then impose the condition that all variables occurring in the expressions to be reduced are either loop-constants or are defined only iteratively within the loop to which the reduction process is to be applied. Within the context delimited by this general condition, the simple case of strength reduction shown in Figure 25 is distinguished from the more complex case shown in Figure 26 by the fact that the redefinition affecting the iteratively defined variable K has the form K = K + RC (where here and in what follows we denote a loop constant by the symbol RC), and hence redefines K in terms of its own former value and not in terms of the value of any other variable. Since this restriction makes possible substantial simplification in our strength reduction algorithms, we shall in the present section only attempt to apply strength reduction to situations in which this condition is satisfied. That is, we shall regard an integer variable redefined within a loop as being an iterative variable within the loop if and only if all its redefinitions have either the form K = K + RC or the simpler form K = RC. The candidate expressions considered for strength reduction will be polynomial expressions in integer variables I each of which is either a loop constant or an iteratively defined variable.

Strength reduction applies not only to polynomials linear in iteratively redefined variables, but also to polynomials of higher order. Thus, for example, we may reduce the computation of the fourth power of an iteratively redefined variable to a set of additions. Figure 27 below shows the result of such a reduction.

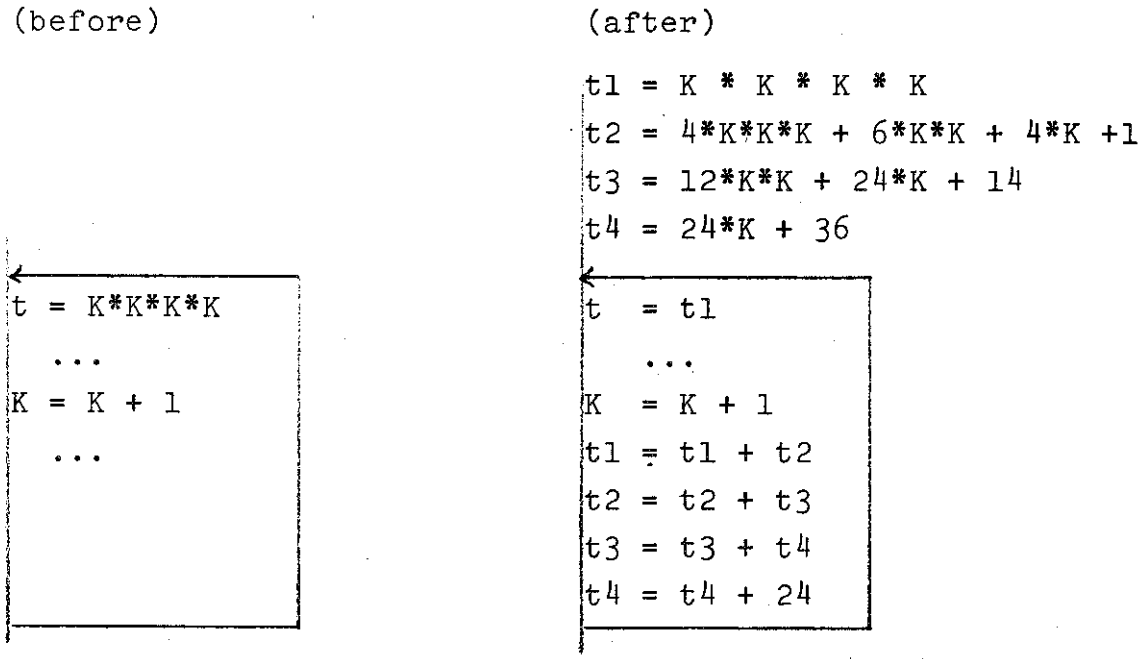


Figure 27. Strength reduction applied to a polynomial in an iterative variable.

Note that the three multiplications occurring in the unreduced code are replaced by four additions in the reduced version; of course, we count only operations occurring within the loop L shown in Figure 27. On the other hand, if L contained more than a single iterative redefinition of the variable K, we would have to attach an addition to each such iterative redefinition of K, one addition being necessary for each power of K occurring in the expression being reduced. If the number of points within L at which the iterative variable K is incremented exceeds the ratio of multiplication time to addition time, strength reduction of expressions involving K will be undesirable, as an excessive number of additions will have to be performed for every multiplication which is avoided.

Reduction in strength also applies to those somewhat more esoteric cases in which polynomial expressions involving more than one iteratively redefined variable occur. Figure 28 shows such a case.

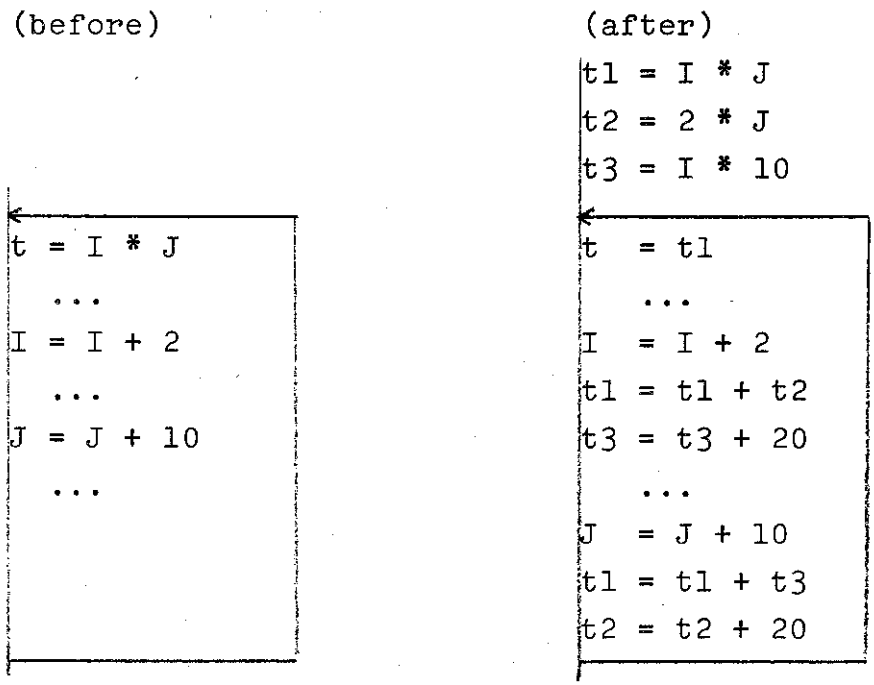


Figure 28. Strength reduction applied to a polynomial in two iterative variables.

In this case, one multiplication has been replaced by four additions. The number of additions replacing a single multiplication when a given monomial is reduced in strength may be estimated as the product of the number of distinct iterative redefinitions affecting each of the separate variables occurring in the monomial. As soon as this product exceeds a limit characteristic of the particular computer for which code is being optimized, strength reduction of a monomial becomes undesirable and ought to be cut off. A similar criterion applying to general polynomials is stated below.

Reduction in strength of polynomials occurring within the innermost loop of a nest of loops will move instructions from within this loop to a point within an outer loop. If optimal code is to be produced, these instructions, once moved, ought to be subject to the normal process of redundant calculation elimination, constant propagation, and to any additional code motion and/or outer-loop strength reductions that apply.

Figure 29 below shows the improvement that can be attained by applying outer-loop optimization processes to code moved in the course of strength reduction.

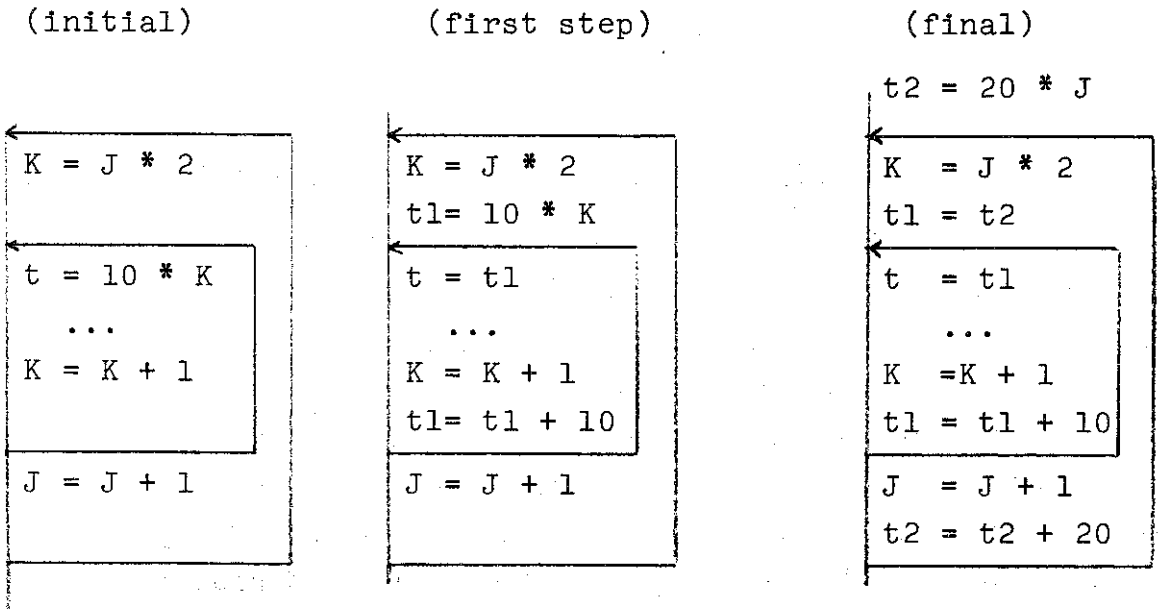


Figure 29. Iterated strength reduction in a nest of loops.

In order to apply outer-loop optimizations to code moved out of inner loops during a strength reduction process, we must save a certain minimum amount of information concerning the state of the optimization process immediately before the inner loop was entered. In the algorithms which follow, we choose to save all the value numbers associated immediately preceding entrance into the loop with all variables modified within a loop.

The overall structure of the strength reduction process (which forms an extension of the basic linear nested region optimization process described by Table II) is as follows. Operator strength reduction is applied to the code within a loop when the terminating branch of the loop is encountered. More precisely, on encountering the terminating branch of a loop, we apply the strength reducation process to all code properly within the loop: Code is said to be properly within the loop if it lies within the loop but not within any sub-loop of the loop. Figure 30 below illustrates this concept.

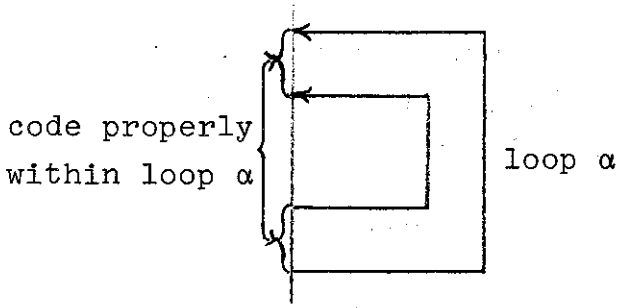


Figure 30. Code properly within an outer loop of a nest of loops.

In order to prepare for strength reduction, the optimization algorithm of Table II is extended in various ways. In the first place, we add an extra one-bit field to each entry of the stores array used by the optimizer; this bit is used during the processing of a loop to distinguish iterative assignments, i.e., those having either of the forms I = I + RC or I = RC, from non-iterative assignments. As motion out of a loop of any loop-independent expression on the right hand side of an assignment statement will precede processing of the assignment statement itself, iterative and non-iterative assignments may be distinguished without difficulty. After the reduction in strength process has been applied to a loop, we go back and flag every assignment in the loop as non-iterative. Thus if a loop to be processed contains an inner sub-loop, none of the assignments lying within the sub-loop will be treated as iterative during processing of the outer loop.

Once having distinguished between the iterative and the non-iterative assignments lying within a loop L, we begin the main reduction in strength process, as follows. A pass over all the assignments in L is made to find all those variables whose only assignments within L are iterative. For each such variable, the total number of its iterative reassignments within L is recorded, and a list of all value numbers assigned to the variable within L is collected. These value numbers, for which at certain points in further processing we will wish to substitute the value numbers assigned to the corresponding variables immediately before entry into L, are entered into a

hash table called the reducible values table to facilitate the associative look-up processes which will subsequently apply to them. Each such value number references the iterative variable to which it belongs. The reducible values table will eventually contain both value numbers assigned to iterative variables and value numbers representing reducible quantities; entries of these two distinct types will be separately flagged.

On completing the initialization steps described above, we begin the main part of the strength reduction process. All instructions properly belonging to the loop L are scanned, integer sums and products (reducible operations) being treated in one way and other operations in another. Non-reducible operations are treated as follows: their arguments are examined, and any argument found to be reducible is flagged within the reducible values table as a terminal reducible value. Integer sums and products are treated as non-reducible operations unless both of their input arguments either belong to the reducible values table or are loop-independent. In the reducible case, a nominal degree is ascribed to the operation (cf. Figures 28, 29 and the discussion associated with these figures). An operation of excessively large degree is treated as non-reducible. The value numbers corresponding to the results of operations definitely determined to be reducible are entered into the reducible values table, and the operations themselves moved temporarily to an auxiliary block associated with L.

Once the whole of the loop L has been scanned, we begin to process the instructions in the prefixed auxiliary code block associated with L, first converting all the instructions found there (which instructions direct the calculation of certain polynomials) to a more explicit representation of the polynomials which these instructions calculate. This is done iteratively by associating a polynomial with each instruction in the auxiliary block. In more detail, we associate with each product operation the canonically expanded form of the product of the polynomials associated with its separate inputs; and use a corresponding

procedure for addition operations. A more detailed account of this conversion procedure is found in Table V below.

Table V. Summary of linear nested region strength-reduction process.

((a) Steps carried out during main optimization pass, cf. Table II of Section 3.)

In processing assignment operations, detect those not having the form I = RC or I = I+RC, and flag them as non-iterative assignments;
On encountering label marking head of loop, save list of all pre-loop value numbers belonging to variables reassigned within the loop;

On encountering a loop-terminating backward branch, enter reduction in strength subroutine described below;
On return from reduction in strength subroutine, flag all assignments within loop just processed as non-iterative and erase list of pre-loop value numbers;
Then return to main optimization process of Table II;

((b) Reduction in strength subroutine applying to a given loop L.)

Make pass over all assignments belonging to loop L, detecting each variable all of whose assignments are iterative. For every such variable, calculate the total number of its iterative assignments and collect a complete list of the iterative assignments;
Enter each value number belonging to an iterative assignment into the reducible values hashtable. Flag each such value number as being the value of an iterative variable, and cause it to reference the variable to which it belongs;
Process all instructions properly belonging to the loop L by the following steps;

Scan: Scan the next instruction; if it is an integer addition vi = vj + vk or an integer multiplication vi = vj * vk go to Sumprod;

Non: Else if any input argument of the operation belongs to the reducible values table, flag it in the table as a terminal reducible value.
Then go to Scan;

Sumprod: Find the degree of vj and the degree of vk, and use them to calculate the degree of vi. Here, the degree of a value number belonging to an iterative variable is the number of iterative redefinitions of the variable, and the degree of a loop-independent value number is 0. The degree of vi = vj + vk is the sum of the degree of vj and the degree of vk; the degree of vi = vj*vk is the product max(degree(vj),1) * max(degree(vk),1);

If the degree of vi exceeds the maximum limit maxdeg for strength reduction, treat vi as irreducible and go to Non;

Otherwise, enter the value number vi into the reducible values hashtable, flag it as the value of a reducible operation, and associate its degree with it in the hashtable;

Move the operation (vi = vj+vk or vi = vj*vk) into a temporary code block and go back to Scan;

Continue scan until all instructions properly within loop L have been scanned. Then begin at Convert to process the temporary code block.

Convert: Process the successive instructions in the temporary code block, associating with each instruction a polynomial in the variables iterative within L and

having coefficients which are loop-constants relative to L. Use the following rule to determine these polynomials:

i. if vi is an L-constant value number, the polynomial p(vi) associated with it is itself,

ii. if vi is a value number belonging to an L-iterative variable I, then the polynomial p(vi) associated with it is I,

iii. if vi = vj * vk, then p(vi) is the canonically expanded form of the product polynomial p(vj)*p(vk),

iv. if vi = vj + vk, then p(vi) is the canonically expanded form of the sum polynomial p(vi) +p(vk);

If the result value number vi of the instruction being processed is flagged as a terminal reducible value, call a subroutine expand with argument p = p(vi) to generate all initialization code associated with its successive differences, and record in the reduction quantities structure table a collection of entries describing these differences;

When all instructions in the temporary code block have been processed, erase the temporary code block and go on to Modproc to attach all necessary supplements to redefinitions of iterative variables of L;

Modproc: Make a pass over all the iterative redefinitions α: I = I + RCI and I = RCI of iterative variables I belonging to the loop L;

For each redefinition α, make a pass over all the reduction quantities t;

If α has the form I = I + RCI and s = t_{RCI} is defined in the reduction quantities structure table, attach the compensating instruction t = t + s to α;

If α has the form I = RCI and $s = t_{(RCI)}$ is defined in the reduction quantities structure table, attach the compensating instruction t = s to α;

When all the iterative redefinitions α in the loop L have been processed, clear the reduction quantities structure table calculated for the loop L, and return from strength-reduction loop-processing subroutine.

-- end of algorithm --

When, during the conversion of instructions belonging to the auxiliary code block associated with L, an instruction flagged as defining a terminal reducible value is encountered, a number of additional procedures are carried out. In the first place, we associate with the polynomial P corresponding to the terminal reducible value a number of additional reduction quantities. This collection of quantities is intended to include all those quantities whose current values must be maintained in order that the values of all reducible quantities may be updated using additions only. The set of these quantities may be defined as follows. For each polynomial P = P(I,J,...) corresponding to a terminal reducible value, we must keep on hand

i. the current value of P.
ii. for all variables I, J, etc. iterative within L, the value of all non-zero differences $\Delta_I \Delta_J \ldots P$, combined in all relevant ways with corresponding loop-independent increments RCI, RCJ, etc., belonging to iterative redefinitions of I,J,..., having the form I = I + RCI, J = J + RCJ, respectively.
iii. the value of all substituted versions P(RCI,...) of the polynomial P and all the above differences, as dictated by the presence within L of iterative redefinitions having the form I = RCI.

With each reduction quantity we associate a compiler-generated temporary variable t; for convenience of exposition we use the following notational convention. If P(I,...) is a reducible polynomial with which the reduction quantity t is associated, and Q is the polynomial such that P(I+RCI,...) = P(I,...) + Q(I,RCI,...), we designate the reduction quantity associated with Q by the symbol t_{RCI}; if R is tne substituted polynomial P(RCI,...) we designate the reduction quantity associated with R by the symbol $t_{(RCI)}$. Using these notational conventions, the modifications to the code inside and outside the loop L necessary in order to complete the next phase of the strength reduction process may be described as follows (cf. Table V for additional details).

i. Insert, immediately before the first instruction of the loop L, a set of instructions appropriate for the initialization of every reduction quantity. In doing so, replace each L-iterative variable I appearing in the polynomial representation of a reduction quantity by the value number associated with I immediately before entry into the loop L.

ii. At appropriate points within the block of instructions thus generated, insert the assignment operations necessary to initialize the compiler generated temporary variables t associated with the various reduction quantities.

In the algorithm shown in Table V, the necessary initialization code is assumed to be generated by a subroutine expand which also prepares a table, the reduction quantities structure table, in which all the relationships $t \rightarrow t_{RCI}$ and $t \rightarrow t_{(RCI)}$ are represented.

iii. Make a pass over all the iterative redefinitions α: I = I + RCI of iterative variables I belonging to the loop L. For each such redefinition α, make a pass over all the reduction quantities t, checking to see if the quantity $s = t_{RCI}$ is defined. In this case (i.e., if P does depend on the variable I) attach the compensating instruction t = t+s to α. Then go on to process the next iterative redefinition α in the loop L. When a reduction quantity s is found to be a constant, the

initialization of s is to be omitted and direct use made of the known value of s.

iv. A similar process applies to iterative redefinitions I = RCI of the variable I. To each such iterative redefinition we associate supplementary redefinitions t = $t_{(RCI)}$.

When all iterative redefinitions occurring in the loop L have been processed in this fashion, the strength reduction process, as it applies to the loop L, is complete. It only remains to flag every assignment lying within L as non-iterative, and to proceed with the main strength reduction process shown in Table II until the next loop-terminating backward branch is encountered, at which point one again carries out a strength reduction process.

In order to describe the strength reduction procedures used by the linear nested region optimizer in more detail we must first look at the principal data structures used by the reduction process. These data structures are as follows. A saved value number list records the value number assigned to all loop-modified variables immediately prior to entry into each loop. A stores array, identical with that used in linear nested region otpimization as described above but containing certain extra fields, gives information on all the assignment operations contained in the code to be optimized. A related iterative stores array lists all the elements of the stores array which reference iterative stores, and gives certain additional information concerning these stores. A reducible values table is used during the processing of a loop L to reference and to describe every value number attaching to an iterative variable or to a calculation iterative within L. As has already been noted, given a loop L we call every polynomial P whose calculation is moved out of L by the strength reduction process, and every auxiliary quantity needed to be able to calculate the value of such a polynomial using additions only, a reduction quantity; each such quantity is assigned an identifying number. A polynomial P whose calculation is moved out of L is called a terminal reducible polynomial. A terminal reducible polynomials list containing the identifying number of each

terminal reducible polynomial belonging to a given loop L is maintained. A reducible quantities structure table is used to relate each reduction quantity t and each iterative redefinition α of a variable within L to its appropriately substituted versions and to the increments s which must be added to t in order to account properly for the occurrence of α.

In more detail, the entries in these various data structures have the following forms. A stores array entry has the form:

(targ)	(indexed)	(itrat)
target variable	in	it

Figure 31. Fields in a stores array entry.

The first field in such an entry references the target variable of the store. Two additional one-bit fields flag the store as being indexed/non-indexed and iterative/non-iterative respectively. An entry in the iterative stores array has the form shown below:

(storop)	(incrno)	(targ)	(typ)	
store operation reference	value number of increment or loop constant	target variable	assign type	

Figure 32. Fields in the iterative stores array.

The first field locates an iterative store operation within the modified code text produced by the main part of the linear nested region optimizer. Two additional fields respectively define the value number of the increment (or loop constant) occurring in the store operation and the target variable of the store operation. A final one-bit flag field serves to distinguish assignments I = RC from assignments I = I + RC.

The reducible values table has entries of two types, one describing iterative variables of a loop, the other describing reducible expressions. These have the following form:

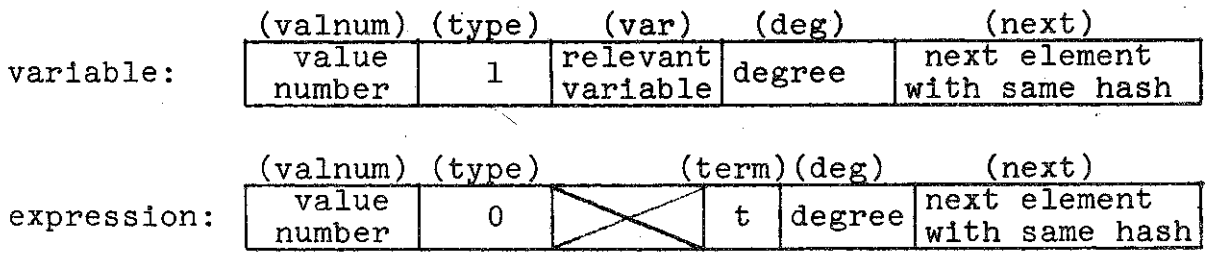


Figure 33. Fields in reducible values table entries.

The first field gives the value number of the reducible expression of iterative variable. A one-bit flag defines the type (variable or expression) of the entry. In an entry of variable type, a reference to the relevant variable is contained in the field which follows; after this we find a field containing the number of iterative assignments to the variable, i.e., the degree of the variable. In an entry of expression type only the second of these two fields is used. Such entries, however, contain an additional one-bit field flagging them as terminal/non-terminal. The entries in the reducible values table will normally be accessed through a hash based on the value number field of the entry; an extra field is reserved for structural use in connection with the associated hash search.

A reducible values counter is used to generate a unique serial number for each reducible quantity; the serial numbers of terminal reducible quantities are listed in ascending order in the terminal reducible quantities list. The reduction quantities structure table uses these serial numbers to relate each reduction quantity t to a reduction quantity s representing the t-increments associated with each iterative assignment I = I + RCI and the substituted quantity associated with each assignment I = RCI in the loop L within which reduction in strength is proceeding. The entries in this table are located via a hash based on the identifier of the variable I, the value number of the loop independent increment or replacement RCI, and the quantity t. The entries in this table have the following form.

(var)	(valinc)	(redq)	(repinc)	(flag)	(next)
incremented variable	value number of increment or replacement	reduction quantity	replacement or increment quantity		next element with same hash

Figure 34. Fields in the reduction quantities structure table.

Note that the first three fields contain the information containing I, RCI, and t described just above, while the fourth field defines the associated t-increment or replacement quantity s. A one-bit flag field distinguishes the increment from the replacement case. A final field is used in connection with hash access.

We evade all the thorny problems having to do with code motion out of loops entered by forward branches by not applying the reducation in strength process to such loops. Note however that, at the cost of a certain complication of our algorithms, they can be extended to give coverage to this case also. For the same reasons as are explained earlier in the present section in connection with code motion, we will not attempt to reduce expressions calculated under conditional branches. Note that the presence of forward branches wholly contained within a loop does not prevent us from reducing expressions in the loop as long as they are calculated unconditionally within the loop. The information necessary to administer those constraints on code motion is available in the loop and branch stack used by the optimizer.

Table VI below gives a more detailed version of the algorithm described in broad outline in Table V.

5. The Flow Structure of a Program. Introductory Considerations.

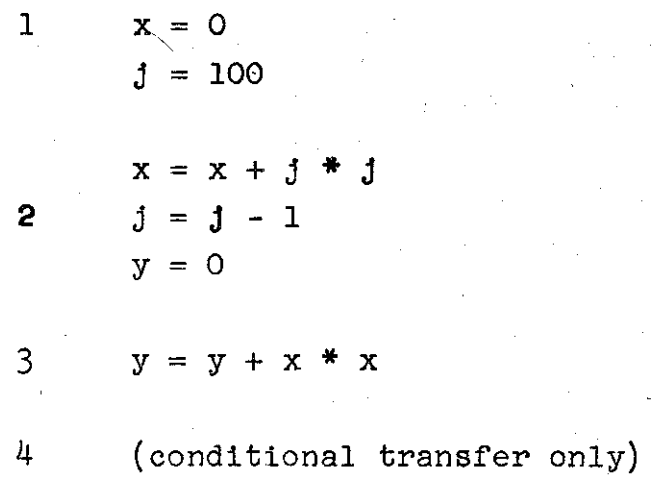
In the previous section we introduced the notion of basic block, i.e., of a program subsection involving only linear control flow. In the present section we initiate our discussion of general control flow and begin to describe the optimization algorithms applicable in this much more complex situation.

The flow structure of a program may be represented conveniently in terms of a program graph. This graph consists of nodes and directed edges. Each basic block of a program is represented by a node of the program graph. A basic block will be terminated either by the occurence of a labeled instruction defining a point in the program to which control may be transferred from elsewhere, by a subroutine or function call, or by a string of conditional transfer operations which transfer control from the end of a given block to the beginning of some other block. If, in a given program, control may flow from the terminating instructions of a certain block B1 to the initial instructions of some other block B2, we introduce a directed edge in the program graph proceeding from B1 to B2. The program graph is then completely defined by its set of nodes (representing basic blocks of the program) and by the set of directed edges connecting nodes (representing transfers of control from one basic block to another). Thus, for example, the code

```
(1)             ...
                x = 0
                j = 100
loop1:          x = x + j * j
                j = j - 1
                y = 0
loop2:          y = y + x * x
                if(y.gt.1000) go to next
                go to loop2
next:           if(j.gt.0) go to loop1
                ...
```

has the following program graph.

(2)



Note that the first two instructions of (1) constitute a first block and are represented by node 1 in (2), that the next three instructions constitute a basic block and are represented by node 2 in (2), and that the next basic block, represented by the node 3 of (2), consists of a single instruction. The two conditional transfers which follow this single instruction are represented in (2) by the two directed edges which leave node 3. The final conditional transfer defines a fourth block containing no actual instructions but represented in (2) by the node 4 and the directed edge passing out of it. In (2), we also show an additional directed edge, passing 'in line' to the remaining code which is not explicitly indicated either in (1) or in (2).

Note that the graph of a program gives a truer representation of its logical structure than the normal sequential form of the same program. Whereas the physical structure of computer memories compels us to arrange the instructions constituting a program in some linear order, this order, except within basic blocks, merely **linearise** a logically nonlinear program graph in an arbitrary way. (Nevertheless, some of the optimization methods to be studied later will make algorithmic use of linear code arrangements. Part of our effort in the present chapter will be devoted to the development of rational linearizations of general program graphs). A basic block terminating in a 'call exit'

statement or 'end' statement defines a terminal node of a program graph. The unique block containing the first executable statement of a program defines the unique initial or entry node of its program graph; we assume that this node is not the successor of any other node in the graph.

If $P \in \mathcal{P}$, $\tau(P)$ is the set consisting of all those nodes which are the terminal nodes of directed edges of the program graph whose initial node is P. A program graph may equivalently be described by a set $\mathcal{P}$ of nodes, and by the multi-valued 'successor' mapping $\tau : \mathcal{P} \longrightarrow \mathcal{P}$. A track in a program graph is a sequence of points $P_1, P_2, \ldots, P_n$, such that $P_{i+1} \in \tau(P_i)$. P_1 is called the beginning of the track, P_n is called the end of the track. If there exists a track whose beginning is P_1 and whose end is P_n, then P_n is said to be a successor of P_1, and P_1 is said to be a predecessor of P_1. In a program graph every node which is a successor of the unique entry node of the program graph is called a reachable point. We shall normally assume that every point of a program graph is reachable; it is clear that instructions belonging to unreachable nodes can never be executed during any run of a program. Such instructions are superfluous and may as well be omitted from the program. Subsequently in the present section, we shall have considerable occasion to examine the flow structure of programs more deeply. However, with this basic introduction, we have sufficient material in hand that we may profitably begin to study the two optimizations to which the preceding section was devoted, i.e., optimization by the elimination of redundant operations and optimization by the elimination of dead calculations, both in the context of general program flow.

An operation A * B (i.e. an operation which combines two inputs A and B to give some sort of result, which we write as A * B) is redundant if there exists no track in the program graph, either beginning at the program entry block,

or beginning at any assignment of a new value to one of the variables A or B, which reaches the given operation without passing through some preceding calculation of the result A * B. All other operations are irredundant. In making this definition, we intend that every operation A * B be followed wherever necessary by a store operation, so that the "last calculated" value of A * B is always available if needed. We call the cell in which this value is stored the "value cell" of A * B." Any single redundant operation can be eliminated and replaced by a load from the corresponding value cell. Of course, value cells need be established and stores inserted only for those operations at least one redundant instance of which is eliminated by our optimization algorithms. Moreover, a large number of the store operations whose insertion we here find it convenient to assume will be eliminated by the "dead operation" optimizations which we shall consider in a later section. Note also that, if an operation A * B is deleted and replaced by a load from the corresponding value cell, it follows that at the time that the load replacing the original operation A * B is performed, the value cell already contains exactly the result which would have been stored into it, were the result A * B to be performed and its result stored again. Hence, the replacement of any redundant operation by a load from the corresponding value cell does not change the result obtained, in the one case by actual calculation, in the second case by a mere load. Hence every redundant operation A * B occurring in a program may be removed and replaced by a load without this replacement affecting the results of any computation. On the other hand, irredundant operations must actually be performed, since there exists a direct track to them from some point in the program at which the value of one of the inputs A or B to the operation A * B has been recalculated. We may therefore assert that common subexpression elimination "in the large," i.e., in the presence of program flow, consists of the elimination of all operations redundant in the sense just explained.

Given this fact, the problem of realising this optimization becomes the problem of deciding which operations in a complex program flow graph are redundant. We aim to describe algorithms for the effective determination of just this. In order to develop the necessary algorithms, it is convenient for us to introduce a certain number of new terms. We call an operation relevant for the operation A * B if it is either an assignment to the variable A, an assignment to the variable B, or a formally identical calculation A * B. Each operation A * B occurs in some basic block. If the last preceding relevant operation in this block is an assignment to A or B, the given operation A * B is irredundant and must be performed. If the last preceding relevant operation is a calculation of A * B, then the operation is redundant and should definitely be eliminated. The interesting case for global optimization is that of an operation A * B not preceded in its basic block by any other relevant operation. This case we treat as follows.

With each edge e in a program graph (and with each particular calculation A * B), we associate a Boolean value x_e, equal to 1 if an occurrence of the calculation A * B as the first instruction in the block in which the edge e terminates would be irredundant, and to 0 otherwise. We also introduce a second Boolean quantity y_b, equal to 1 if an occurrence of the operation A * B as the first instruction in a block b would be irredundant, and to zero otherwise. We call x_e the leaving bit associated with the edge e and the operation A * B, and call y_b the entering bit associated with the block b and the same operation.

Let e be an edge, and let b be the block from which e originates. The relationship between the Boolean quantities y_b and x_e is a characteristic of the basic block considered. If the block contains no instruction relevant to the instruction A * B, then clearly $x_e = y_b$. On the other hand, if the basic block b does contain some instruction relevant to A * B, then the relation between x_e and y_b is as follows.

If the last relevant operation in b is an assignment to A or B, then $x_e = 1$. If the last relevant operation in b is a calculation of A * B, then $x_e = 0$. In any case, we may write $x_e = C_e . y_b + D_e$, where, here as in the remainder of the present section, we write the Boolean "and" as a symbolic product, and the Boolean "or" as a symbolic sum. The Boolean coefficients C_e and D_e are of course, characteristics of the block b and edge e. We may write the preceding equation more succinctly as $x_e = f_e(y_b)$, where f_e is the Boolean function defined by $f_e(y) = C_e . y + D_e$. Call this Boolean transformation the edge transformation associated with the edge e and the operation A * B. It follows at once from our definition that $y_b = x_{e_1} + \ldots + x_{e_n}$, $e_1, \ldots e_n$ being the edges of the program graph which terminate in the block b; indeed an operation A * B, standing at the beginning of the block b, is irredundant if and only if the same operation, standing at the end of some one of the edges entering block b would be irredundant. Note that the operation A * B, occurring as the first relevant operation in its block, is redundant if and only if $y_b = 0$. Thus, the problem of determining the reudndancy of the operation A * B, when it is the first relevant operation in its basic block, i.e., in that case in which its relevancy depends on the global structure of the program graph, is precisely that of calculating the quantity y_b for each block b.

Equivalently, and somewhat more conveniently, we may calculate the quantities x_e for all the edges of the program graph. For the quantities x_e we have the system of Boolean equations (1) $x_e = f_e(x_{e_1} + \ldots + x_{e_n})$ so that our problem is essentially that of calculating the Boolean quantities x_e by solving this system of Boolean equations.

Note that similar definitions may be made and a similar line of reasoning applied not only to binary operations, but also to the calculation of functions of any number of variables. Consider, for example, a hypothetical operation G(A,B,C) producing some result from three inputs A, B, C. This

operation is redundant if there exists no track in the program graph, either from the origin block of the program or from an assignment to one of the variables A, B, or C, which reaches the given operation without passing through some other formally identical operation. All other calculations of G(A,B,C) are irredundant. For the same reason as in the binary case, we may, by associating a hypothetical value cell with each such operation, delete every redundant instance of the calculation G(A,B,C) and replace it by a load from the corresponding value cell. With each edge in a program graph, and with each such operation, we may as in the binary case associate a Boolean value x_e equal to 1 if an occurrence of the operation G(A,B,C) as the first instruction reached by the edge would be irredundant, and to zero otherwise; and also a Boolean quantity y_b having a similar definition but pertaining to entry to the given basic block. We call an operation relevant to the operation G(A,B,C) if it is an assignment of a value either to the variable A, or B, or to C, or is a computation formally identical with the computation G(A,B,C). If a given basic block b contains no occurrence of any instruction relevant to the operation G(A,B,C), then $x_e = y_b$. If the last relevant operation in b is a calculation of G(A,B,C), then $x_e = 0$. Thus the relationship between the Boolean quantities x_e and y_b and the pattern of Boolean equations for the calculation of the composite collection of bits x_e, are the same whether the operation considered has two or more input quantities. We may use this fact to treat composite calculations like (A+B)*C on the same basis as simple calculations. Note that if F_e is the edge transformation associated with such a composite operation while f_e is the block transformation associated with the subcalculation like A+B, then necessarily $f_e(x) \leq F_e(x)$ in the sense of the evident Boolean ordering $0 \leq 1$. It will follow from our subsequent discussion that, of x_e are the irredundancy bits associated with the composite operation (A+B)*C in the sense explained above, while x_e

are the corresponding bits associated with the suboperation A+B, it necessarily follows that $x_e \leq X_e$ for each edge e. Thus, if the calculation (A+B)*C is redundant in a given block, then the associated suboperation A+B is also redundant in the same block. Hence, an optimization procedure based on redundancy bits is consistant with an underlying data structure which represents composite operations by pairs of pointers to the simpler operations of which they are composed.

A very similar line of reasoning may be applied to study the elimination of useless operations, and especially of store operations, when they are "dead," i.e., never subsequently used in the course of a program run. A store instruction is dead if there is no path from it to any use of the value which it stores which does not pass through another store of the same variable. Suppose that with each store and each edge of the program graph we associate a bit x_e, indicating whether the same store operation occurring at the beginning of the block in which the edge terminates would be live rather than dead, and a second bit y_b indicating whether the corresponding store operation, occurring at the end of block b in which e termintes, would be live. Let e be an edge, and let b be the block in which e terminates. Then three cases arise. If the variable A has a use in the block b before any store operation assigning a value to A occurs, then $x_e = 1$. If, on the other hand, an assignment to A occurs in b before any use of A, then $x_e = 0$. Finally, if the variable A has no stores and no uses in the block b, then $x_e = y_b$. Now, since a variable is live at the end of a given block if and only if it is live at the beginning of some block to which transfer may be made from the given block, it follows that $y_b = x_{e_1} + \ldots + x_{e_n}$, where $e_1, \ldots, e_n$ are the edges originating from the block b. The reverse graph of a graph with ordered edges is the

graph with the same nodes and edges, but with the orientation of each edge reversed. In terms of this notion, we see that the general pattern of Boolean equations for the calculation of the live-dead bits associated with a store operation have the same form as the system of Boolean equations determining the redundancy of operations A * B, except that the reverse program graph rather than the original program graph must be used.

The above argument shows that systems of Boolean equations of the form (1) play a considerable role in two situations of great interest to us. We now begin an examination of the solutions of systems of Boolean equations of this form. Note first that the equations (1) are "linear" in the Boolean sense. It follows that the Boolean sum of two solutions of our system of equations is also a solution of the same system of equations. Since the sum of any two distinct solutions of these equations is greater in the Boolean sense than either of the two solutions out of which it is composed, it follows also that there exists a maximal solution of our system of equations, i.e., a solution X_e with the property that $X_e \geq x_e$ for all e and for any other solution x_e of the same system of equations. Suppose, on the other hand, that we put $x_e^1 \equiv 0$, and, iteratively, put

$$(2) \qquad x_e^{(m+1)} = f_e(x_{e_1}^{(m)} + \ldots + x_{e_n}^{(m)}) ,$$

where $e_1, \ldots, e_n$ are the edges of the program graph terminating in the block b from which e originates. Then, since every one of the block transformations f_e is a monotone Boolean function, it follows by induction that the quantities $x_e^{(m)}$ are monotone increasing with m, and hence reach a limit after a finite number of steps. Moreover, it follows by a similar induction that, if x_e is any solution of the system of equations

$$(3) \qquad x_e = f_e(x_{e_1} + \ldots + x_{e_n}) ,$$

then $x_e \geq x_e^{(m)}$ for every e and m. Therefore, if we let $x_e^{(\infty)}$ be the limit of the quantities $x_e^{(m)}$, then $x_e^{(\infty)}$ is that unique solution of the system (3) of Boolean equations which is less than or equal to any other solution of the same system of Boolean equations. We call this solution of the system (3) its minimal solution. We conclude that there exists a minimal and a maximal solution of the system (2) of Boolean equations. Any other solution of the system (3) lies above the minimal solution and below the maximal solution in the sense of the natural order of boolean quantities.

Let x_e be the minimal solution and x_e the maximal solution to the system of Boolean equations (3). We call those edges e of the program graph for which $X_e = x_e$ determinate edges for the system (3), and those edges for which $x_e = 0$, while $X_e = 1$, indeterminate edges for the system (3). Then since for each indeterminate edge e the edge transformation f_e can assume either of the values 0 or 1, we plainly have $f_e(x) \equiv x$ for each such edge. Moreover, since $x_e = 0$, we must have $x_{e_1} = 0$ if e_1 is a determinate edge from which terminates in a node from which an indeterminate edge originates. It follows that the system of Boolean equations (3) reduces, for the set of indeterminate edges, to a separate subsystem having the form

$$x_e = x_{e_1} + \ldots + x_{e_m} . \tag{4}$$

where $e_1, \ldots, e_n$ are the indeterminate edges terminating in the block from which the edge e originates. It is then plain that both $x_e \equiv 0$ and $x_e \equiv 1$ are solutions of the system (4) of equations, so that, in the situation just described, indeterminacy of solution actually exists.

Since, in all our applications, irredundancy of an operation always is a consequence of the existence of a track from the origin node of the program graph or from a store operation assigning a new value to one of the inputs of a given operator, the significant solution of the system (3) of Boolean equations, for our purpose, is always the minimal solution.

Our method for the calculation of the block entry and edge exit bits y_b and x_e for any given operation, will be based on the consideration of subregions of graph. Let R be some subset of the nodes of a program graph. Let $a_1,\ldots,a_k$ be the collection of all nodes in R reached by edges from nodes of the program graph external to the region R. Let us write, as a system of Boolean equations, the subsystem of (3) applying to the nodes in R, but modify the equation for each of the nodes $a_1,\ldots,a_k$ to include one extra inhomogeneous term, i.e., to read

$$x_e = f_e(z_a + x_{e_1} + \ldots + x_{e_m}) , \tag{5}$$

where z_a is the inhomogeneous term in question, and where $x_{e_1},\ldots,x_{e_m}$ are the edges of the program graph originating at some node <u>in the region</u> R from which the edge e originates. Forming the minimal solution of this system of equations in the sense explained above, we obtain a Boolean function $f_e^R(z_1,\ldots,z_k)$; this function depends on as many Boolean variables as there are entry nodes to the region R. In particular, if R is a <u>single entry region</u>, f_e^R is a Boolean function of a single Boolean variable, and hence a function having the same form and (as we shall soon see) much the same intuitive significance as the edge transformation functions with which we are already familiar. Note that, if e is an edge originating in R but terminating outside R, equation (4) will still define a value $f_e^R(z_1,\ldots,z_n)$; this case is simpler than the case in which e both originates and terminates in R, since in this case the values of the variables $x_{e_1},\ldots,x_{e_m}$ occuring in (4) may be determined from the other equations (4) applying to the region R and the value x_e does not appear in any of these other equations.

The theoretical significance of the functions f_e^R may be understood as follows. Suppose that we partition the program flow graph $\mathcal{P}$ into a collection $R_1,\ldots,R_k$ of disjoint single-entry regions with entry nodes $b_1,\ldots,b_n$. Let $\{x_e\}$ be the solution of the Boolean redundancy equations (2), and, for each j, let $E(b_j)$ be the set of edges originating outside the region R_j and ending in b_j; then put

$$z_{b_j} = \sum_{e \in E(b_j)} x_e \quad . \tag{6}$$

If e originates in R_j, it follows from the definition of the functions f_e^R that the value of x_e, as determined by (2), is also determined by the set of equations

$$x_e = f_e^{R_j}(z_{b_j}) \quad . \tag{7}$$

Out of this set of equations we may extract the subset consisting of all those equations applying to edges e which go from one region R_k to another region R_j. The subset of the equations (7) applying in this case may be written as

$$x_e = f_e^{R_j}(x_{e_1} + \ldots + x_{e_i}); \tag{8}$$

have e is any edge leaving R_j for another region, while $e_1,\ldots,e_i$ are all the edges which originate in another region and terminate in the region R_j from which x_e originates. Note then that the system (7) has a definition exactly parallel to the definition of the system (2), except that

(a) where the basic edge transformations f_e appear in (2), the region-associated transformations f_e^R appear in (8);

(b) where all the edges originating outside a block and terminating in a block appear in an equation (2), all the edges originating outside a region and terminating in the region appear in (8);

(c) whereas all edges of the graph P appear in (2), only edges which cross from one region to another appear in (8).

In particular, (8) may be a much smaller system of equations than (2). Note that, once the equations (8) have been solved, the values x_e for all intra-region edges e may be determined by use of the equations (6) and (7). Of course, the use of the region functions f_e^R in the manner just outlined will only be of advantage if it is substantially easier to solve the system (5) of equations determining these functions than to solve the full set (2) of equations. Assuming this to be the case, however, we may elaborate the following strategy for the complete solution of the equations (2):

i. Partition P into a disjoint set of single entry subregions R_j, each having an internal structure sufficiently simple so as to make the calculation of the associated functions $f_e^{R_j}$ easy.

ii. Treating each region R_j as if it were a block, eliminating all intra-region edges, and using the functions $f_e^{R_j}$ in place of the functions f_e, set up the system of equations (8).

iii. Solve the system of equations (8), determining all the redundancy bits x_e fpr inter-block edges e;

iv. Use equations (6) and (7) to determine the redundancy bits x_e for intra-block edges e.

It is also worth noting that the transformation defined by i) and ii) is available for iterative reapplication in developing the solution required in step iii).

In what follows, we shall show how subregions R_j of a program graph P which are especially well-adapted for use in connection with the above strategy may be developed.

The fact that, in discussing solutions of the system (2) of Boolean equations, we are able to treat single entry regions on much the same basis as we treat basic blocks gives to single entry regions in a program graph the exceptional importance which the reader will preceive they have in our subsequent discussion. Note that the function f_e^R is a monotone function of its variables, i.e., that its Boolean value always increases if one of its arguments is increased in the Boolean sense.
We may also define, for a region R and the node in that region, a function which has the same significance for block entries as the function $f_e^R(z_1,\ldots,z_k)$ has for block exits. This function is given simply by

$$(9) \qquad E_b^R(z_1,\ldots,z_k) = f_{e_1}^R(z_1,\ldots,z_k) + \ldots + f_{e_m}^R(z_1,\ldots,z_k)$$

in case the block b in question is one which is not the terminus of any branch originating outside R, and the form

$$(10) \qquad E_a^R(z_1,\ldots,z_k) = z_a + f_{e_1}^R(z_1,\ldots,z_k) + \ldots + f_{e_m}^R(z_1,\ldots,z_k).$$

if the node a in question is one which is the terminal node of some branch of the program whose origin is external to R. In (9) and (10) $e_1 \ldots e_m$ are those edges of the program graph which originate in R and terminate in b (or in the sense of (9), in a). We call the function $E_b^R(z_1,\ldots,z_k)$ the Boolean entering operator function for the block b and region R. The corresponding function f_e^R is called the Boolean leaving operator function for the given edge and region.

In subsequent sections of the present chapter, we will see that the functions f_e^R and E_b^R are highly significant. Indeed, much of our analysis will relate directly or indirectly to these functions. For the present, we confine ourselves to discussing one single application of these region functions: the manner in which they may be used to study the possibility

of moving an occurrence of a given operation from one point of a program to another. Suppose, to be more specific, that A * B is an operation occurring in a certain block of a single entry region R of a program and that this operation is not preceded in its block by any other relevant instruction. Let $E_b^R(x)$ be the Boolean entering operator function for the given block and region. If $E_b^R(0) = 1$, then the operator is not redundant in its location, no matter whether it is redundant on entry to the region R or not. If $E_b^R(0) = 0$ and $E_b^R(1) = 0$ also, then the operation A * B is redundant in any case, and may be deleted and replaced by a load from the corresponding value cell. If, finally, $E_b^R(0) = 0$ and $E_b^R(1) = 1$, then the operator is redundant if and only if the corresponding operation placed immediately previous to the entry block of the region R would be redundant. Thus, if while calculating the value of the redundancy bits for larger and larger regions we find that the operation A * B would not be redundant on entry to the subregion R, but nevertheless that $E_b^R(0) = 0$ and $E_b^R(1) = 1$ (and with the important "safety" provision that performance of the operation cannot produce any "side effects" such as overflows or machine traps), it follows that if a copy of the instruction is placed immediately previous to the entrance to the region R, the operation A * B may be deleted in its present location. The net effect of the program transformations just described is, of course, to move the operation A * B from the point within the region R at which it originally occurs, to a point immediately preceding the entry block of the region R. If the region R contains internal cycles, it may well be that the nodes in R are traversed during a run of the program much more frequently than the region R is entered; this would surely be the case if, for example, the region R represents an inner loop of the program being optimized. In this case, by moving the operation A * B from one place to another, we obtain a valuable gain in efficiency. As our procedure for the calculation of the

redundancy bits x_b is, in any case, based on the calculation of the region functions E_b^R and F_e^R for larger and larger subgraphs of a program graph, starting with innermost subgraphs and working outward, it is particularly fortunate that the criterion for code motion has so simple an expression in terms of the region functions E_b^R.

Note in this connection that the insertion into a code of an additional calculation A * B (which, as always, we assume to store its result into an associated value cell) cannot change the quantity present in the value cell at any subsequent redundant operation. This value could only be changed if there existed a track from the inserted calculation to a redundant calculation of A * B, starting from some operation which assigns a new value to either A or B, and containing no other intervening recalculation of A * B. This, however, is impossible by our definition of redundancy.

Note also in this connection that it is obvious that during every run all the instructions constituting a single basic block will be executed equally often. Thus, we may associate an execution frequency with each node of a program graph. This frequency will of course be data dependent. Since program optimization must be performed at compile time, and hence must be independent of any knowledge of the data which a program has to process, one generally assumes or implicitly estimates data-independent frequencies in making decisions concerning optimization.

The fact that different nodes in the program graph will be executed with widely different frequencies makes it possible to improve the running time of programs by moving instructions, where possible, from their initial positions to nodes in the program graph which will be executed less frequently. This is the fundamental point of optimization by code motion. Of course, it will not be feasible for us to study such optimizations in complete generality; the most general optimization by code motion would hardly be distinguishable from the most general reorganization of

a given program into an equivalent algorithm. We can however, using the methods described above, consider those special code motion optimizations which can be derived from iterated simple motions. Of course, by moving first one and then another instruction in an iterative manner, whole groups of instructions may eventually be moved. Nevertheless, the code-motion optimizations which can be derived in this fashion form a subset of the considerably more general class of optimizations involving the unitary motion of whole blocks of instructions.

In some cases, as an aid to optimization, a programmer is asked to supply, along with his source language program, the probabilities with which various branches leaving a basic block will be taken. From these probabilities the frequency of execution of any given block may be estimated in a simple way. Let the nodes constituting a program graph be enumerated in some order, and let f_i be the execution frequency of the i-th node. On the assumption that the first node in the enumeration is the unique entry node of the program, it clearly follows that $f_1=1$. For each i and j, let p_{ij} be the probability of a transition leaving the j-th node of the program graph along the directed edge leading to the i-th node of this graph. Then the frequencies f_i are determined from the probabilities p_{ij} by the equation $f_i = \sum_j p_{ij}f_j$. The coefficients p_{ij} are of course non-negative. Moreover, the sum $\sum_i P_{ij}$ has the value 1. Thus, the norm of the transformation represented by the matrix $p = \{p_{ij}\}$ is 1, and all eigenvalues of this matrix are confined to the circle $|z| \leq 1$. Moreover, it is clear that the point $\lambda = 1$ of the complex plane is an eigenvalue of the adjoint of the matrix P and hence of the matrix P itself; $\lambda = 1$, regarded as an eigenvalue of the adjoint matrix to the matrix P, has a positive eigenvector, all of whose components are equal to one. The classical Frobenius theorem concerning matrices with non-negative elements now enables us to assert that $\lambda = 1$ is the eigenvalue of the matrix P which is largest in magnitude, and that, both for the transpose of P and for P, this eigenvalue has a one dimensional eigenspace. It follows, therefore, that the equation $f_i = \sum_j p_{ij}f_j$ has a solution which is uniquely defined up to a constant factor, and that this solution is strictly positive. The additional condition that the program is entered only once then determines the frequencies f_i uniquely as a solution of the preceding equation.

6. "Intervals", Program Flow, and the Boolean Redundancy Equations.

In the previous section, we saw that a certain system of Boolean equations, and a certain solution of these equations, plays a very significant role in optimization by redundant instruction elimination on a global scale, i.e., in the presence of program flow. We indicated that we intended to base our calculation of the solution of these Boolean equations on the analysis of subgraphs of the program graph, and, in this connection, introduced a pair of functions $E_b^R(z_1,\ldots,z_k)$ and $f_e^R(z_1,\ldots,z_k)$, the so-called entering and leaving functions for a given block and region or edge and region and for a given operation. In calculating these functions, certain special single entry subgraphs of a graph are particularly significant.

We now define, in a precise way, the subgraphs in which we are interested. A subgraph of a program graph $\mathcal{P}$ is said to be loop-free if it contains no track, more than one point long, whose first and last points are identical. We define an interval in a program graph $\mathcal{P}$ to be a subset $\mathcal{Q}$ containing a unique distinguished entry node q, and having the property that $\mathcal{Q} - q$ is loop free. We will sometimes call the node q the head of the interval $\mathcal{Q}$. Since, by definition of a program graph, every node in the interval can be reached from the entry node of the program graph, it follows that every node of an interval can be reached along a track beginning at the head of the interval. Suppose that, given an interval $\mathcal{Q}$, we order the nodes of $\mathcal{Q} - q$ by selecting the nodes successively, never selecting any node p while we can still select some other node in the interval which precedes p along a track of the interval. Proceeding in this way, we establish a linear ordering for all the nodes of $\mathcal{Q} - q$, such that every predecessor of any node p but the first lies previous to this p in this linear ordering. Using this fact, we may calculate the Boolean leaving operator function for each edge e originating in the region $\mathcal{Q}$ as follows.

Let $\mathcal{R}$ be the program graph obtained from the interval $\mathcal{Q}$ by keeping the same set of nodes but deleting all edges originating at a node of $\mathcal{Q}$ and terminating at the head p of the interval $\mathcal{Q}$. It is apparent that $f_e(z) = f_e^{\mathcal{R}}(z)$ for all the edges e of the interval $\mathcal{Q}$ if $L_e^{\mathcal{R}}(0) = 0$ for every edge in $\mathcal{Q}$ terminating in the node p. On the other hand, if there exists any edge in $\mathcal{Q}$ which terminates in p, such that $f_e^{\mathcal{Q}} = 1$, then $f_e^{\mathcal{Q}}(z) = f_e^{\mathcal{R}}(1)$ for all b. Thus calculation of $f_e^{\mathcal{Q}}$ is reduced to calculation of $f_e^{\mathcal{R}}$. However, since the graph $\mathcal{R}$ is wholly loop free, and since we have already seen that its nodes may be arranged in linear order in such a way that every node comes before any of its predecessors within $\mathcal{R}$, the calculation of the function $f_e^{\mathcal{R}}$ is straightforward. We have $f_e^{\mathcal{R}}(z) = f_e(z)$ for each edge originating from the first node p in sequence (interval head), and have the following formula for an edge originating in any subsequent node b:

$$f_e^{\mathcal{R}}(z) = f_e(f_{e_1}^{\mathcal{R}}(z) + \ldots + f_{e_n}^{\mathcal{R}}(z)) \quad (1)$$

In formula (1), $e_1,\ldots,e_n$ are the edges in $\mathcal{R}$ which terminate in the node b.

Note, as a practical matter, that all the calculations that we have described may be carried out in parallel for the Boolean functions corresponding to a given region and block, and for any number of distinct items, provided only that the relevant Boolean values for each of the items in question are assigned particular bit positions in a single word, and that we make use of the bit-parallel Boolean machine operations with which almost all computers are supplied.

We have seen above how to calculate the region function $f_e^{\mathcal{Q}}$ for any interval $\mathcal{Q}$. In the following paragraphs, we shall show that a general program graph may be decomposed into a disjoint collection of distinguished subsets called <u>maximal intervals</u>. The strongly connected subregions of these intervals

can be considered to form an abstract collection of "inner loops". By identifying each interval to a single point, we shall obtain a new graph $\mathcal{P}'$ from $\mathcal{P}$. The graph $\mathcal{P}'$ is called the derived graph of $\mathcal{P}$. Iterating this construction, we will obtain higher derived graphs $\mathcal{P}''$, $\mathcal{P}'''$, etc. In many cases, this sequence of derived graphs will ultimately converge to a graph consisting of but a single node; programs whose graphs have this property may be considered to be completely regular structures made up of nested sets of iterative loops. A program graph containing no intervals consisting of more than a single point, and hence identical with its derived graph, may be called an irreducible program graph. We shall, in the present section, develop modified algorithms for handling irreducible program graphs by a procedure called "node splitting." Taking all in all, we shall develop an algorithmic procedure for the solution of the Boolean redundancy equations based upon separate calculations for disjoint intervals, and successive calculations for nested intervals. Because of the nested interval structure on which our algorithm rests, the calculation corresponding to the algorithm will grow only slowly, both in regard to computations performed and in regard to storage required, with increasing size of the program being optimized.

The following example will illustrate the principal notions introduced above. Consider the graph shown in Figure 1 below:

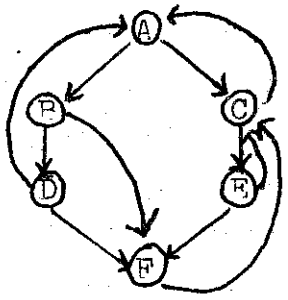


Figure 1. An illustrative program graph.

Node A is the entry node of the graph shown in Figure 1. The sets $\{A,B\}$, $\{A,B,D\}$, $\{A,B,D,F\}$ are all intervals, and A is the head of all of these intervals. The node C belongs to the interval $\{C,E\}$. The maximal intervals of the program graph shown in Figure 1 are $\{A,B,D,F\}$ and $\{C,E\}$.

If J is an interval, all the points of J which are immediate predecessors of the head of J are said to belong to the strongly connected subpart of J. In Figure 1, $\{A,B,D\}$ is the strongly connected subpart of the maximal interval $\{A,B,D,F\}$; the node F belongs to the maximal interval, but not to its strongly connected subpart. More generally, a set L of nodes in a program graph is said to constitute a strongly connected region if any node of L can be reached from any other node of L along a track lying wholly in L. Since any node of an interval can be reached from its head along a track lying within the interval, it follows at once that the strongly connected subpart of an interval is a strongly connected region. In Figure 1, $\{A,B,D\}$, $\{C,E\}$, $\{C,E,F\}$, as well as the whole graph constitute strongly connected regions.

Let us now begin a systematic investigation of the intervals of a program graph. Let $p \in \mathcal{P}$. We define a family of sets $S_n(p)$ by putting $S_0(p) = \{p\}$ (the set consisting of the single point p) and by putting

$$(2) \qquad S_n(p) = \{q \in \mathcal{P} \mid \tau^{-1}(q) \subseteq S_0(p) \cup \ldots \cup S_{n-1}(p)\} ,$$

inductively.

We let I(p) be the union of all the sets $S_n(p)$. The set I(p) is called the interval of the point p. We now note the following facts.

(a) If $q \in I(p)$, there exists a track in $I(p)$ reaching q from p. This is obvious from the definition of a program graph.

(b) If $q \in I(p)$, and $q \neq p$, any track from a point in $\mathcal{P}$ but not in $I(p)$ which reaches q must include p. (This is the single entry property of an interval.) Indeed, a track reaching a point q in $S_n(p)$ must, by formula (2), consist either of the single point q or must contain a point in $S_0(p) \cup \ldots \cup S_{n-1}(p)$. Thus our assertion follows readily by induction on n.

(c) $I(p) - p$ is loop-free. This assertion may be proved as follows. For each q in $I(p) - p$, define the level number $L(q)$ at the smallest integer n such that $q \in S_n(p)$. Then if $q \in \tau(q_1)$, and if $q_1 \in I(p)$, we have $q_1 \in S_0(p) \cup \ldots \cup S_{n-1}(p)$ by formula (2), unless of course $q = p$. That is, $L(q) > L(q_1)$ if $q \in I(p) - p$. Thus, the level number always increases along a track in $I(p) - p$. Consequently, $I(p) - p$ contains no closed track, and therefore is loop free, proving our assertion.

(d) If $q \in I(p)$, then $I(q) \subseteq I(p)$. Indeed, let $q \in S_m(p)$. Then it follows inductively from (1) that $S_n(q) \subseteq S_{n+m}(p)$, from which our assertion is obvious.

Next we shall show that any program graph may be decomposed uniquely into a union of disjoint maximal intervals.

Lemma.1. Let $p_1, p_2 \in \mathcal{P}$. Suppose that $p_1 \notin I(p_2)$ and that $p_2 \notin I(p_1)$. Then the intervals $I(p_1)$ and $I(p_2)$ are disjoint.

Proof: Suppose that our lemma is false, and let $q \in I(p_1)$ intersect $I(p_2)$. If $q \neq p_1$ and $q \neq p_2$, it follows from formula (2) that all the points of $\tau^{-1}(q)$ also belong to $I(p_1) \cap I(p_2)$. But these points all have a diminished level number in the sense of the proof of (c) of the preceding paragraph. It follows by induction on the level number that either $p_1 \in I(p_2)$ or $p_2 \in I(p_1)$, contradicting our hypothesis.

Q.E.D.

Using the preceding lemma, we may readily define a canonical decomposition of any program graph into intervals. We define sets B_n and D_n of nodes and C_n of intervals inductively as follows:

$B_0 = \{p_0\}$ (set consisting of entry node alone)

$C_n = \{I(p) \mid p \in B_n\}$

D_n = union of all points in the intervals belonging to $C_0, \ldots, C_n$

$B_{n+1} = \tau(D_n) - D_n$ (immediate successors to D_n not contained in D_n).

We let $\mathcal{C}$ be the union of all the sets C_n of intervals. It follows readily by induction that D_n includes $\tau^n(p_0)$. Thus, since every point in a program graph can be reached from the entry point of the graph, the whole graph is contained in the union of the sets D_n. From this, the following statement is plain:

Lemma 2. Every point in a program graph belongs to one of the intervals of $\mathcal{C}$.

Next, we note the following fact:

Lemma 3. The sets of $\mathcal{C}$ constitute a disjoint partition of the whole program graph $\mathcal{P}$.

Proof: If p may be reached from q along a track, we say that q is a predecessor of p and that p is a successor of q. Let $p \in B_{n+1}$ for some $n \geq 0$. Since the graph entry point p_0 is not a successor of p, $I(p_0)$ and $I(p)$ are disjoint by Lemma 1. Thus, $I(p)$ and D_0 are disjoint. We shall prove by induction that $I(p)$ and D_m are disjoint for all $m \leq n$. Suppose this assertion true for m-1. Then, if $p_1 \in B_m$, p_1 has

a predecessor in B_{m-1} by definition. Therefore, $p_1 \not\in I(p)$. Plainly, $p \not\in I(p_1)$. Thus, $I(p)$ and $I(p_1)$ are disjoint by Lemma 1. Hence, $I(p)$ and D_n must be disjoint, completing our induction. Thus, the intervals of C_{n+1} and the intervals of C_m, $m \neq n+1$, are disjoint. On the other hand, if p and q are distinct points of B_{n+1}, they both have predecessors in B_n, and, by what has just been proved, these predecessors do not lie either in $I(p)$ or $I(q)$. It follows that $p \not\in I(p)$, $q \not\in I(q)$, so that by Lemma 1, $I(p) \cap I(q) = \emptyset$. This completes the proof of the present lemma. Q.E.D.

We also wish to note that the interval-defining process described by formulae (2) and (3) above is algorithmic and corresponds to the following simple flow analysis algorithm:

a) Start with the entry node p of the program graph $\mathcal{P}$; put it in the set I, and designate p as the head of I.

b) Iteratively add to I every node all whose immediate predecessors already belong to I and continue this process until I cannot be extended any further. This defines an interval I with head p.

c) Add all the nodes belonging to I to a collection of "nodes already processed." If there are any nodes in the program graph not belonging to this set, choose one such node p which is an immediate successor of a node already processed, put it in a new set I, and designate it as the head of I. Then apply process b) above to extend I to its maximal size, etc.

d) Eventually every node of $\mathcal{P}$ will have been processed. At this point, the set of intervals constructed by our algorithm will be precisely the set $\mathcal{P}$ defined above.

Note that, for the example shown in Figure 1 above, this process would first construct the interval $\{A,B,D,F\}$ and then the interval $\{C,E\}$. These two sets are in fact the unique maximal subintervals of a program graph shown in Figure 1.

We now study the reduction of program graphs by means of the identification of subregions of the graph to single nodes. Let $(\mathcal{P},\tau)$ be a program graph, and let $\mathcal{J}$ be some decomposition of the nodes of $\mathcal{P}$ into a family of disjoint sets. If $J \in \mathcal{J}$ we let $\tau_1(J)$ be the set of all elements of $\mathcal{J}$, distinct from J, which also contain a point $\tau(p)$ with $p \in J$. We suppose that (as would be the case if $\mathcal{J} = \mathcal{C}$) no point in the set J_0 containing p_0 has a predecessor not in p_0. Then the pair $(\mathcal{J},\tau_1)$ is also a program graph. As entry point $(\mathcal{J},\tau_1)$ we take the particular set J_0 belonging to $\mathcal{J}$ which contains the entry point p_0 of the program graph p. It is clear from the corresponding property of $(\mathcal{P},\tau)$ that every set of $\mathcal{J}$ can be reached from J_0 along a track of the graph $(\mathcal{J},\tau_1)$. Moreover, J_0 has no predecessor in $\mathcal{J}$. We call the graph $(\mathcal{J},\tau_1)$ the <u>derived graph</u> of the graph $(\mathcal{P},\tau)$, <u>according to the decomposition family</u> $\mathcal{J}$. In the special case in which $\mathcal{J} = \mathcal{C}$, i.e., in which we take the decomposition of a program graph into the <u>intervals</u> defined above, the graph $(\mathcal{C},\tau_1)$ is simply called the <u>derived graph</u> of the graph $(\mathcal{P},\tau)$; we shall generally denote the derived graph by the symbol $(\mathcal{P}',\tau')$. The derived graph $\mathcal{P}'$ will, in turn, have a derived graph of its own, which is the <u>second derived graph</u> $(\mathcal{P}'',\tau'')$ of the original graph $(\mathcal{P},\tau)$. Proceeding inductively in this fashion, we define an n-<u>th derived graph</u> $(\mathcal{P}^{(n)},\tau^{(n)})$ of the graph $(\mathcal{P},\tau)$. This sequence of derived graphs gives a coarser and coarser way of looking at the original graph $(\mathcal{P},\tau)$ in which larger and larger nests of loops are identified to a point. If the graph $(\mathcal{P},\tau)$ admits any interval of length > 1, its derived graph has fewer nodes than $\mathcal{P}$. The contrary case is that of an <u>irreducible graph</u>, in which every interval consists of just one point, so that $(\mathcal{P}',\tau')$ and $(\mathcal{P},\tau)$ are isomorphic. It is clear that the sequence $(\mathcal{P}^{(n)},\tau^{(n)})$ of successive derived graphs of a graph eventually reach an irreducible graph; this irreducible graph is called the <u>limit graph</u> of the original graph $\mathcal{P}$ and may, where convenient, be

written $(\mathcal{P}^{\infty}, \tau^{\infty})$. A graph $\mathcal{P}$ whose limit graph is a single point, is called a fully reducible graph. It is this class of graphs to which our procedure of successive interval decomposition applies most effectively. The majority of program graphs occurring in actual practice are fully reducible. On the other hand, irreducible program graphs do sometimes occur. The following figure gives a number of examples of irreducible program graphs.

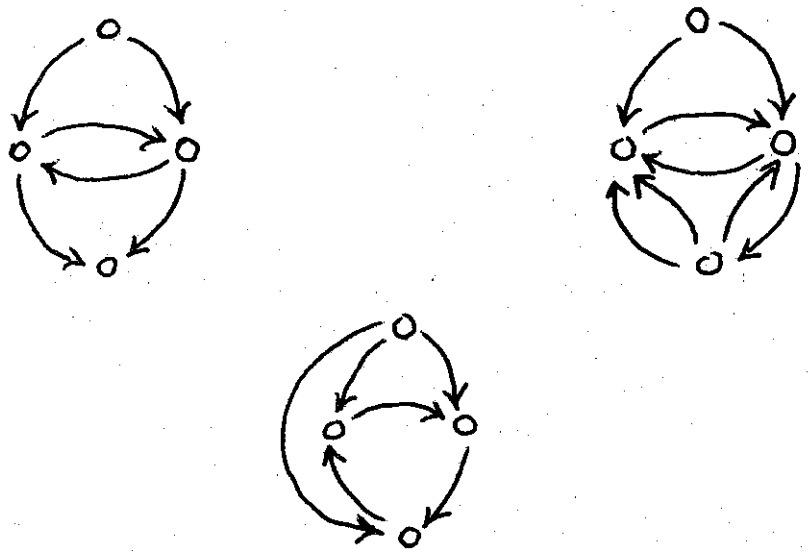


Figure Various irreducible program graphs.

If, in order to become more familiar with the intuitive significance of the formal notions just defined, we consider the graph shown in Figure 2 below, we see that the maximal intervals of this graph, which may be found by the algorithmic procedure defined above, are I1 = {A} , I2 = {B}, I3 = {C,D,E,F,G}.

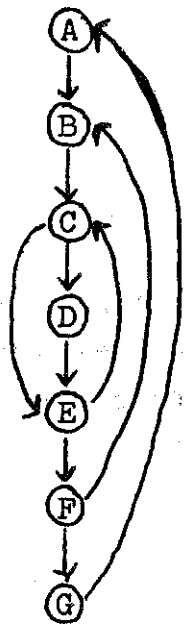


Figure 2. A program graph representing 3 nested loops.

The derived graph of the graph shown in Figure 2 is represented in Figure 3 below. Note that in passing from the graph of Figure 2 to a derived graph we have collapsed the "innermost loop" of the graph shown in Figure 2 to a single node; this inner loop may in fact be defined as the strongly connected portion of the maximal interval I3.

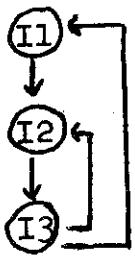


Figure 3. Derived graph of the graph of Figure 2.

The graph of Figure 3 may be analyzed into intervals using the same method; it consists of the two maximal intervals J1 = {I1} and J2 = {I2,I3}. Its derived graph has structure shown in Figure 4 below.

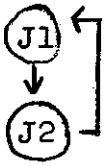


Figure 4: Second derived graph of the graph of Figure 2.

Note that, in this second derived graph, two levels of nested loops have been collapsed to a single point. The graph shown in Figure 4 consists of one single maximal interval, and its derived graph therefore consists of one single node.

If a program graph $\mathcal{P}$ is fully reducible, the observations made in the first paragraphs of the present section may be combined to the properties of the region functions deduced in the preceding section (c.f. formulae (5) through (9) and the associated discussion) to give an efficient systematic method for the solution of the Boolean redundancy equations associated with the graph $\mathcal{P}$. First of all, if R is an interval of $\mathcal{P}$, the functions f_e^R can, as we have seen, be calculated directly. The points r of the derived graph $\mathcal{P}'$ correspond to the maximal intervals of $\mathcal{P}$, and the edges of e of $\mathcal{P}'$ correspond to the edges of $\mathcal{P}$ which originate in one maximal interval of $\mathcal{P}$ and terminate in another. If we supply $\mathcal{P}'$ with edge transformation functions f'_e by setting $f'_e = f_e^R$, where R is the interval of $\mathcal{P}$ containing the origin of e, then, as shown in the concluding paragraphs of the preceding section, the system of Boolean redundancy equations for the edges of $\mathcal{P}$ which connect disjoint maximal intervals are identical with the Boolean redundancy equations for the derived graph $\mathcal{P}'$. Note also, for essential use below, that if we are able to calculate the solution x_e of the Boolean redundancy equations for edges e of $\mathcal{P}'$, which is to say for the inter-interval edges of the original graph $\mathcal{P}$, then, using the region functions f_e^R we can readily calculate the value of x_e for every edge. Indeed, if R is a maximal interval, and p is its head, let $y_p = \Sigma x_e$ the sum being extended over all edges e originating in an interval other than R and terminating in R, i.e., terminating in p. Next, let e be an edge connecting two nodes of R. It follows by the observations made in the preceding section (c.f. formulae (5) through (9) and the associated discussion) that $f_e^R(p)$ gives the required Boolean value x_e for the edge e.

Thus the problem of solving the Boolean redundancy equations for the graph $\mathcal{P}$ reduces to that of solving the corresponding equations for the derived graph $\mathcal{P}'$. But these equations may be solved by an iterative application of the method just described. More specifically, we

i. Find the maximal intervals R' of $\mathcal{P}'$, and for each such R' and each edge of the second derived graph $\mathcal{P}''$ which originates in R', calculate the region function $f_e^{R'}$ by the elementary method outlined in the first paragraphs of the present section;

ii. Use the region functions $f_e^{R'}$ to define edge transformation functions in the second derived graph $\mathcal{P}''$ of $\mathcal{P}$;

iii. Iterate this construction as often as necessary, defining edge transformations for all the derived graphs $\mathcal{P}^{(n)}$ of ;

iv. If $\mathcal{P}$ is fully reducible, some first one $\mathcal{P}^{(n)}$ of the successive derived graphs of $\mathcal{P}$ will consist of a single interval. For this graph, the Boolean redundancy equations may be solved directly by the elementary method outlined in the first paragraphs of the present section;

v. Once this is done, the region functions f_e^R, $f_e^{R'}$,..., already calculated may be used to obtain solutions x_e of the Boolean redundancy equations belonging to the edges e of the graphs $p^{(n-1)}, p^{(n-2)}, \ldots,$ in turn.

The method just described leads to a solution of the Boolean redundancy equations for every fully reducible program graph $\mathcal{P}$. We shall now generalize this method, deriving an algorithm for the efficient solution of the Boolean redundancy equation for every program graph, whether fully reducible or not. To this end we require some additional information concerning the intervals of a program graph: specifically we shall have to use the fact that the intervals I(p) defined by the essentially algebraic procedure outlined in the paragraphs surrounding formula 2 may also be defined by an intrinsic property. The two lemmas which follow establish this assertion.

Lemma 4. If J is an interval with head p, then J is contained in I(p).

Proof: Write $q_1 > q_2$ if q_1 and q_2 belong to J, $q_1 \neq p_1$, and q_1 is a successor of q_2. Since J-{p} is loop-free, this definition establishes a partial order on J, having the property that $q > q$ is always false. Next, define a level number L(q) for each $q \in J$ as follows. Put L(q)=0 if q = p, and, inductively, put

$$L(q) = \max_{\mathcal{I}(q_1)=q} L(q_1) + 1 \tag{4}$$

where $\mathcal{I}$ is the immediate successor function in the graph . We shall shown inductively that

$$\{q \mid L(q)=n\} \subseteq S_n(p) , \tag{5}$$

where $S_n(p)$ is as in formula (2) above. Indeed, this is plain for n = 0. Assuming inductively that it is true for n-1, it follows from (2) that it is true for n. This makes it clear that J is contained in the union of all the sets $S_n(p)$, which is to say that $J \subseteq I(p)$. Q.E.D.

Lemma 5. Any interval J of a program graph is contained in some one of the intervals I(p) of the family e; thus theintervals I(p) of the family e are the maximal intervals of .

Proof: Let q be the head of J. By Lemma 2, q belongs to some interval I(p) of the family e. Thus, by d) of the second paragraph preceeding lemma 1, $I(q) \subseteq I(p)$. But, by Lemma 4, $J \subseteq I(q)$. Thus $J \subseteq I(p) \in \mathcal{L}$ Q.E.D.

We now return to perfect our discussion of the solution of the Boolean redundancy equations. We wish to describe a method for the solution of the redundancy equations even for irreducible graphs. Our method, which from the algebraic point of view may be called the introduction of extra variables and from the geometric point of view may be called the method of node splitting, will be inductive. To describe the method, we require a number of auxiliary notations. If e is an edge, let b(e) be the origin node of the edge; if b is a node, let E(b) be the set of edges which terminate in b. The boolean redundancy equations may be written in a form explicitly showing the role of the nodes of the program graph as follows:

$$(6)\qquad \begin{aligned} x_e &= f_e(y_{b(e)}) \\ y_b &= \sum_{e \in E(b)} x_e \ . \end{aligned}$$

Suppose that we partition the set of all edges of the program graph into (possibly overlapping) sets $F_1,\ldots,F_n$, and, at the same time, introduce new variables x_e^j and y_b^j, $j = 1,\ldots,n$ for each of the variables x_e and y_b occuring in (6). Then it is clear from the linearity of all the equations involved that the system (6) may be replaced by the modified system

$$(7)\qquad \begin{aligned} x_e{}^j &= f_e(y_{b(e)}^j) \\ x_e &= \sum_{j=1}^{n} x_e^j \\ y_b^j &= \sum_{e \in F_j \cap E(b)} x_e \end{aligned}$$

of equations. Moreover, a slightly closer examination, which we leave to the reader, will show that the minimal solution of the equations (6) corresponds to the minimal solution of the equations (7) and vice-versa. Thus, for our purposes, solution of the equations (6) and of the equations (7) are fully equivalent.

By substituting the second equation (7) in the third of these equations, we can rewrite (7) in yet another fully equivalent form as

$$
\begin{aligned}
x_e^j &= f_e(y^j_{b(e)}) \\
y_b^j &= \sum_{i=1}^{n} \sum_{e \in F_j \cap E(b)} x_e^i \, .
\end{aligned}
\tag{8}
$$

The equations (8) permit of a simplification which will be of practical importance in what follows. Suppose, in order to describe this simplification, that the set $F_j \cap E(b)$ is null, so that the corresponding variable y_b^j in (8) has the value 0. Then for each edge e for which $f_e(y) \equiv 0$ or $f_e(y) \equiv y$, we plainly have $x_e^j = 0$, $j = 1,\ldots,n$. It is less obvious that, if we put $x_e^j = 0$ in (8) whenever $F_j \cap E(b(e)) = \emptyset$, we do not change the solution of (8) in any essential way. To see this, note that if in this case $x_e^j = 1$ in (8), we must have $f_e(y) \equiv 1$, so that $x_e^i = 1$ for every i. Thus, if we replace x_e^j by 0 in (8) whenever $F_j \cap E(b(e))$ is null, we neither change the value of any sum $x_e = \sum_{i=1}^{n} x_e^i$ nor the value of any quantity y_b^k. However, using this last observation, we may omit every variable y_b^j for which $F_j \cap E(b) = \emptyset$ and every variable x_e^j for which $F_j \cap E(b(e)) = \emptyset$ from the equations (8). Of course, one variable $y^i_{b_0}$ corresponding to the entry node b_0 of the program graph ought to be retained, together with all the associated variables x_e^1 for which $b(e) = b_0$.

Making the above reductions, we obtain the following reduced set of equations, fully equivalent for all our purposes to the initial set of equations (6):

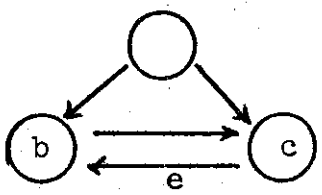
$$
\begin{aligned}
x_e^j &= f_e(y^j_{b(e)}), \quad F_j \cap E(b(e)) \neq \emptyset \text{ or } j=1 \text{ and } b(e)=b_0; \\
y_b^j &= \sum_{i} \sum_{e \in F_j \cap E(b)} x_e^i \, , \quad F_j \cap E(b) \neq \emptyset \, .
\end{aligned}
\tag{9}
$$

In the preceding paragraphs we have associated with each system (6) of equations a schematic representation of this system by a program graph, and it is interesting to describe the transformation of graphs that corresponds to the algebraic transformation (6) → (9). Geometrically, this transformation may be described as follows:

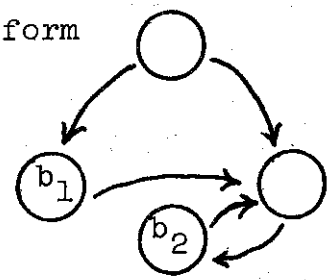
(a) Begin with a program graph P. Divide all the edges of the graph into a family $F_1,\ldots,F_n$ of (possibly overlapping) sets. For each node b of the program graph, and for each class F_n containing an edge terminating in b, introduce a "split node" b_j. The nodes b_j are the nodes of the "split graph" P^F.

(b) Introduce an edge e connecting the split node b_j of P^F to the split node C_k if, in the graph P, there exists an edge e' connecting b to c and belonging to the set F_j. Define the edge transformation f_e by putting $f_e \equiv f_{e'}$.

For example, consider the following irreducible graph



and partition its edges into two sets F_1, F_2 by including the edge e in F_2 while all other edges are put into the set F_1. It follows according to the above description that the node b splits into two nodes b_1 and b_2, b_1 receiving the input edge e, and b_2 receiving the other edge originally going to b; the split graph has the form



This latter graph, however, is fully reducible, so that the corresponding system of Boolean redundancy equations may be solved by the methods described earlier in the present section.

We shall now show that the same process can be applied to an arbitrary irreducible program graph P to define a split graph P^F whose derived graph has fewer nodes than does P, and hence, iteratively, to reduce any program graph to a single node, thereby giving a complete solution to the Boolean redundancy equations for the original graph. We proceed as follows. Let b be a node of P, different from the entry node of P and having as few predecessors as possible. (Note that, since P is irreducible, b has at least 2 predecessors.) Enumerate the edges terminating in b as $e_1,\ldots,e_n$, and let the origin nodes of these edges be $c_1 \ldots c_n$. Let $F_1=\{e_1\},\ldots F_{n-1} = \{e_{n-1}\}$, and let F_n be the set of all remaining edges of P. This defines a partition F of the set of all edges of P and hence defines a split graph P^F. It is clear from the above definitions that the node b is represented by n separate nodes $b_1,\ldots b_n$ in P^F, while every other node of P is represented by a simple node in P^F. It is also clear from these same definitions that the unique predecessor of b_1 in P^F is c_1, the unique predecessor of b_2 in P^F is c_2, etc. Thus, by Lemma 5, b_j and c_j belong to the same maximal interval of P^F for all $j = 1,\ldots,n$. It follows that the derived graph $(P^F)'$ has at least 1 node less than P, completing the proof of the above assertion.

The above observations lead to an algorithm for the solution of the Boolean redundancy equations belonging to the most general program graph $\mathcal{P}$. The overall structure of the algorithm is as follows.

i. Construct the maximal intervals R of $\mathcal{P}$ and form the derived graph $\mathcal{P}'$; to an edge e of $\mathcal{P}'$ originating in an interval R assign the region function f_e^R, thereby defining edge transformation functions for the derived graph. Iterate this procedure, calculating the successive derived graphs $\mathcal{P}'$, $\mathcal{P}''$, $\mathcal{P}'''$,... of $\mathcal{P}$ as long as these graphs contain

nontrivial intervals; iteratively associate edge transformations with the edges of these graphs. This procedure leads ultimately to an irreducible limit graph $\rho^{(n)} = \rho_{(1)}$ consisting either of a single node or of several nodes.

ii. If $\rho_{(1)}$ consists of a single node, then, as we have already seen, the Boolean redundancy equations for the graph $\rho^{(n-1)}$ may be solved in elementary fashion and this solution may readily be used to calculate the solution to the Boolean redundancy equations for all of the graphs $\rho^{(n-2)}, \rho^{(n-3)}, \ldots, \rho', \rho$. If $\rho_{(1)}$ consists of more than one node, we use the node-splitting procedure described above, introducing a partition F of the nodes of $\rho_{(1)}$ such that $(\rho^{F}_{(1)})'$ has fewer nodes than ρ and defining the edge transformations of $\rho^{F}_{(1)}$ from those of $\rho_{(1)}$ in the manner indicated in the previous paragraphs. As has been seen above (c.f. equations 1-5 and the associated discussion), a solution of the Boolean redundancy equations for $\rho^{F}_{(1)}$ leads immediately to solution of the same equations for $\rho_{(1)}$.

iii. Starting with $\rho_{(1)}$ and iterating the steps i) and ii) as often as necessary, we come through a sequence of graphs with fewer and fewer nodes. ultimately to a graph consisting of one single interval. For this graph, the Boolean redundancy equations may be solved by elementary means. Once this is done, the region functions already calculated may be used to obtain the solutions of the Boolean redundancy equations for all the preceding graphs in turn, and thus, <u>in fine</u>, for the original graph ρ.

It is to be noted that an empirical survey of program graphs shows the great majority of them either to be fully reducible or to have a limit graph consisting of a very few nodes. Thus the basic step i) of the procedure outlined above will in many cases serve for the complete solution of the Boolean redundancy equations. Even if this is not the case, the steps ii) and iii) will not normally have to be applied very often to solve the redundancy equations.

The section which follows contains a considerably more detailed account of the algorithm sketched and outlined above.

6.9 Optimization by Reduction in Strength. Introduction.

Program optimization by code motion, as described in the preceding section, is constrained by the condition that an instruction may not be moved either past a definition of one of its operands nor past a labelled entry point leading back to a definition of one of its operands. Reduction in strength is a general device permitting these relatively severe constraints on code motion to be relaxed considerably, and allows us to move an instruction out an inner loop even in the above adverse circumstances. However, the instruction moved must be replaced by another instruction (or sequence of instructions) lying in the loop. This replacement may, however, be an instruction which can execute faster than the one removed. The net gain in efficiency attained is then the difference between the execution times of the moved instruction and its replacement.

The general possibility of replacing an instruction by a faster instruction depends on the fact that in many situations the successive values of a function may be calculated more rapidly than would otherwise be possible if each value of the function is used to calculate the next value. Thus, multiplication by successive integers, or more generally by integers forming an arithmetic sequence, may be replaced by a succession of additions; exponentiation to successive powers may be replaced by a succession of multiplications, etc. To take a far-fetched but perhaps more illustrative example, suppose that, in a certain high frequency region of a program, we must calculate the successive values of a function f, known to be smooth. If the direction calculation of f(J) were known to be slow, we might choose to adopt the following modified procedure.

1. Calculate the values f(J) for J = 0, J = N, J = 2N, ..., J = KN, ..., directly. At each such step of direct calculation, retain the previously calculated values f(N*K-N) and f(N*K-2*N).

2. Use the three function values f(N*K), f(N*K-N), f(N*K-2*N) to calculate first and second differences suitable for subsequent use in obtaining the remaining values of f(K) by interpolation.

3. Calculate any additional value f(N*K+J), J=1,...,N-1, as required by quadratic interpolation.

The procedure which is outlined above is in fact one of the principal classical methods for reducing the amount of labor required in the calculation of mathematical tables. A considerable theory of the accuracy of the above reduction and other related more general approximations is available. All such classical approximations are of course available for use in connection with computer programs; they will, in many cases, enhance the efficiency of a computer program considerably at little loss in accuracy.

A transformation of this sort may be regarded as a 'reduction in strength' which replaces the possibly lengthy series of steps needed to calculate a function f by the much simpler series of steps needed to interpolate between available values of f. Of course, an automatically acting optimizer algorithm cannot be expected to make use of reductions as sophisticated as the one that we have described above. Nevertheless, there exists a simpler related family of reductions, applicable especially to the standard code sequences generated by references to dimensioned arrays, which are quite suitable for automatic application. These strength reductions may be regarded as simple instances of the classical reduction described above.

The transformation of instructions which must be carried out during reduction in strength may be described in more specific detail as follows. Suppose that:

i. A transformation $y = f(x_1,...,x_n)$ (expressed possibly by a single instruction) occurs in a region R of code, and it is desirable to move this transformation from the point at which it occurs to a lower frequency section.

ii. The desired motion is blocked by redefinitions $x_j = h_j(x_1,\ldots,x_n)$ of the arguments of the transformation $y = f(x_1,\ldots,x_n)$ which is to be relocated.

iii. It is known that identities of the form $f(x_1,\ldots,x_{j-1},h_j(x_1,\ldots,x_n), x_{j+1},\ldots,x_n) = g_j(x_1,\ldots,x_n,f(x_1,\ldots,x_n))$ hold.

iv. The function g is simpler than the function f, in this sense that g may be evaluated more rapidly than f.

In this case we

1. Introduce a new variable t.

2. Replace the (macro) instruction $y = f(x_1,\ldots,x_n)$ by the statement $y = t$.

3. Replace each redefinition $x_j = h_j(x_1,\ldots,x_n)$ of an argument of f by the pair of statements $x_j = h_j(x_1,\ldots,x_n)$, $t = g_j(x_1,\ldots,x_n,t)$.

4. Place the initializing instruction $t = f(x_1,\ldots,x_n)$ at each point to which the (macro-) instruction $y = f(x_1,\ldots,x_n)$ would have had to be moved if no redefinitions $x_j = h_j(x_1,\ldots,x_n)$ were present in the loop containing $y = f(x_1,\ldots,x_n)$.

The following examples will illustrate the general transformation described above. Suppose that a section of code contains the iterative loop

(1)
```
          ...
          J = 1
          I = 1
LØØP:     X = CØS(FLØAT(J))
          ...
          J = 10 * I
          ...
          I = I + 1
          ...
          I = I + 2
          ...
          If (I.le.1000) go to LØØP
```

Reduction in strength of the instruction J = 10 * I transforms this sequence into

(2)

```
                ...
                J = 1
                I = 1
                t = 10 * I
LØØP:           X = CØS(FLØAT(J))
                ...
                I = I + 1
                t = t + 10
                ...
                I = I + 2
                t = t + 20
                ...
                If (I.le.1000) go to LØØP
```

Optimizations similar to the above may be employed in a wide variety of fixed and floating point situations. In the discussion that follows, we will always describe the fixed rather than the floating point version of a given reduction in strength. We do this because, whereas a fixed point reduction in strength replaces an algorithm by a strictly equivalent transformed algorithm, application of reduction in strength in floating point situations is only approximate, and, owing to the accumulation of round-off errors, may actually change the ultimate result of a calculation.

Before proceeding to a detailed theoretical discussion we shall give a somewhat more general second example of the specific strength-reduction with which we shall be primarily concerned. In describing this and later examples, a number of notational conventions will be convenient. We let I, J, K, etc., denote integers. We suppose that a region R of a program is given, out of which we aim to move instructions by the method of reduction in strength. We let IRC, JRC, etc., denote integer quantities which are region constants, that is, quantities which are not redefined within the region R out of which code is to be moved.

If a code contains an integer multiplication by a region constant, and the argument variable occurring is defined iteratively

and only iteratively within the region R with which we are concerned reduction in strength is still applicable. In this case, we have a code sequence

(3)
```
I = M * IRC
...
M = M + JRC
...
M = M + KRC
...
```

which may be replaced by a reduced code sequence

(4)
```
I = t
...
M = M + JRC
t = t + IRC * JRC
...
M = M + KRC
t = t + IRC * KRC
...
```

The initializing instruction t = M * IRC must then be moved out of the region R. As noted above, the application of this reduction is constrained by the condition that all definitions of the quantity M within the region R must be of the incremental form M = M + JRC. Note that in (4) the region independent multiplication operations IRC * JRC and IRC * KRC may also be moved out of the region R.

A reduction strength similar to the above is applicable in situations in which we calculate the product I = M * N of two integer variables, on the hypothesis that, within the region R out of which such a multiplication operation is to be moved, both arguments of the product are defined iteratively and only iteratively. However, instead of illustrating this more general reduction by an example at the present point, we now proceed to a general theoretical discussion of operator strength reduction.

To repeat what has been said above, the systematic procedure of reduction in operator strength, as it applies to a particular expression $f(x_1,\ldots,x_n)$ to be calculated within region R, may be defined as follows.

i. Within R and at all points within R, we maintain the current value of the expression $f(x_1,\ldots,x_n)$, keeping its value in some compiler-generated temporary location t set aside for this purpose.

ii. To each redefinition $x_j = h_j(x_1,\ldots,x_n)$ of an argument of f we append the associated transformation $t = g_j(x_1,\ldots,x_n,t)$ required to update the value of $t = (x_1,\ldots,x_n)$.

iii. We replace each use of the expression $f(x_1,\ldots,x_n)$ with the use of the value t.

iv. The value of t is initialized on entry to the region R by explicit calculation of the expression $f(x_1,\ldots,x_n)$.

The procedure described above is still insufficiently general in one essential regard. In order to insure that the arithmetic operations required to update the value of an expression $f(x_1,\ldots,x_n)$ are as simple as possible, it may be appropriate to keep available the current values of a number of auxiliary quantities $u = p(x_1,\ldots,x_n)$, $v = q(x_1,\ldots,x_n)$, etc. This generalized procedure will be of advantage whenever the availability of these values simplifies the updating procedure necessary to keep the value $f(x_1,\ldots,x_n)$ current. (Of course we require that the calculations needed to keep the values of the auxiliary functions $p(x_1,\ldots,x_n)$, $q(x_1,\ldots,x_n)$, etc. be simple also). Consider, in order to illustrate this point, the application of a strength reduction procedure to the expression i*j in a program region in which the only definitions of i and j are the iterative definitions i = i + 2 and j = j + 3. If t is a core location set aside to contain the current value of i*j, then the rules for updating t which must be inserted are as follows.

a) Insert t = t+2*j after each occurrence of i = i+2;
b) Insert t = t+3*i after each occurrence of j = j*3.

These rules involve calculation of the products 2*j and 3*i, and hence do not lead to a code which is more efficient than the original code in which i*j is to be calculated. To obtain a code sequence not involving multiplications, we must keep on hand current values of u = 2*j and v = 3*i. If this is done, the rules necessary to update all the quantities t, u, and v may be stated as follows:

a') Insert t = t+u and u = u+6 after each occurrence of i = i+2;
b') Insert t = t+v and v = v+6 after each occurrence of j = j+3.

To employ the strength reduction process described above, an optimizer must make corresponding transformations of the code sequences with which it is presented; of course, these transformations must be made during optimization, using only information available at that time. In order that this be logically possible, and in order that we may have reasonable grounds for expecting that the resulting code is more rather than less efficient than the original code, the code sequence originally given must satisfy various constraints.

1. At any point in the original code sequence in which a redefinition $x_j = h_j(x_1,\ldots,x_n)$ occurs, the current values of a set of expressions $u = p(x_1,\ldots,x_n)$, $v = q(x_1,\ldots,x_n)$, etc., sufficiently ample to allow the easy calculation of the updated value of $t = f(x_1,\ldots,x_n)$ must be available. In particular, no variable whose value cannot be predicted at optimization time may occur in any redefinition $x_j = h_j(x_1,\ldots,x_n)$. This observation makes it plain that no indexed variable with an index which may change in R can occur among $x_1,\ldots,x_n$; indeed, the occurrence of such

a variable introduces into the calculation of $h_j(x_1,\ldots,x_n)$ a value which is completely unpredictable at optimization time. Otherwise put, the occurrence in an expression of variable whose definition traces back directly or indirectly along paths of R to the value of an indexed variable with a changing index excludes consideration of this expression as a candidate for reduction in strength. Similarly, the occurrence in an expression of variable whose definition traces back directly or indirectly to a value read from an external medium excludes consideration of this expression as a candidate for reduction in strength.

2. A recalculation $t = g_j(x_1,\ldots,x_n,t)$ must be attached to each redefinition $x_j = h_j(x_1,\ldots,x_n)$ of an argument of $f(x_1,\ldots,x_n)$. If such a redefinition occurs in a strongly connected proper subinterval Q of the region R, insertion into Q of the recalculation $t = g_j(x_1,\ldots,x_n,t)$ is undesirable, since, generally speaking, the instructions in Q will be executed more frequently than those instructions belonging to R but not to Q. We conclude that the occurrence in an expression of the variable whose definition traces back directly or indirectly along paths of R to a definition lying in a strongly connected subinterval Q of R excludes consideration of the expression as a candidate for reduction in strength.

3. The values of the output arguments of any separatly programmed subroutine called at a point in a code sequence can normally be expected to be related within the body of the called subroutine either to an internal loop, or to the value of a variably indexed quantity, or at any rate to be calculated in a manner sufficiently complex as to make reduction in strength impossible. It is therefore reasonable for an optimizer to assume that the occurrence in a expression of a variable whose value is set by a separately programmed subroutine excludes consideration of this expression as a candidate for reduction in strength.

We may call an arithmetic assignment lying within R elementary if no expression occuring on the right-hand side of this assignment involves a value tracing back directly or indirectly along a path of R either to the value of an indexed variable with an index change in R, to a value read from an external medium, to a value set by a separately programmed subroutine, or to a value calculated within a strongly connected proper subinterval of R. All other assignments may be called nonelementary. The elementary and the nonelementary assignments belonging to R may be distinguished from each other by use of the following straightforward algorithm.

a) Using the method outlined in the preceding paragraphs, move as many calculations as possible out of R. This makes the region constants, i.e., the expressions not changing within R, apparent.

b) Take up each of the assignments in R, and flag each as being nonelementary if the assignment in question is made either by a separately programmed subroutine, an input routine, or by use of an expression containing any indexed variable whose index is not a known region constant, or if the assignment in question is made within a strongly connected proper subinterval of R.

c) Once an assignment has been flagged as nonelementary, find all the uses within R of this assignment and flag each such use as nonelementary.

d) Repeat step c) as long as it leads to new non-elementary assignments.

Those assignments lying in R which are not eventually flagged as nonelementary by use of a)-d) are elementary. If an expression occurring in R does not use any nonelementary definition in R, we may call this expression an elementary expression, or, more properly, an expression elementary in R. Only expressions elementary within R are candidates for reduction in strength in any sense whatever. However, it is by no means the case that every elementary expression

can be reduced in strength. To make explicit the additional criteria which must be satisfied if strength reduction is to be possible, we must examine the reduction process in somewhat closer detail. In order to apply strength reduction to simplify a calculation, we must

i) Define a set C of formal expressions $f(x_1,\ldots,x_n)$ whose values are to be kept current at all points within R. These are the expressions which are candidates for reduction.

ii) Define a set of "easy" calculations and make sure that all the calculations necessary at any point in R to update the values of the expressions belonging to C are "easy".

Thus, for example, in reducing the strength of products occurring within the region R, we wish to replace products by sums; additions but not multiplications are accordingly considered to be easy operations during this reduction. Suppose that an expression $f(x_1,\ldots,x_n)$ occurring at a given block of R is to be reduced in strength. If, tracing forward in R from this block, we come to a redefinition $x_j = h_j\ (x_1,\ldots,x_n)$ of one of the arguments of f, we must require that the substituted expression $E = f(x_1,\ldots,h_j(x_1,\ldots,x_n),\ldots,x_n)$ be expressible in terms of other currently available expressions using some set of easy calculations. If, tracing still further along paths in R, we come to yet another redefinition of an argument x_k of F, we must require that the expression obtained by replacing each occurrence of x_k in E by the expression occurring in the redefinition of x_k be obtained from the value of other currently available expressions by some easy set of calculations, etc. Continuing in this way, and assuming R to be strongly connected, we return eventually to the place in R from which we started. Depending on the extent of the internal branching within R there may of course be any number of simple circular paths in R leading us from our starting point back to the same point. When we traverse

any such circular path, those variables $x_1,\ldots,x_n$ not ruled out of consideration by their occurrence in a nonelementary assignment will undergo a composite substitution of some forms

$$x_j = \phi_j^{(1)}(x_1,\ldots,x_n)$$

or

$$x_j = \phi_j^{(2)}(x_1,\ldots,x_n)$$

or

$$x_j = \cdots$$

or

$$x_j = \phi_j^{(n)}(x_1,\ldots,x_n);$$

If the region R admits a large number of distinct elementary return paths there will in general, be many different possible composite substitutions. Any particular composite substitution is of course merely the resultant of all the elementary redefinitions of the variables x_j which occur along some particular simple circular path. If an expression $\phi(x_1,\ldots,x_n)$ occurring in at a given point in a program is to be reducible in strength, we must demand that there exist a finite set $E_1,E_2,\ldots,E_n$ of expressions including $f(x_1,\ldots,x_n)$ and having the following.

Closure property: If $x_j = \phi(x_1,\ldots,x_n)$ is any composite substitution, and $1 \leq \ell \leq n$,

$$(1)\quad E_\ell(x_1,\ldots,x_{j-1},\ \phi(x_1,\ldots,x_n),x_{j+1},\ldots,x_n)$$

must have an expression in terms of $E_1,\ldots,E_n$, this expression involving only easy calculations.

Study of an example will help us to understand the force of the above remarks.

Suppose that R is the program interval shown in Figure 1, and that, in addition to the conditional transfers corresponding to the edges shown in that figure, there occur only the operations and assignments explicitly indicated.

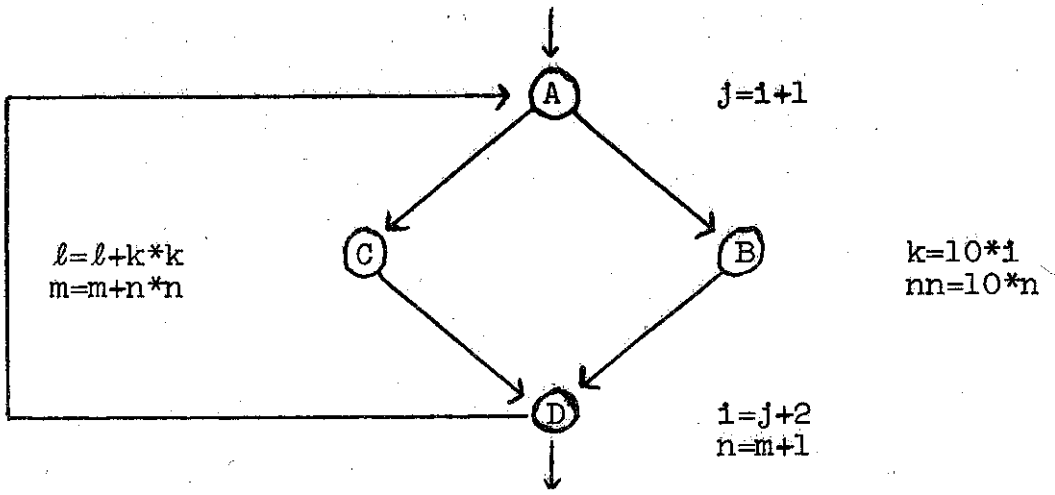


Figure 1: An example for the study of strength reduction.

We aim to reduce multiplications to additions wherever possible in the loop shown in Figure 1; the expression 10*i occurring in block B is a candidate for this reduction. To attempt a reduction, we first follow the substitutions made for i in any circular elementary path leading from block B back to block D; we find that only the composite substitution i=(i+1)+2=i+3 is made. It follows that strength reduction is possible; the values t_1=10*i and t_2=10*(j+2) must be kept available. The operation $t_1=t_2$ must be attached to the assignment i=j+2, and the operation $t_2=t_1+30$ to the assignment j=i+1. Once this strength reduction is made, the code shown in Figure 1 takes on the reduced form shown in Figure 2.

Note that code shown in Figure 2 is only superior to that shown in Figure 1 if block B is executed with substantially the same frequency as block A. If this is false, our purported optimization will in fact have degraded the speed of the code, since we will have replaced a rarely executed multiplication by a frequently executed addition. (On the other hand, the further case in which a loop like that shown in Figure 2 contains conditional branches on a loop -- independent condition, the methods of optimization applicable in such situations, and discussed in a later section of the present

chapter, apply and enable strength reduction to be used advantageously even in certain cases in which a multiplication to be reduced lies in a rarely executed block while an addition affecting it is frequently executed.)

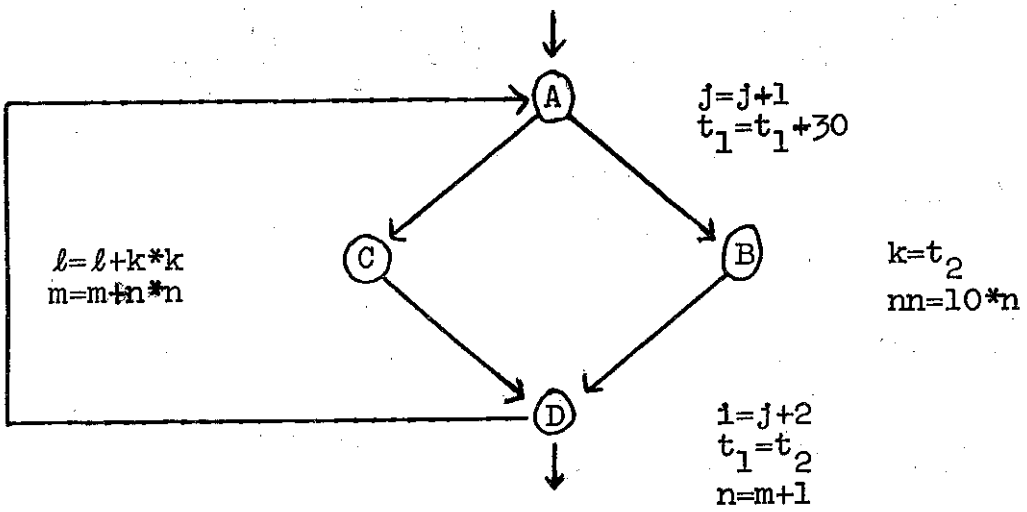


Figure 2: First stage of strength reduction.

Ignoring this issue for the time being however, we now go on to observe that the expression k*k occurring in block C of Figure 2 is a candidate for reduction in strength. The composite substitution $k=t_1+30$ is seen to apply to the variable k, so that we must also take account of any composite substitution applying to the variable t_1. Examining Figure 2, we see that there is only one such transformation, to wit, the transformation $t_1=t_1+30$. Reduction in strength is therefore possible if we maintain current values not only of $t_3=k*k$, but of $t_4=(t_1+30)*(t_1+30)$, $t_5=(t_1+30)*60+90$, and, as auxiliary quantities, $t_6=t_2*t_2$ and $t_7=t_2*60+90$. To the assignment $t_2=t_1+30$ occurring in block A of Figure 2, must append the assignments $t_6=t_4$, $t_7=t_5$; to $k=t_2$ in block A we must append $t_3=t_6$; and to $t_1=t_2$ we must append $t_4=t_6+t_7$ and $t_5=t_7+60$. Once these modifications have been made, the code of Figure 2 takes on the still further

reduced form shown in Figure 3. (In Figure 3 we have explicitly indicated the presence of an entry block E in which all necessary variable initializations can be placed).

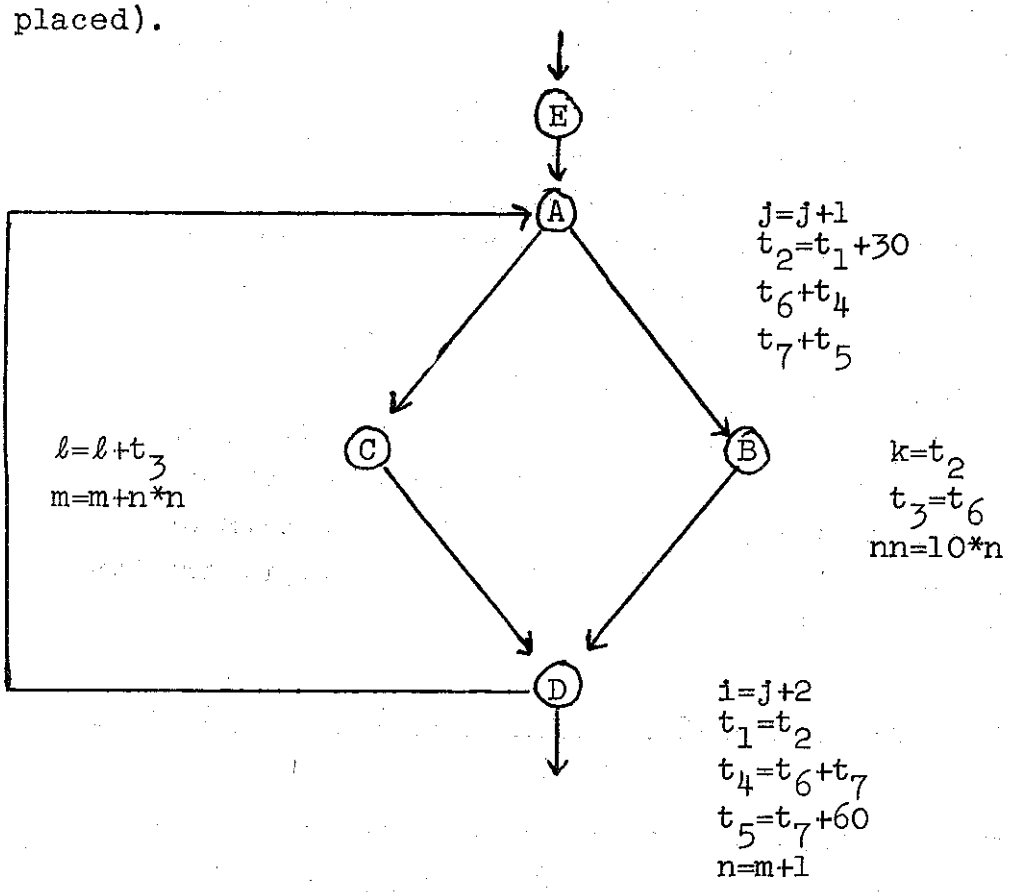


Figure 3: Second stage of strength reduction.

The next expressions to appear as candidates for reduction in strength are the expressions 10*n occurring in block B and the expression n*n appearing in block C. On inspecting Figure 3 we find that the composite transformations applying to the variable n are n=n+1 and n=n+2+n*n. The occurrence of this latter transformation implies that any set of expressions containing either 10*n or n*n and having the closure property described above must contain expressions of arbitrarily high degree. Hence no such set can possibly be finite. We conclude that neither the expression 10*n nor the expression n*n occurring in our loop can be reduced in strength.

While we intend to reserve systematic discussion of register and core location assignment to a later section of the present work, such assignment will have a substantial effect on the code shown in Figure 3; for this reason, it is appropriate to have a preliminary glance at this process here. A variable is <u>live</u> along any track in a program graph connecting a definition of the variable to a use of the variable and not passing through any subsequent definition of the same variable. (Compiler generated temporary variables such as t_1 - t_7 in the above example are of course live only along tracks internal to the region R in which they are generated). If two variables are never both alive along any track of a program graph, their values may be held in the same core location or register. Examining Figure 3, we see that the sets of edges along which the various quantities $t_1,\ldots,t_7$ are live, are as indicated in Table I.

Table I. <u>Live-dead pattern for variables in Figure 3</u>.

variable \ edge	EA	AB	BC	AD	DC	CA
t_1	X					X
t_2		X	X	X	X	
t_3	X		X	X	X	X
t_4	X					X
t_5	X					X
t_6		X	X	X	X	
t_7		X	X	X	X	

It is clear from Table I that t_1, t_4 and t_5 respectively are never live at the same time as t_2, t_6 and t_7 respectively. Hence, t_1 and t_2 may be assigned to a common register r_1; t_4 and t_6 to a common register r_2; and t_5 and t_7 to a common register r_3. Making this modification and indicating initializations explicitly, the code shown in Figure 3 takes on the final form shown in Figure 4.

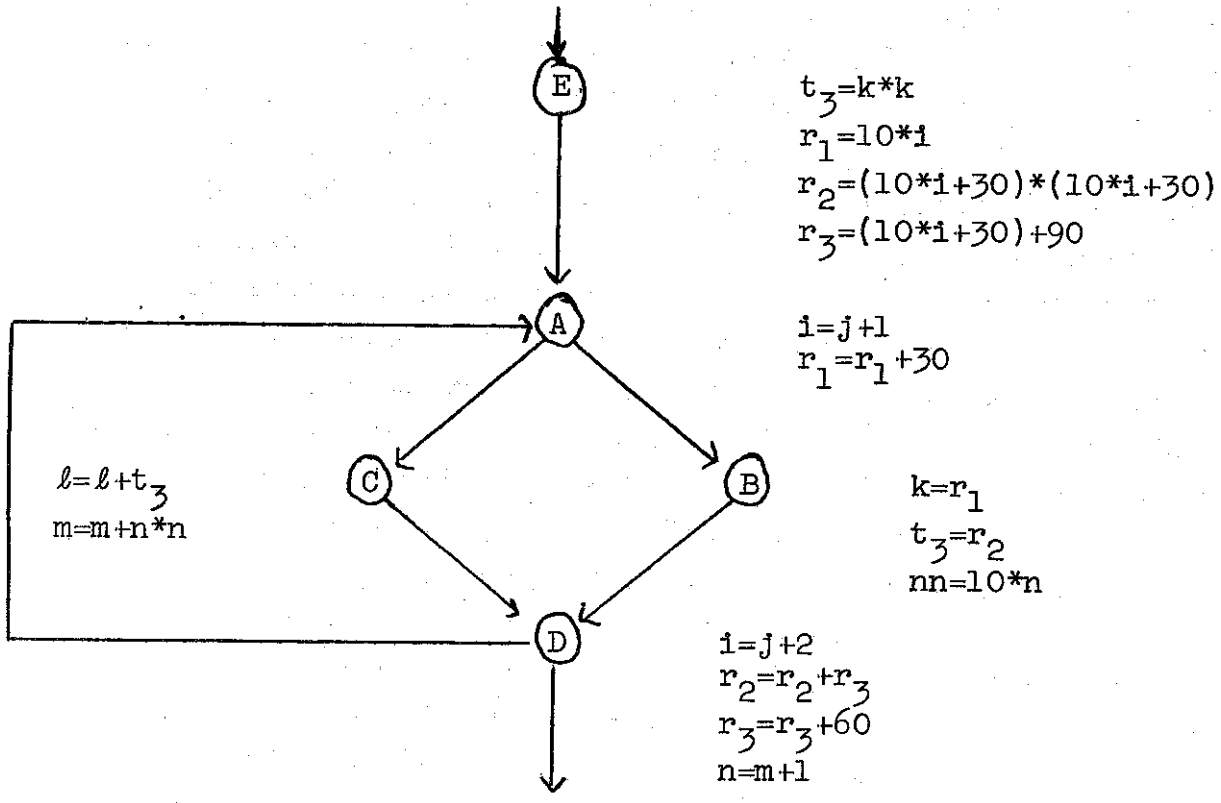


Figure 4: Reduced code after register overlay.

Comparing Figure 1 with Figure 4, we see that two multiplications present in our original loop have been replaced by three additions. Note also that the variable k has no use in the code shown in Figure 4; if k were known to be dead on exit from this loop, the code shown in Figure 4 could be further simplified. Note also that the variables i and j occurring in the loop have no use except to define each other. If our optimizer is able to detect this fact, and if i and j are dead on exit from block D, then still further simplification of our code is possible.

The most commonly applicable and hence the most useful operator strength reduction is the reduction of multiplication to addition; this transformation applies to various familiar code sequences generated during compilation of source code in which doubly or multiply subscripted arrays occur. For this reason a closer analysis of this particular strength reduction is in order. In reducing multiplications to additions we regard addition but not multiplication as an easy calculation. Thus we are able to reduce an expression $f(x_1,\ldots,x_n)$ in strength only if it can be included in a finite set $E_1,\ldots,E_n$ of expressions having the property that any expression (1) obtained from an expression E_ℓ by carrying out a composite substitution can be expressed as a simple sum in terms of the various expressions E_j and of region constant expressions. Otherwise put, we require that, modulo region constant terms appearing additively, the set of all expressions which can be obtained from f by iterated applications of the variables $x_1,\ldots,x_n$ of composite substitutions shall be included in a finitely generated additive group. The expressions $E_1\ldots E_n$ may then be taken as the generators of this group. In the most common case the expression $f(x_1,\ldots,x_n)$ will be a polynomial, and we demand that no composite substitution applying to a variable $x_1,\ldots,x_n$ raise the degree of the polynomial f or introduce new multiplicative factors to this polynomial. This will normally be the case if and only if every elementary assignment to a variable $x_1,\ldots,x_n$ appearing in the region R has one of the three allowed forms $x_i=\pm x_j\pm x_k$, $x_i=\pm x_j+$region constant, or $x_i=$region constant. Moreover, this criterion, requiring that only linear assignments be made within the region R to any of the variables x_i, is relatively easy to apply and hence relatively practical. If a set of variables $x_1,\ldots,x_n$ satisfies this condition, then any polynomial expression in these variables occurring in R can be reduced in strength. After this reduction is fully carried out, the value of the expression f will

be obtained via a sequence of assignments involving additions only. Thus, assignments originally involving f may come into a form permitting further reductions in strength. The full set of all expressions ultimately reducible in this way may be determined by application of the following iterative set of rules.

6.11 Subroutine Linkage Optimization -- Introductory.

In the present section, we initiate a discussion of the important question of subroutine linkage optimization.

In order to conserve code space, frequently used sections of code are often compiled in a single copy, and used in the normal transfer-and-return style as subroutines. To the extent that entirely separate subroutine compilation and frequent modification of subroutines is allowed, subroutine calling sequences, i.e., the patterns in which arguments are transmitted to and from subroutines, must be standardized. Since subroutines may call other subroutines in turn, standardization may be necessary if any subroutine in the whole library of precompiled subroutines used by a given program may be recompiled. On the other hand, the use of rigorously standardized linkages produces inefficiencies which may be considerable, in that such linkages may imply code full of the long sequences of linking operations and tables actually required in very general cases but potentially eliminable in frequently occurring special cases. Such inefficiencies may be particularly costly when a subroutine is called from within a critical inner loop of another program. In an attempt to describe methods by which these problems may be alleviated, we shall, in what follows, describe various possible forms of subroutine linkage more flexible thatn the normal, entirely standardized, linkage; such linkages could be employed to produce code efficient in regard to execution time (though sometimes requiring more storage space) than standard linkages. Note in this connection that the ability to use any given subroutine linkage depends not only on the construction of compilers capable of producing linkable code incorporating this linkage, but at a more encompassing level on the existence of a "linking loader" able to merge separate subroutines into a single executable body of code. Many interesting problems relating to the maintenance and use of subroutine libraries, the linking together of programs written in different source languages, the use of assembly language subroutines in connection

with source-language code, etc. relate intimately to the structure of the linking load program provided within a given operating system.

With the intent of supplying considerable additional details in the paragraphs which follow, we begin by distinguishing between four distinct types of subroutine linkage:

1. Fully closed linkage - This is the most familiar linkage form, making use of strictly standardized argument transmission and register use conventions. Note however that even in this case appropriate design of transmission and register use conventions can yield advantages.

2. Semi-closed linkage - In this form of linkage we require a subroutine S to inform its calling program R of the quantities and addresses which it expects to receive in registers, the quantities which it will return in registers, the registers which it will use, the global variables and arrays which it may affect, and to supply miscellaneous related information. This linkage information is then passed along to the latter stages of a register allocation process like that described elsewhere in the present chapter; during calling routine register allocation, the subroutine is treated as a program block within which certain register assignments have been made. In order to allow for subroutine modification even after calling program compilation, a record of the subroutine linkage information used during calling program compilation must be associated with the calling program; during final subroutine integration (linking and loading) before program execution, the continued validity of this information must either be verified, or instructions compensating appropriately for subroutine changes inserted.

Properly matched compilation of groups of subroutines in this semi-open fashion will require that called routines be compiled before their calling routines. If an entire collection of routines is to be compiled together, the compiler may have to trace the pattern of subroutine calls occurring in this set of routines and adjust its order of subroutine compilation accordingly.

Even though all these conditions must be satisfied if semi-open subroutine linkage is to be employed, it is still feasible to use semi-open linkage even for separately compiled and for library subroutines within an appropriately designed program loading system.

3. Open linkage - A subroutine S is said to be used as an open subroutine if the code constituting the body of S is explicitly inserted in place of a call to S. This type of subroutine linkage, which permits calling and called routines to be optimized together in fully integrated fashion, is maximally efficient in regard to time. However, since it may require the duplication of extensive sections of code, it may be prohibitively wasteful of space. Nevertheless, there are certain circumstances in which open linkage is feasible and even clearly preferable to other forms of linkage. If a subroutine is called only once in an entire program, an open linkage will be more efficient in regard both to time and to space than a closed linkage. More generally, if a subroutine of length ℓ is called n times, then if $nc \geq (n-1)\ell$, where c is the length of the subroutine calling sequence, open linkage is clearly preferable to closed linkage in terms both of time and of space. This observation will apply to various short subroutines of common use which compile to a very few machine instructions. Finally, it may be important to allow a programmer to specify that certain subroutine calls occurring within a crucial loop of a program are to be treated as open calls; the savings in time thus afforded may greatly outweigh the cost in space. Note also that joint optimization of a calling program and an open subroutine may afford subroutine simplifications through constant propagation that reduce the length of the subroutine very substantially.

If a library subroutine is to be called via an open linkage, the library must supply the subroutine either in original source language or in an intermediate language form sufficiently close to symbolic as to permit the symbol table references of the open subroutine to be merged with those

of the calling subroutine in a manner permitting subsequent optimization.

4. Semi-open linkage - To link a subroutine S to a calling routine using a semi-open linkage, we separate S into an initial portion S1 consisting of all those initial instructions of S not belonging to loops of S, and a remainder S2 consisting of the residue of S. S2 is then treated as if it were a separate subroutine called by S1 through a semi-closed linkage, and S1 is inserted in open style in place of any call to S. This permits S1 to be optimized together with the program calling S, but avoids the repetition of such extensive code passages as may be relatively unlikely because of contained loops to yield substantial improvement when optimized together with calling code rather than separately. We may note, much as in the case of open subroutines, that if semi-open linkage is used for S a symbolic or intermediate near-symbolic form for S1 is required; in additon, S1 and S2 must be perfectly matched to each other.

We now turn to a more detailed discussion of the various styles of subroutine linkage outlined above. Closed linkage, employing fixed argument transmission conventions, is simpler than semi-closed and semi-open linkage. However, even for closed linkage significant problems having to do with fundamental source language features must be faced. If a subroutine S is to be used and properly linked to its calling routine R, S and R must agree as to the location and structure of all information to be transmitted between them. If S and R are written in a language not permitting many different data types, it can be sufficient to transmit information in terms of its address only. If, on the other hand, one deals with language incorporating a considerable variety of data types, and in which the meaning of source language expressions depends upon the type of data to which these expressions are applied, it may be necessary to attach descriptors to data during the execution of code. In this case, transmission of data from a routine R

to a subroutine S will also involve transmission of descriptors, perhaps by transmission of descriptor locations, the descriptors themselves serving to locate the information structures which they describe. Unless care is taken in the design of such a language, the constant necessity to interpret descriptors may force all or nearly all code to be executed interpretively at a substantial cost in execution efficiency. If the descriptors attaching to particular variables can be determined at compile time, direct rather than interpretive program execution becomes possible. It is, of course, particularly difficult to preserve static descriptor information across subroutine linkages, and, if a language permits subroutines applicable at run time to operate on argument data with arbitrary descriptors this may be impossible. In response to this problem, one may simply allow subroutines to treat their arguments in fully interpretive fashion. Alternatively, one may restrict the type of argument to which a subroutine may be applied, insisting that matching argument declarations occur both in a subroutine and in the routine which calls it; this would preserve efficiency but at substantial cost in flexibility. The mixture of these two extreme techniques, in which one requires that some argument attributes be specifically pre-declared in matching fashion between a calling routine and its subroutines, while allowing other argument attributes to be dynamically transmitted through the use of descriptors, may be employed. Such a mixed subroutine convention should aim to provide flexibly usable subroutines while still avoiding the development of intolerable inefficiencies.

As a third possibility, one may interpret argument descriptors on subroutine entrance, and use this information to perform slight internal recompilations of the proper body of the subroutine in an effort to attain efficient execution once this subroutine body begins to function; this approach must be scrutinized carefully from the efficiency point of view, and, to the extent that it depends on modification of code, may be unavailable in a system allowing shared or re-entrant code or recursive subroutines.

The problems described above will require particularly careful treatment in a system allowing linkage between subroutines written in distinct source languages. In general, it will only be possible to transmit a data structure D back and forth between a routine R written in one source language L_R and a routine S written in another language L_S if both languages provide mechanisms for the description of a structure of the type of D, and only if the compilers for both languages are capable of producing code for accessing the component parts of such a structure. This means that if D is, e.g., an array originally described in L_S which is to be transmitted to a routine written in L_R, then L_R must admit a data type having the same storage layout as D. In addition, either the L_R compiler, the L_S compiler, or more probably the linking loader integrating R with S must furnish code for the conversion, at the point of an R to S call or of an S to R return, of the standard data descriptors used within R to those used within S and vice versa.

An idea of the kind of problem that may arise in attempting to link disparate languages may be gained by considering some of the problems that arise in linking FORTRAN to PL/1. In PL/1 one may write a procedure

```
(1)        X:  PROC (A,B)
               DCL   A(*), B(*,*) ;
```

this indicates that the first argument A will be a vector of undetermined bound; similarly, B will be a doubly dimensioned object. Observe that one can obtain a similar effect in FORTRAN by writing

```
(2)        SUBROUTINE  X(A,B,L,M,N)
           DIMENSION   A(L), B(M,N)  .
```

Thus in FORTRAN one is required to include the actual array bounds explicitly in the calling sequence. Accordingly, in FORTRAN, when an array is passed as an argument it is sufficient

to pass the address of the beginning of the array. In PL/1, on the other hand, one must pass the address of a descriptor for the array. If FORTRAN and PL/1 are to call each other, it is therefore necessary that both be provided with mechanisms for the conversion of explicit to implicit information and vice versa.

Additional difficulties will arise from differences in the housekeeping schemes employed by various languages. For example, FORTRAN incorporates virtually no housekeeping code. COBOL incorporates some, generally connected with mechanisms for handling various interrupts (end of file, etc.). PL/1 permits recursive procedures, dynamic storage allocation, independent tasking, etc., all of which require certain global housekeeping mechanisms, maintained in the form of stacks and lists. Since these are global, and are used and updated in every routine, their proper treatment during linkage between PL/1 and other languages may present quite complex problems.

One general plan for the treatment of at least the simplest problems of inter-language linkage is as follows. A call from R to S can be compiled as a transfer to an interface routine T to be produced by the loader linking R to S. We may suppose that T is produced with knowledge not only of the form of the call occurring in R but with knowledge of form in which S expects its parameters. In many languages such information is made available in a special subroutine declarator statement occurring at the very start of S (a subroutine feature in the sense of the present discussion will only exist if such information is definable). If implicit data structures such as list or string areas and implicit pointers associated with these data structures play a role in a particular language, then the information to be transmitted to T may include the location not only of explicitly designated parameters, but also of this additional information. Both the called and the calling routines may produce descriptor information concerning arguments of calls and of called subroutines; part of this information can be digested statically at load time to control

the form of the interface routine T and part handled dynamically by T at run time. On receiving control from R, T will execute any necessary interfacing initializations, and transfer control to S. When S completes its execution, it will return control to T, which will take any necessary finalizing actions and return control to R. Interfacing actions may include both the conversion of data descriptors and the physical transmission of data in either direction between the storage areas controlled by R and S. The precise interfacing actions to be performed by T will be known to the loader program, which must be able to recognize the languages L_R, L_S, etc. in which subroutines to be loaded have been written, and which must be able to link routines written in all possible pairs of languages.

We shall at this point not attempt any systematic discussion of the rather general issues sketched out in the preceding paragraphs, but shall confine ourselves to the discussion of linkage efficiency in situations where in short sequences of data locations or values can be transmitted without any special conversion action being necessary. In a strictly closed linkage style, the calling routine will not know which of its arguments a subroutine will modify; thus all arguments will be passed to the subroutines in terms of their addresses rather than as values. If the source language being compiled assumes static storage allocation, these addresses will tend not to vary dynamically, except in the case of indexed arguments of the form A(I). In such situations, a linkage style in which an actual subroutine transfer instruction is followed by a table of argument addresses ("transfer vector") can be efficient. Note that in this conventional linkage style, a single address will suffice to specify both the location to which return is eventually to be made and the location of the parameter address table. If a subroutine is called with indexed arguments of the form A(I), the value of the corresponding parameter address must be calculated and inserted into the argument address table. Note that this linkage procedure, which involves modification of table locations interspersed

with code, is not compatible with the recursive use of subroutines. A roughly corresponding procedure that could be employed if recursive subroutines or re-entrant code were at issue is as follows:

i. Within whatever storage area or storage stack is appropriate, locate or allocate a block of storage sufficient to store the parameter information for the sturoutine to be called.

ii. Insert all necessary parameter information into this parameter area.

iii. Load the address of the parameter area into a standard register.

iv. Enter the subroutine.

The manner in which arguments will be referenced from within a subroutine will depend to a certain extent on the structure of the machine on which code is to run. If a computer is provided with what in practical terms is a sufficiency of index registers, argument addresses may be held in these registers when needed, permitting efficient fetching and storing of arguments. If the order code set of a particular computer includes double indexing, holding argument locations in index registers will also be compatible with efficient reference to indexed arguments. If double indexing is not provided, or if only a few index registers are provided, other methods of argument reference may be preferred. Reference to simple (i.e., non-indexed, non-array) arguments may be facilitated by supplying each subroutine S with a prologue in which the values of such arguments are fetched using their addresses and stored in locations M local to S, as well as an epilogue executed just before return in which the values of all output arguments are transmitted back to the calling routine. Arguments treated in this way may be referenced during execution in the same fashion as quantities local to S. Transmission of arguments in terms of location references is sometimes designated as call by reference, and this special technique of import followed by export of simple arguments is sometimes designated

as call by value.

Indexed and array arguments must be treated differently. One scheme for the treatment of indexed subroutine arguments furnishes each subroutine with a section of prologue which, before entry into the proper body of a subroutine, stores the address of each indexed argument into all instructions referencing the argument, thus affecting linkage dynamically each time a subroutine is called. This makes a costly linkage but does allow subroutine bodies to execute efficiently. Another possible treatment of the linkage problem for indexed arguments, applicable on machines with sufficiently many registers but without double indexing instructions, would attempt to reduce the strength of address calculations implicit in indexed argument references A(I) inside of any loop within which the index I was iteratively calculable. Within such loops, A(I) could be referenced by a single instruction; in other situations, A(I) would be referenced by adding the location of A (brought either into a register or into a location local to a subroutine) to the index value I and then using a separate load or store instruction to complete the reference.

Optimizers for machines on which such procedures may be necessary ought to make greater efforts to avoid redundant address calculation than have been indicated in the present section. To this end, various techniques related to those already described are available. One may, for example, assign separate value numbers to the value of A(I) and the calculated address of A(I) during linear nested region optimization, allowing repeated address calculation to be avoided, even in such sequences as

```
(1)      X  =  A(I) + ···
       A(J) =  ···
         Y  =  A(I) + ···  .
```

Going beyond this, one might try in a systematic way to detect and exploit such facts as the fact that the addresses of A(I) and B(I) necessarily differ by a quantity constant throughout an entire subroutine.

The processes of global optimization which have been described above apply with strong effect to code containing subroutine calls only if the subroutines behave in "normal" fashion. Thus, for example, in order that code flow be traceable even in the presence of subroutine calls we require that subroutines, when called, make normal returns, or at least that the set of addresses to which they may return be known. In particular, subroutines to which transfer addresses may be passed as arguments and within which such addresses may be unrestrictedly manipulated can constitute absolute boundaries for global optimization methods. A similar remark will apply in cases wherein a variable location function (XLOCF in FORTRAN) and an indirect assignment function can be called, making it feasible to pass the location of one variable as the value of another variable to a subroutine. If a global optimizer has to deal with such things, no better method may be available than to treat each call to an exceptional subroutine as a point at which all global optimization information is invalidated, all registers containing live information must be stored, all locally stored argument values must be transmitted to their true locations, etc. The language designer intending to produce a highly optimizable language must bear these considerations in mind; decisions justifiable from the point of view of abstract generality and flexibility may be catastrophic for optimization.

As we have already noted, semi-closed linkage differs from fully closed linkage in that information concerning a subroutine S is furnished to each routine R calling S during allocation of registers within R. Using this information, substantially improved linkages can be provided. Such techniques are often used, not really for library subroutines in general, but for a more limited class of "system" subroutines of whose special properties a given compiler can be explicitly aware.

In compiling a subroutine S for semi-closed linkage, we treat all subroutine return statements as transfers to a nominal return block, and treat the entry point to S as a

nominal entry block within which all subroutine arguments are defined. Registers are then allocated within S in normal fashion. On completion of register allocation for S, it will appear that S requires that certain of its simple arguments be loaded in particular registers on entry and provides certain simple output arguments in registers on exit; here, a simple argument m is to be considered as an output argument if it is impossible to traverse S from entrance to return without encountering an assignment to m. This information concerning the state of registers on entrance to and on exit from m is then used during the register allocation phase of R-optimization, during which the call to S is treated as a program block within which certain register assignments have been made. Any output argument whose value is used from storage within R after the call to S is stored immediately on return from S; other output argument values may be available in registers at all subsequent uses and need never be stored. If a simple argument of S is used from storage within S, it is flagged to indicate this fact; this information is also passed back to R, and used during the final stages of the register allocation process to force appropriate insertion of a store. If a simple argument m of S can be set from within S, but need not be set within S before return is made, the normal process of register assignment within S will introduce stores with target m into S. For some machine structures it will be appropriate to expand such stores as indexed stores replacing the argument value at its true location within the calling routine R; for other machines, it will be preferable to use the "call by value" technique of first storing m in a cell local to S and transferring this value to the calling routine R only on exit from S.

Even if semi-closed linkage to S is used, indexed arguments contine to present the problems we have already recognized in our discussion of closed linkages. On computers having sufficiently many index registers and allowing double indexing, indexed arguments may be referenced directly from

within a subroutine by the use of double indexing. In other cases, we must use one of the alternate treatments of indexed arguments described above. During the optimization of S, we can determine which arguments and which COMMON arrays S may modify. This information can be passed back to R, where its availability will allow more effective optimization than would otherwise be possible.

If a semi-closed linkage is used, then optimization of a calling routine R involves the use of certain information concerning called subroutines S. In a functioning subroutine library system, it would be possible that, after R was compiled, but before R and S were linked together into a common subroutine, S should have been recompiled, invalidating some of the register usage information already utilized by R. Corrective action leading to successful linkage and execution will only be possible in such cases if the information about S used in the compilation of R is recorded with R, and any changes in S noted and compensated for at linking time. For example, compensation could be made for a change in the register in which S expected a certain argument by allowing the linking loader to precede calls to S by inter-register move operations, while compensation could be made for changes in the collection of arguments referenced from core within S by preceding calls to S by appropriate load-store sequences. Changes in S for which the linking loader was unable to compensate would have to be treated as link-abort conditions requiring recompilation of R. Note however that none of these problems arise when a calling routine is compiled together with its subroutines, as may tend to be the case for ALGOL, PL/1, or other languages which encourage the embedding of subroutines within the text of the routines which call them.

If a semi-closed linkage style is to be used for a language which permits recursion, so that the calling pattern for a family of subroutines need not be a tree, certain relatively arbitrary decisions concerning the order of subroutine compilation must be made. A plausible scheme for handling this

problem is as follows. Find a set K of subroutines such that the subroutine calling pattern P for the whole family of subroutines to be compiled becomes a tree if all calls to subroutines of K are deleted from P. Then compile all subroutines not in K, using a strictly closed linkage at every call to a subroutine of K. Finally, compile the subroutines of the collection K.

Code generated by open subroutine calls enters into global optimization on the same basis as code originally present in a calling program, and hence affords no special difficulty during optimization. It is, however, worth commenting on the manner in which subroutines S capable of being invoked via open calls must be stored in a program library, and on some of the details of the process for incorporating an open subroutine into a calling code. Such subroutines may be maintained in a library in the form of suitably compacted intermediate language code, within which all ordinary symbol table references are replaced by references either to sequentially numbered local variables or to sequentially numbered dummy arguments of S; an associated table of variable attributes must also be provided. If R is the routine calling S, compilation of R will, in the ordinary course of things, transform every call to S (implicitly) into one in which all arguments are either simple variables or simply indexed variables. To compile what then remains of an open call, we expand the compacted intermediate code constituting the open subroutine body, replacing every local variable Vn of the subroutine by a pointer referencing a compiler-generated symbol Gn corresponding to Vn; Gn is inserted into the symbol table E being developed for R and all attribute data associated with Vn is transferred into E from within the condensed attribute table associated with S. When, during the same process, a dummy argument D of S is encountered, we replace the reference to it by a reference to the corresponding actual argument m of the call to S, if m is simple. On the other hand, if m is an indexed argument

and D occurs in an expression, we insert a matching indexed load operation into R and replace the reference to D originally occurring in S by a reference to the result of this indexed load operation. If m is an indexed argument and D occurs as the target of an assignment statement, we replace the simple assignment operation in S containing D by an indexed assignment operation containing the indexed variable m.

On completing the expansion of all open subroutines we will have produced an intermediate language text in standard form and optimization can go forward in normal fashion.

In practical terms, semi-open and semi-closed linkage differ only in the extent to which they insist on sterotypy of information transmitted from a subroutine to a routine calling it. The technique, described above, of treating a semi-open subroutine as consisting of two parts, the first being called an open subroutine and in turn calling the second in semi-closed fashion, reduces the implementation problem for this linkage to cases already discussed. It may be desirable in using semi-open linkages to establish a fixed upper limit for the length of program portions to be compiled in the open style, as otherwise excessively large codes may result in particular cases.

6.16 Notes and Comments

A) Introduction

In spite of the fundamental importance of the subject, code optimization is a subject with a relatively small literature. Optimization may be divided into two principal subbranches: optimization of size and optimization of speed. Size optimization of code has been little studied, partly because the presence of tables alongside code means that the effect of code size reduction is in many situations limited. We remark only that considerable code size reductions can often be obtained by the representation of code in a specially designed interpretable format: this makes it possible to use short rather than full length addresses, implicit references to certain operands and operations, and special dense calling sequence conventions as well. The discussion of APL contained in a subsequent chapter will indicate some of the possibilities that lie in this direction, which is one requiring more work. Table space can be minimized by use of an automatic overlay scheme; see Ershov [2] for an account of one algorithm of this sort. Most work on optimization has concerned itself with speed optimization, and that specifically for algebraic languages. There is, however, a developing if still rather incoherent literature on the special optimization problems associated with access to and searching of large files, and with the associated problem of optimized data location and heirarchically organized memories. We shall not report on this literature in the present section.

Arden [2] discusses code generation for Boolean operations, and describes a method of using conditional transfers for the optimization of Boolean target code. A rather similar discussion of machine-dependent target code optimization for Boolean expressions is found in Huskey and Wattenberg [2]. Floyd [5] describes a number of single statement target code optimizations, some machine independent, others specially adapted to an accumulator machine. These optimizations include detection of equivalent subexpressions, constant propagation, and the choice of a machine-optimal pattern of loads, stores, and input operations. Nakata [1] describes a number of machine-dependent optimizations as they apply to single arithmetic statements. Nakata notes that if one describes an arithmetic expression by its parse tree, then the number of registers required for the storage of intermediate results during the corresponding calculation can be minimized by arranging the necessary computations in an order determined by the branching structure of this tree. This paper also discusses the question of which intermediate quantities occurring during an arithmetic calculation are optimally stored if sufficient registers to contain all such intermediate quantities are available. Irshov [2] is an early paper discussing a similar optimization, and also describing an algorithm for the elimination of redundant calculations in basic blocks.

Stone [1] discusses the rearrangement of computation order in arithmetic expressions using an analytical approach like that of Nakata, but with a different end in mind, namely the development of as many simultaneous operations as possible for a micro-parallel machine. Note that in optimizing a calculation Nakata will arrange it as serially as possible to minimize the number of registers necessary, while Stone would arrange the same calculation in as parallel fashion as possible. Thus, for example, the sum A + B + C + D + E + F + G + H would be compiled

as follows by Nakata and Stone respectively:

(Nakata: single register machine)

```
Load A
Add B
Add C
Add D
Add E
Add F
Add G
Add H
```

(Stone: multi-register microparallel machine)

```
Load A, R1
Load C, R2
Load E, R3
Load G, R4
Add B, R1
Add D, R2
Add F, R3
Add H, R4
Add R2, R1
Add R4, R3
Add R3, R1
```

Allen [1] is one of the first published accounts of an extensive global optimization algorithm. The methods described in this paper may be regarded as preliminary versions of the machine-independent optimization methods to which Section 3 and the later sections of the present chapter are devoted. Allen gives a more detailed account of the use of linear test replacement than is given in the present chapter. Medlock and Lowry [1,2] summarize the implementation of a very similar optimizer in the IBM FORTRAN H compiler; this survey of optimization is particularly readable. Their compiler incorporates many of the optimizations described in the present chapter, including global flow analysis, global common sub-expression finding, code motion, reduction in strength, constant propagation, and test replacement. It also incorporates an interesting algorithm for the rearrangement, using the associative and distributive laws, of sub-parts of a calculation in order to develop more constants and loop-independent expressions than are present in the code as originally compiled. The register allocation problem is discussed briefly, and

an efficient method for conditional code generation by the use of "bit strips" is outlined. The Medlock-Lowry compiler was written in FORTRAN and used to optimize itself.

Ershov [1] describes an optimizing compiler developed for a computer with small memory and a drum backup store; this compiler uses 24 passes for the total process of translation, optimization, and code generation. The optimizations performed include detection of redundant code in basic blocks, code motion, reduction in strength, combination of loops having identical numbers of iterations and containing independent operations, various subroutine calling sequence optimizations, and a careful storage space optimization. The calling sequence optimizations include the following:

1) if a subroutine parameter is always called with the same value, the subroutine is coded so as to use this value directly rather than as a variable parameter;
2) values of subroutine parameters which are not modified by a subroutine are stored locally on entry and used as local parameters thereafter rather than being accessed indirectly through the subroutine calling sequence in all cases.

Storage optimization is handled by tracing the code flow to determine all variables which are "live" at the same time. From this information, a packing algorithm is able to devise an efficient assignment of variables to core locations.

Allard-Wolf-Semlin [1] describe a number of machine-dependent optimizations for code running on computers like the CDC 6600, including optimizations based on the reordering of instructions to attain optimal micro-parallel "scheduling" of the independent multiple arithmetic units available in a computer of this structure; cf. Thorlin [1] for the use of PERT scheduling techniques to this end. The early paper by Nievergelt [1]

lists a number of machine-independent optimizations of the type discussed in the present chapter. Gear [1] describes an optimization scheme simpler than but somewhat resembling the linear nested region optimizer to which the first few sections of the present chapter are devoted. The optimizer described by Gear incorporates procedures for the detection of common sub-expressions, the propagation of constants, motion of code out of loops, and strength reduction in a DO-loop context. The rearrangement of calculation order in an attempt to develop more absolute constants and loop constants than are present in originally compiled code is discussed, as is a generalization of the common subexpression finding process to detect the occurrence of expressions necessarily differing by constants. (For rearrangement to develop extra constants, cf. also Gear [2].) A proposal for using code optimization methods to move input statements in a code so as to provide improved input buffering is also made by Gear. Hill, Langmaack, Schwarz, and Seegmuller discuss strength reduction for indices occurring in loops defined by ALGOL FOR-statements, and cite measurements showing a running time improvement by a factor of 2.5 resulting from this optimization. McKeeman [1] describes various local optimizations, somewhat adapted to an accumulator machine, which suppress unnecessary loads and stores occasionally exploiting the commutativity of operations; cf. also Anderson [1].

Iverson [1] comments on the exploitation of formal identities, especially those involving array operations, for increasing the efficiency of the compiled code. Strachey and Wilkes [1] comment on those features of ALGOL that make it difficult to compile good code for that language.

B. Machine-independent optimization of COBOL

Whereas certain of the algebraic language optimizations outlined in Section 1 of the present chapter remain useful in optimizing COBOL programs, others will normally be of little use. COBOL programs typically make little use of indexed arrays, and thus optimization by polynomial strength reduction will not be of importance. Loops will more typically be terminated by file end conditions than by iteration count limits, and so linear test replacement is not likely to find application.

A problem of particular significance in COBOL optimization is that of eliminating unnecessarily frequent variable conversion and reconversion operations. COBOL integer variables are normally stored in character-string form (one byte per character). Two principal reasons favor the use of this storage convention:

1) A COBOL program will often read records containing many fields and modify but a single field in a record. In such a situation, keeping data in its external form automatically eliminates conversions that would otherwise be necessary.

2) The COBOL language allows the overlaying of variables, the overlay correspondence being specifiable on a character-to-character basis.

Arithmetic, however, is almost necessarily performed on data converted into a different (binary or packed decimal) form. This second form of a variable may be called its converted form. Thus COBOL arithmetic operations may come to carry a heavy load of conversions and reconversions. For example, the COBOL sequence

(1)
```
ADD  1.00 TO A
ADD B to A GIVING C
...
```

would naively be translated as

(2)
```
temp = convert (A)
temp = temp + 1.00
A = reconvert (temp)
temp = convert (A)
temp2 = convert (B)
temp = temp + temp2
C = reconvert (temp)
...
```

and so forth.

The following method, closely related to the global optimization methods described earlier, may be used to improve the quality of such code. With each COBOL variable A we associate its converted form, which we write as A', and which is assigned a separate storage location. Arithmetic operations referencing variables A specified by COBOL source

code are translated by corresponding references to A'. Implied assignments to a variable A are translated by an intermediate language assignment statement

(3) A = reconvert (A').

During optimization, one traces through a program to determine, for each use of a variable A', whether there exists any path back to an assignment to the variable A having a form other than (3). (Such assignments may appear either as READ statements or as assignments to variables overlaying A; we call such assignments abnormal assignments to A.) If this is the case, the reference to A' is prefixed by an interpolated instruction

A' = convert (A).

A bit-vector method may conveniently be used to determine when such conversion operations must be inserted. With each basic block b and each variable we associate a bit x_b equal to 1 if the value of A' on exit from the block may differ from convert (A). With each b, we associate a boolean function f_b defined as follows:

$f_b(x) = x$ if b contains no assignment to A or any variable overlaying A, and no use of A;

$f_b(x) = 1$ if b contains an abnormal assignment to A not followed by a normal assignment to A or a use of A;

$f_b(x) = 0$ if b contains a normal assignment to or use of A not followed by an abnormal assignment to A.

Then, if $b_1,\ldots,b_n$ are the immediate predecessor blocks of b we have the boolean equation

$$x_b = f_b(x_{b_1} + \ldots + x_{b_n}).$$

These boolean equations may be solved by the methods described earlier. This will determine the bits x_b; knowledge of these bits allows reconversion operations to be eliminated as redundant. Moreover, in calculating these bits using our normal method of regions, we will find the information needed to move the conversion of a variable A out of a loop not containing any abnormal assignment to A.

Dead variable elimination in essentially standard fashion will then remove instructions A = reconvert (A') from the compiled form of a COBOL program unless these instructions are followed either by a WRITE statement involving A or a use either of A or of a variable B overlaying A to which an instruction B' = convert (B) has been prefixed. This "conversion optimization" could improve the code sequence compiled for (1) very greatly from its primitive form (2).

The converted-form variables A' can be kept in a set of compiler-assigned temporary locations. Optimal assignment of converted variables to locations will aim at a global placement which avoids recopying and motion of these variables to the extent possible. This task resembles an aspect of the ordinary register assignment problem, namely the association of variables with registers in such a way as to avoid unnecessary register-to-register copy

or move operations. The methods for treatment of this problem which have already been described can be applied with little change to the converted-variable location problem; these methods will of course make use of variable live-dead information. Note that in assigning locations to converted variables we may make use of as many temporary locations as are necessary; thus a problem of key significance in register allocation, i.e. deciding which variables are to be kept in registers and which are to be stored, need not be faced here.

Either a two-address or a three-address register allocation algorithm should be used as a model for converted variable location assignment, depending on the character of the instructions which are ultimately to be used to perform arithmetic on converted quantities.

COBOL variables must normally be maintained to specified precision, floating-point type rounding not being allowed. This fact, and the inconvenience often involved in translation between character-per-byte unconverted variables and binary integers, often implies the use of decimal fixed-point operations for COBOL arithmetic. Such arithmetic will involve frequent and inefficient rescaling of variables; a COBOL optimizer should aim to reduce such operations to a minimum. The technique most appropriate for the avoidance of inefficiencies here seems to be highly machine dependent, and we shall for this reason not attempt to discuss it at this time.

COBOL loops will normally be used for the iterative processing of a sequence of records, and their efficiency will therefore be bound closely to input-output considerations. With this caveat, however, we may say that constant propagation should be as useful for COBOL as for FORTRAN. Constant propagation should always aim to supply constants in the form directly needed by those arithmetic operations in which they are used.

Since COBOL makes little use of indexed arrays and still less of doubly indexed arrays, redundant subexpression elimination is apt to be less important for COBOL than for FORTRAN. The availability in COBOL of the PERFORM statement, which allows out-of-line code to be invoked conveniently, will tend to make this assertion all the more true.

The peculiarities of the COBOL PERFORM statement also have interesting implications for the relationship between translator and optimizer. A PERFORM statement involves an out-of-line block of code, and allows specification of repetition for a given number of iterations or until some other termination condition is satisfied. It is appropriate to compile a PERFORM using a parameterless subroutine call, imbedded if necessary in a repetition control block. COBOL allows the statement range ("invoked" block) governed by one PERFORM statement to overlap with blocks governed by other perform statements. The range of a PERFORM statement

may also be entered directly either by transfer or by straight-line control flow. If a PERFORM block is entered in this way, control leaves it in normal fashion once the end of the block is reached.

The target-code style appropriate for achieving these effects is as follows. With every PERFORM block B, associate a compiler-generated label BACKB; as the last instruction of B, insert the assigned transfer instruction

```
        GO TO BACKB .
```

The statement

```
        PERFORM B
```

is then translated as

```
        SET BACKB = A
        GO TO B
A:      SET BACKB = NEXTB
        ...
```

where A is a compiler generated label having the indicated location, and NEXTB is a compiler generated label immediately following the last instruction of the block B.

The following global optimization techniques may advantageously be applied in this situation:

a) PERFORM sections called only once and not otherwise entered should be placed in-line; subsections of complete sections called only once should be placed in-line too. More generally, the best available criteria for choosing

a subroutine linkage form (closed, semi-closed, open, etc.) should be applied to each PERFORM statement.

b) Once the PERFORM blocks to be invoked in other than open fashion have been determined, we may exploit the fact that the whole text of every PERFORM block is available to us during compilation. This may be done as follows:

i) Survey the source text to determine the pattern of calls between PERFORM sections. As the COBOL language does not permit recursive use of PERFORM invocations, it will then be possible to rank the PERFORM section in such a way that each section invokes only blocks of lower rank.

ii) Start with the code contained in a section of minimal rank. Since each PERFORM section in COBOL is eventually required to return to the point from which it was invoked, such a section may be regarded from the point of view of a program invoking it as a single-entry, single exit region. By analyzing the flow of this region and solving any relevant systems of boolean equations for it, we may assign a bit-vector to the section, this bit-vector describing initial variable uses, variable assignments, etc. which occur in the section. This information will correspond closely to that associated for optimization purposes with a basic code block.

iii) Once a bit vector of the above kind has been associated with each PERFORM section of given rank, we may repeat the same process and calculate similar bit-vectors for every PERFORM section of next higher rank. Ultimately,

bit vectors will be assigned to every PERFORM section in a program.

Within a total COBOL program, each PERFORM section invocation can be represented by much the same sort of intermediate text item as would be used for a subroutine call in a FORTRAN-like algebraic language, except that, instead of associating argument-use information with the item, we associate with it a full bit-vector describing the effect of the invocation on the status of every program variable. Using this information we may apply normal procedures of global optimization. The following constraint must be applied in addition to the usual constraints on code motion: code belonging to any particular collection of PERFORM sections, if it is to be moved, can only be moved to a block belonging to this same collection of PERFORM sections.

Once the PERFORM-section optimizations described above have been applied to COBOL code, loop-bookkeeping overhead is not apt to be a significant part of the total process constituting a loop, and optimization by loop unrolling is therefore not apt to be of much interest.

Since COBOL makes use of higher-level instructions than does FORTRAN, and since it works with a wider variety of data types, optimization by code selection down to a machine-dependent level can be very important. To the extent that these operations are performed by library

subroutines, general procedures for subroutine linkage optimization, able to choose linkages intelligently for efficiency, will be of benefit. Beyond this, we may note the importance of applying special-case selection mechanisms in compiling higher-level operations of particular importance, especially the MOVE statement, which plays a central role in determining the efficiency of many COBOL programs.

C. Index register allocation for Algebraic languages

Whereas rigorously based results applying to simplified models of register allocation in linearly ordered code are available (cf. Horwitz, Karp, Miller, and Winograd [1]), no such results have been derived for code containing even simple loops, much less for code with general flow. Thus register allocation in the presence of flow must use a heuristic lacking complete theoretical foundation. The particular method used will depend strongly on the extent to which global information is to be collected in support of register allocation. Machine structure will, of course, play a fundamental roll as well. The simplest scheme, incorporated in many industrial compilers, is to put a newly generated quantity in an empty register if possible; if none is available, the contents of the register used farthest away in the following code is displaced. In this method, register contents are remembered over the extent of a basic block, but all registers are stored and reloaded at each label. This method requires only local look-ahead.

A more elaborate register allocation heuristic has been outlined in a preceding section. Here we shall describe the allocation methods actually used in two interesting production compilers, the original FORTRAN I compiler and a FORTRAN compiler for the RCA SPECTRA 70.

The method which was used for index register allocation in the FORTRAN I compiler is described in Backus et al [1];

in the present paragraphs, relying on the generous aid of Mr. Sheldon Best, we proceed to set forth a somewhat more detailed account of certain concepts and details concerning the register allocation methods used which were not given in the cited article. The overall flow of the FORTRAN I register allocation was as follows.

a) Simulation for Frequency Determination

As a prelude to register allocation, the absolute execution frequency was estimated for all basic blocks of the program, during which process certain control statements, namely the DO, GO TØ, and ASSIGN statements, were executed interpretively while the action of others, namely the IF and the computed GO TO statements, was merely simulated. To aid this simulation FREQUENCY statements were used to define both the branching probabilities applicable at IF statements and computed GO TO statements, and also the average iteration counts for DO-statements with variable limits. When frequency statements were not attached to IF-statements, nominal transition frequencies were assumed. Similarly, at DO statements not provided with frequency information, a certain small number of repetitions was assumed. The possibility of solving the simultaneous equations determining path frequency in terms of transition frequency using known methods for solving sparse matrix equations was considered, but no methods which would work in the presence of DO-loops and

assigned GO TO statements was hit upon, although IF-type branches alone could have been handled without explicit interpretation.

The frequency-estimating simulation traced the flow of control in the program through a fixed number of stops, and was repeated several times in an effort to secure reliable frequency statistics. Altogether an odd method!

b) Region processing order choice

In order to simplify the index register allocation, it was implicitly assumed that calculations were not to be reordered. The contrary assumption would have introduced a new order of difficulty into the allocation process, and required the abstraction of additional information from the program to be processed. The order of region analysis during register allocation was determined by the frequency order of paths in the program and this made it virtually certain that the most frequent loop in the program would be the region processed first (assuming that frequencies were well estimated).

c) Register-use look-ahead in basic blocks and larger regions

As long as one is concerned only with minimizing the number of loads of symbolic indexes into actual registers and not with minimizing the stores of modified indexes it is obvious that the optimum strategy for straight-line code consists in selecting that index register for replacement which contains the symbolic index which is re-used most

remotely. Of course, if an index value becomes "dead" then it should be selected for replacement before any "live" value. In dealing with loops constructed from basic blocks and larger regions the situation is different from the straight-line case in two related respects: 1) every index is re-used during each turn through the loop (unless it becomes dead) and 2) the contents of the registers at the beginning and end of the loop must ideally be made to match.

The FORTRAN I method for dealing with those problems for loops and more general sets of basic blocks was as follows. Branches and blocks were selected in decreasing order of estimated frequency to form connected regions or paths for processing. Each such region consisted of groups of blocks or previously processed connected regions reachable one from another by traversing highest frequency branches from already selected blocks. In processing a new path connecting two previously disconnected regions, register usage was matched by permuting all the register designations of one region to match those of the other, as necessary. In processing a new path linking a block to itself and thus defining a loop, the loop was first considered to be concatenated with a second copy of itself, and straight-line register allocation carried out in normal fashion through the first of the two copies, with look-ahead extending into the second copy. The contents of registers

on exit from the first copy thus determined was now applied both as the initial condition and as a "terminal condition" applicable to the second copy. This terminal condition was in turn used during register allocation within the second loop copy in the following way. Straight-line allocation was carried out for the second loop copy in essentially normal fashion. However, the look-ahead procedure employed during this allocation was modified as follows: a register seen on look-ahead to the end of the loop to contain the correct terminal quantity was considered to have a "subsequent terminal use", while a register not in this condition was not considered to have such a use. This information in turn controlled the pattern of register loads and unloads in the loop, in the manner already described; the allocation produced for the second loop copy was that ultimately used in generating machine code.

d) Activity of indices and the insertion of local stores

When an operation α involving an index changed the index, the index was said to become active. It then had to be stored in its cell before displacement from a register. This store was inserted right after α if the displacement of the index occurred within a loop. However, in the case of indices which could be retained in a register throughout a loop, store instructions were inserted on all exits from the loop, thus moving the necessary store operation outside the loop.

e) Adjustment of entry and exit conditions

Stores of active registers upon exit from a loop, and initial loads of registers upon entry into a previously processed region were regarded as occuring between basic blocks. Any necessary initialising load instructions were arranged in little clusters provided with various entry points as in the algorithm sketched in the preceeding chapter, allowing different subsets of registers to be loaded by branching to different parts of the cluster. Each cluster was so arranged as to minimize the number of instructions executed during "most frequent" branches to the cluster. Terminal storage of active registers was accomplished by attaching store-index instructions to region exits.

This completes our brief account of the FORTRAN I register allocation algorithm. Machines like the IBM 360 and RCA Spectra 70 have features which introduce additional complexity invalidating the simple assumptions upon which the FORTRAN I analysis of the index register assignment problem was based. Moreover, although these groups of machines all have essentially the same instruction codes, it is still the case that the timing of instruction execution varies so much among member machines as to make it unlikely that an assignment algorithm which would be near optimum for the whole class could be developed. The following list of conditions and constraints gives

an idea of the facts which must be faced in treating the register allocation problem for these machines:

1. Sixteen fixed point registers serve as fixed point accumulators, and, except for register 0, also as index registers and base registers.

2. Nearly every reference to memory requires the use of a base register or index register since the displacement field available in an instruction is limited to 12 bits.

3. Adjacent registers are used in coupled fashion in certain double precision arithmetic operations.

4. The time required to load several adjacent registers from adjacent memory locations may be significantly less than the loading time applicable to non-adjacent memory locations, and the time to move one register to another may be much less than a store-load combination. Two registers may be interchanged in minimum time on some machines without the use of a third register; on others, using a third register gives minimum time.

5. Two three, or even four registers may be used or affected by a single fixed point instruction (even excluding multiple load and store instructions, which can involve all the registers).

6. The four floating point registers of these machines can contain either single or double precision floating point quantities.

7. The floating point registers can only be loaded and stored individually; no movement of information between the fixed and floating point registers is possible. Information can be moved from one floating point register to another without using memory but two floating point registers cannot be exchanged without using a third register or using memory.

8. Standard subroutine linkages use two to four fixed point registers in a rather involved way. Certain features of these linkages are dictated by the fact that only a short address field (called the displacement) is available in instructions. This fact has the following consequences:

i. Storage references generally require the use of a register, and references to distinct arrays may require the use of distinct registers. On subroutine call, all such base quantities may have to be stored; on return, they may have to be reloaded. Even references to the section of program currently executing may require a register.

ii. Inter-block transfers generally require the use of a register.

It can be seen from the above that register allocation for this class of machines involves two separate, rather different allocation problems; fixed point and floating point allocation. However, these two problems interact, since floating point instructions may have to use bases and indices to locate their operands.

These facts, plus the great variety of instructions available on the machines under consideration, imply strong interactions between the problems of register allocation, code generation, and storage assignment. The resulting optimization problem is sufficiently complex as to defy rigorous treatment. We shall, in order to give the reader an idea of the manner in which such questions have been treated in practice, outline in considerable detail the treatment of register allocation actually used in a FORTRAN compiler for the RCA Spectra 70. In this compiler, several simplifying constraints were imposed on the allocation process:

i. Floating-point register allocation was to be accomplished independently of and logically before fixed-point register allocation.

ii. Only intermediate results arising during expression evaluation were to be kept in floating point registers; i.e., no analysis was to be made to see whether a variable or constant was used frequently enough in a basic block to warrant keeping it in a register throughout the block. Note that this limitation applied only to floating point quantities occuring during expression evaluation, not to base values and indices used in auxiliary fashion; such index quantities were retained globally under conditions explained below.

iii. No look-ahead other than that involved in a

basic block common subexpression finding process incorporated in the compiler was to be done. This meant that local decisions on which quantities were to be displaced when a register was needed could not use look-ahead information except in the special cases of intermediate quantities used more than once in a basic block and variables defined and used subsequently within a basic block.

iv. No reordering of computations to improve the register allocation was to be done.

v. No flow-analysis was to be done beyond the recognition of innermost and "simple" DO-loops, i.e., DO-loops with no branches out of their range which could conceivably be followed by a branch back into the range. Within such simple DO-loops, quantities (especially index quantities) dependent on the control variable were to be maintained entirely in registers, if desireable.

In an effort to avoid the proliferation of base quantities, a decision was made to group all simple variables and program constants, as well as all register save locations, into one contiguous area, addressed if not too long by one standard register, and otherwise addressed by up to five registers. One additional register was reserved for references to the section of program currently executing and for subroutine linkage. Four additional registers

were generally used for function result return, parameter list location, return address, and entry point address, though where possible those registers did double duty, storing fixed point temporary quantities and indices. This left from 4-8 registers available for the storage of bases, i.e., data locations not varying through a given program or subroutine. Since, as stated, register allocation was to be done without the use of flow analysis, the following crude method for allocation of registers to base quantities, which embodies a concept of semi-permanent register assignment, was employed:

Base values that were used sufficiently often would be assigned semi-permanently to registers, loaded at the start of the program, and assumed to be in registers at the start of every basic block. If, however, it was necessary to displace these global bases within certain particular basic blocks, then they would be reloaded via a single load-multiple instruction at the end of each of such block.

The bases which were to be assigned semi-permanently were chosen as follows. During the preliminary steps of compilation, the base quantities to be used within a program were determined, and the number of references to each base counted. Then the k most frequently referenced bases were assigned semi-permanently to registers, and the remaining registers used

to load bases more dynamically as needed within a block. The integer k was determined using the following heuristic considerations: From the frequency of base reference, and the total number of blocks in a program, the probability of referencing any given base in a given block may be estimated. The "marginal advantage" of assigning one additional base semi-permanently is proportional to the number of blocks in which this base is referenced; if g is the total number of registers available for bases, the "marginal cost" of doing so is proportional to the number n of blocks in which more than g-k bases not semi-permanently assigned are referenced. From the probability that any single base be referenced in a block, and on a simplifying assumption of statistical independence, n may be estimated, and k chosen so as to balance marginal advantage against marginal cost.

It would of course have been of advantage in all of this to assign blocks a weighted significance according to their frequency of execution, but this would have required recourse to a certain amount of flow-tracing.

A similar algorithm for register assignment is used in TSS FORTRAN for the IBM 360. Within a given loop, a count is made of the frequency of occurrence of integer expressions and offset quantities. A number of index registers are assigned to the most frequently used quantities of this kind. A number of additional registers then remain available

for local assignment. Local assignment is done by a straightforward sequential process; during this process a record is kept of the variables and non-globally assigned expressions which are available in registers. When variables needed for particular operations are available in registers, load operations are elided. If a given register contains the value of more than a single variable, the list of all variables which it contains is available.

Index register assignment methods of the very simple type just described operate rapidly and can produce satisfactory FORTRAN object programs, but they certainly cannot

Really good register assignment cannot be done without tracing flow and without attaching estimated frequencies to blocks and branches. In view of the importance of frequency information, it has been proposed that frequencies should be measured rather than estimated by running a program before optimization, using typical input data and counting the number of times each branch path is taken. Then time-optimization should concentrate on very frequently used loops and memory space should be optimized in low frequency regions. The highest frequency loops should be compiled first so that register and memory allocation can be adjusted to match them. It is also doubtful if register allocation should be treated separately from code generation and storage assignment if really top-notch code is to be

produced.

Sophisticated register allocation, as well as other forms of complex optimization, can make debugging more difficult because variable values may be found in unexpected places; presumably such optimization would only be applied to nearly debugged programs. As a debugging aid a suitably generalized storage map could be used to show where a variable is at different times; this information could either be used by a person interpreting a dump or by a debugging routine.

CHAPTER 7. SPECIAL PURPOSE LANGUAGES: LSIP AND SNOBOL

1. LISP and related recursive languages.

Since its introduction by John McCarthy in 1960, LISP has become one of the best known and most successful of the languages intended specifically for non-numerical computation. In the present section, we shall discuss the principal features of this language. We note, to begin with, that, as is true of most programming languages, LISP exists in several dialects differing slightly in those language features which are not of common use. The volume entitled The Programming Language LISP: Its Operation and Applications, Information International, M.I.T. Press, Cambridge, Mass., 1964, gives a reasonably authoritative account of the dialect LISP 1.5. M. Harrison's Introduction to Non-Numeric Programming, New York University, 1967-68, presents a slight variant of the LISP 1.5 dialect, corresponding to a LISP system developed for the CDC 6600. We shall rarely be concerned with those language details which are apt to vary between different LISP versions; where this is the case, however, we shall tend to follow Harrison.

LISP may be regarded as a powerful and extensible framework for the specification and coordination of complex logical processes. The framework provided by LISP is constructed of a remarkably small number of mechanisms, which are however, among the most powerful general mechanisms available to the programmer. LISP notably makes systematic use of recursion. It works with basic data items which are pointers to more complex data structures, thereby making effective, organized use of the powerful programming technique of indirection, on the basis of which very effective techniques of substitution may be employed.

Since a LISP pointer may in fact point to a machine language subroutine, and more generally to a block of code executable by any specified interpreter routine, LISP provides a framework within which routines written in any arbitrary language can be invoked. A LISP pointer may also point to yet another LISP pointer, or to a pair or structured collection of such pointers. This implies that the LISP

system allows indirection to any arbitrary level. Moreover, since a list structure is nothing but a collection of pointers referencing each other, a list processing scheme in harmony with the more basic idea of indirect reference results.

Any list consists of a first item and a collection of items which follow. LISP gains additional power by making systematic use of the simple observation that such a list may be interpreted as a function invocation if the first item on the list is considered to be a function and the following items are considered to be its arguments. Employing this idea in a systematic way, one is able, in LISP, to assign a value to every list. Moreover, since the several arguments, both input and output, of a function can be conglomerated into a single list of arguments, LISP can, as a standard matter, work with functions having single outputs, which outputs may on occasion be lists. This technique also allows functions having indefinite numbers of arguments to be used freely.

At the level of implementation LISP makes use of a simple, powerful, completely dynamic storage allocation scheme. Moreover, since LISP keeps systematic track of all the information available to it at a given moment, unreferenced storage areas can be collected for re-use without explicit programmer indication being required. This simplifies the task of the programmer and avoids what otherwise would be a common source of programming error.

In order to specify the internal workings of LISP in all necessary detail, we shall define it in terms of a still simpler, quite rudimentary language, embodying the principle of recursion and establishing various specific recursive subroutine linkage conventions, but not incorporating those additional specific list processing features with which the full LISP system will be concerned. This rudimentary language will be called RUDE. Recursive programming techniques are based essentially on the use of pushdown stacks; the RUDE language will therefore consist most essentially of a set of conventions for the use of such a stack.

The RUDE language provides only eight simple statement forms, of which six are executable and two are declarations. The six executable statements are an IF statement, a GO TO statement, a version of the FUNCTION CALL statement, a RETURN statement, and two statements, PUSH and PULL, required for the explicit manipulation of the pushdown stack with which RUDE is concerned.

We may justify specific conventions employed by RUDE in the following way. When a subroutine is called in a fully recursive system involving a simple pushdown stack, all the information supplied to the called subroutine from the calling program will be placed on the pushdown stack. This information must include whatever address will eventually be required to return from the subroutine to the calling program and to restore the pushdown stack to the condition expected by the calling program. In addition, space must be reserved on the pushdown stack for whatever additional internal, intermediate information the called subroutine will generate.

Three quantities must be available in places known to the subroutine: the bottom of the stack area allotted to it, which may also be the address of the first of its arguments; the total number of its arguments; and the top of the stack area allotted to it. These three information items may be embodied in a set of three pointers. The particular convention actually employed by RUDE is shown in Figure 1.

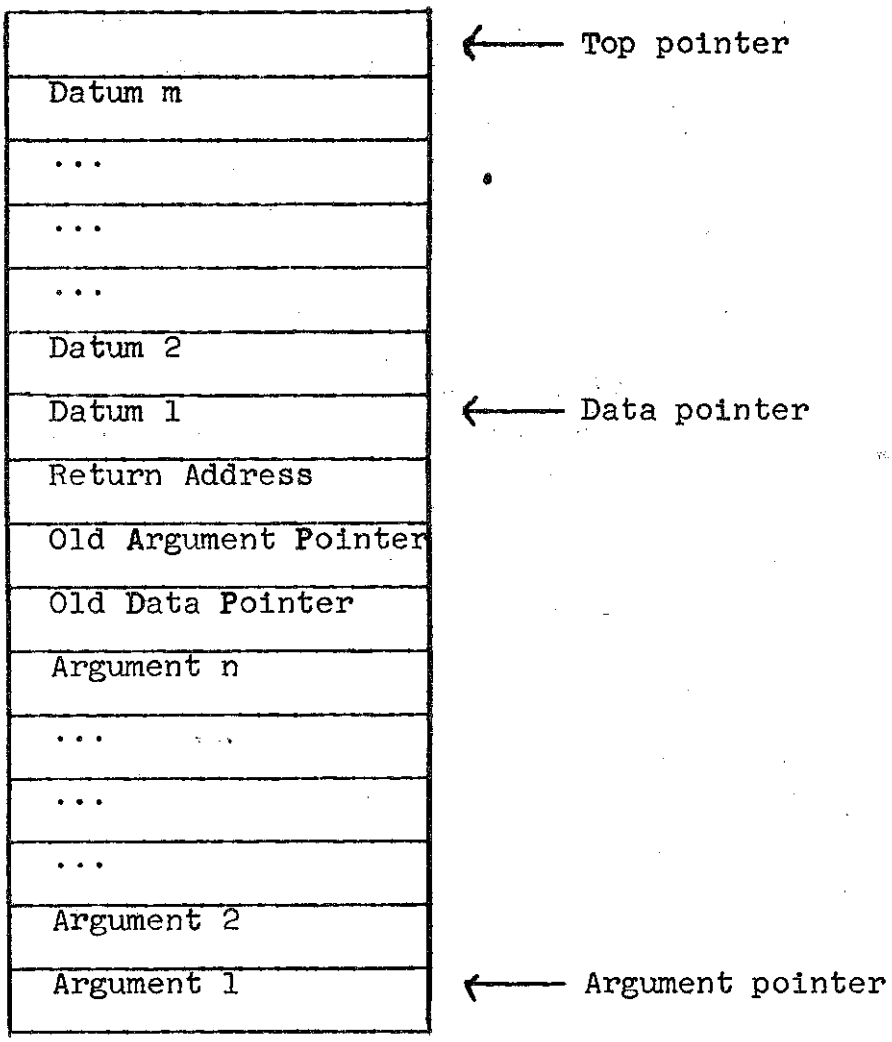


Figure 1: Data fields accessed by a subroutine in RUDE recursive linkage convention.

The arguments supplied to a subroutine by the calling program are grouped in a sequential block at the bottom of the stack area available to the subroutine. Immediately above the last of the subroutine arguments, three locations on the stack are reserved for storage of two pointers and of the return address which the subroutine is to use on concluding its operation. Immediately above the return address as many locations as are required by the subroutine for the storage of "internal temporaries" is kept. Three pointers are used by a RUDE subroutine: an argument pointer referencing the stack location of the first argument; a data pointer, referencing the stack position of the first internal datum in use by the subroutine; and a top pointer, referencing the first unused stack location. A subroutine may determine the number of arguments with which it has been called as the difference between the data pointer and the argument pointer, less three. The value to which the top pointer must be restored on returning from a subroutine is simply the value of the argument pointer used by the subroutine. The values to which the argument pointer and the data pointer must be restored on return from the subroutine are stored on the pushdown stack in the data locations indicated in Figure 1 above, and may evidently be referenced easily by use of the data pointer.

RUDE makes use of the following "source language" conventions. A subroutine argument will be referenced by a subroutine as a1, a2, ... A data item (i.e., temporary variable) local to the subroutine will be referenced as x1,x2, ... We assume all subroutines to return a single value (which in the intended application will often be a pointer to some larger data structure). We assume that, as a standard matter, this value is returned in the position of the first subroutine argument, i.e. in the position which after return from the subroutine is referenced by the "top pointer." Introducing a useful, slightly special convention, we will allow a RUDE program to refer to this location, i.e. to the stack location whose address is given by the "top pointer," as x0.

A RUDE program is made up of a succession of statements. A statement may or may not be labelled. We allow several statements to be placed on a single line, except that only the first of these statements may be labelled. Successive statements placed on a single line are separated by semicolons.

The form of the RUDE GO TO statement is

(1) go to ⟨label⟩ ,

where "label" may be any statement label. The form of the RUDE IF statement is

(2) if (⟨quantity⟩) ⟨label⟩ ,

where again, "label" may be any statement label, and where "quantity" may be any one of a1, a2, ..., x0, x1, x2,... .

(3) return (⟨quantity⟩) ,

where the quantity appearing in the return statement determines the particular value which will be placed in the location of the first subroutine argument for return to the calling routine.

The RUDE function call statement has a slightly more complicated structure; it combines a rudimentary function call with a simple "assignment" operation. It has the form

(4) ⟨quantity⟩ = ⟨function name⟩ (⟨quantity⟩, ⟨quantity⟩, ...,⟨quantity⟩)

Examples of RUDE statements of this structure are

(5) a1 = f(a1,x0)
x0 = gab(x1,x2,x3,x3) .

A statement of this form is assumed to act as follows. The quantities listed as arguments are placed on the pushdown stack in successive locations beginning with the location referenced by the "top pointer."

Immediately above this are stacked the 3 additional items shown in Figure 1, i.e., the current value of the argument pointer, the current value of the data pointer, and the return address. The subroutine corresponding to

whatever function is named is then called. On return, the subroutine must restore the values of the argument pointer, the data pointer, and the top pointer, and must also place the function value which it has calculated on the stack. If the quantity appearing on the left of the RUDE function call statement which has just been executed is something other than x0, the returned value will then be transferred to the location specified by this quantity. On the other hand, if the quantity appearing on the left of the RUDE function call statement which has just been executed is x0, no such transfer is required.

The RUDE GO TO and IF statements operate in an evident fashion. The IF statement uses the standard convention according to which any non-zero argument is treated by it as a logical "true," and the zero argument is treated as a logical "false." In the LISP system to be described below, the all-zero item will also be the basic "NIL" item of LISP. We may therefore say that the RUDE IF statement is a conditional transfer on a non-NIL argument. The operation of the RETURN statement and of the function-invocation-assignment statement has just been explained. The two final RUDE statements are PUSH and PULL. These two statements both consist of just one word: 'PUSH' in the one case, 'PULL' in the other. The PUSH statement merely increases the "top pointer" by one; the PULL statement merely decreases the top pointer by one. Using these statements, stacks of arguments may be built up prior to the execution of a subroutine call which requires a variable number of arguments; the reader will find a number of examples of this technique in the LISP algorithms which follow later in the present chapter.

The RUDE language also contains two declaration statements. The first of these declaration statements is the FUNCTION statement. The second is the START statement. The FUNCTION statement has the form

(6) FUNCTION ⟨function name⟩

It assigns a name to the function whose definition follows

the FUNCTION statement and marks the entry point to the function. The second RUDE declaration statement is the START statement. This statement may only be used once in a RUDE program and must be the first statement of the RUDE program. We define it to have the form

(7) START ⟨function name⟩ (⟨constant⟩,⟨constant⟩,...,⟨constant⟩)

An example of such a statement would be

(8) START PROCESS (1,100,3.5)

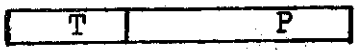
This statement begins the execution of a RUDE program by supplying the indicated constant parameters to a "main" subroutine and invoking this subroutine. The subroutine invoked may then call "exit" by executing a RETURN statement.

Function names occurring in RUDE programs are understood to belong to a list of available functions. In particular, the translator which converts a RUDE algorithm into a block of executable machine code must make use of a list of elementary function names and of the machine addresses at which the blocks of code corresponding to these names begin. A more ample language would of course contain various mechanisms for making additions to the list of available elementary functions. In the present exposition we shall merely assume however that any function occurring in a RUDE program and not defined within the set of RUDE algorithms given is available as an elementary function. We will always explain the (generally rather simple) action of each elementary function used.

The reader will note that the syntactic form of each of the statements of the RUDE language is so simple that the translation of this language into executable machine code is more closely akin to the normal process of assembly than to anything which may truly be called compilation. This assembly process would have the following form. Each statement is taken up in turn and its type immediately determined from the first two symbols which it contains. Code for each of the statements may be generated immediately in a straightforward manner, except that labels referenced

in statements which precede their occurrence cannot be supplied immediately. The normal assembly procedure of leaving all machine addresses corresponding to labels blank in a first pass, accumulating the machine address of all labelled statements in a label table, and returning to fill in all machine addresses in a second pass may be used to deal with this slight difficulty. In this sense RUDE is as much an assembler macro-language put into the form conventionally used for an algebraic source language as an algebraic source language requiring compilation.

We intend RUDE merely as a step toward the definition of the algorithms internal to LISP, and have now reached the point at which we may begin these definitions. In the RUDE language, as we have described it above, we may invoke any one of an arbitrary collection of elementary functions. We now make the additional assumption that the elementary data items which the processes invoked by RUDE manipulate are pointer items having the form shown below.

(9) 

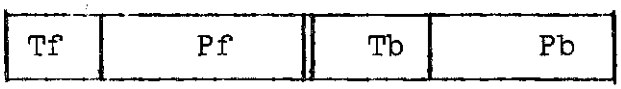
In (9) the field T is assumed to consist of a relatively small number of bits, flagging the type of the data item to which the second or P field points. The P field is assumed to be a field of approximately the same length as a machine address, and to function, like a machine address, as a pointer to some other data structure. The data structure which is referenced by the P field may be any data structure with which a processing system to be written in the RUDE language is concerned. In particular, it may be a word-length real number, an integer, a double precision quantity, an array, a block of executable machine code, a block of interpretable stylized code, or any other data structure that may be required. It is also convenient to allow an immediate data type, which we shall assume as a standard matter to be flagged by the condition T = 0, in which the P field is interpreted not as a pointer but as a bit field to be manipulated directly. Whereas the provision of an immediate data type in this form is not logically

necessary, it is convenient, and serves to enhance the efficiency of the RUDE algorithms which result.

The algorithmic shceme which we have thus far outlined becomes a list processing language if among the data types provided, we include a data type which is a pointer to a "list node" consisting of two or more pointers. Nodes consisting of at least two pointers must be provided in a prospective list processing system, or else the system reduces to a simple linear scheme of indirect addressing. On the other hand, it is perfectly possible and even quite common in setting up a list processing scheme to base the whole list structure on simple binary nodes. More complex nodes may in fact be built up out of sets of binary nodes, so that nodes having a more complex structure are not logically necessary. On the other hand a hypothetical list processing language aiming at high efficiency for specialized applications might profitably make use of a specially designed nodal structure not only containing more than two nodes but also containing nodal fields intended for the storage of useful immediate data.

LISP uses the simplest possible binary node, having the following form:

(10)



Such a node consists quite literally of a pair of data items of the kind shown in (9). We distinguish the two elements of this pair as the "front" and the "back". Each has its own type field and pointer field, as shown in (10). An item of type (9) which points to a node of type (10) will be called a list item. We may then define two basic and entirely elementary functions of a list item; the front function, whose value is the front item of the node to which a list item points, and the back function, whose value is the back item of the node to which an item points. In this sense we may write such expressions as

(11) front (x0), back(a1), etc.

Both the front and the back items of a node may themselves be list items. In this case, successive application of the functions 'front' and 'back' may be used to refer from a given initial item successively and indirectly to some other item.

We provide the LISP system with a basic elementary mechanism for the construction of new nodes by introducing a function 'join' of two arguments. The two arguments of the 'join' function are both required to be items. Given two items to be joined, the join function operates as follows. Using a dynamic storage allocation mechanism which the LISP processing system must contain and which will be described in more detail below, the join function obtains a slot of otherwise unused storage precisely sufficient for the storage of a single node. This node is then constructed by direct insertion of the two items specified as arguments to the join function. Finally, the join function constructs a new item, of list type, and causes its pointer field to point to the newly constructed node.

This new list item is returned by the join function a its value. Thus, join is a function of two items whose value is a single item, namely, a pointer to a node into which the two original items have been inserted.

For reasons which will become clear below it is convenient to provide not one, but two distinct kinds of list items. These will differ only in that the second type of list item is flagged in a recognizably different way from the first in its T field, is called an atom, and is used in a slightly different way at certain points in the basic LISP algorithms. Aside from the differences which will be noted in the algorithms below, items of list type and of atomic type (sometimes called atomic list type for clarity), may be manipulated in the same way; both types of items are pointers to list nodes.

A full LISP system might include items of at least 8 types. Besides the immediate data items, the list items,

and the atomic list items mentioned above, items of machine code type, of integer type, of real type, and of BCD type are useful. In the LISP system which we shall describe, an additional distinction between two kinds of machine code item is made. The first of these machine code types will be called a prefunction code item; the second will be called a postfunction code item. Both kinds of items are pointers to blocks of machine code which may be entered and executed in identical fashion. However, we will, in setting up the detailed LISP algorithms which follow, make use of a convention according which to a machine code postfunction receives its arguments after certain preliminary processing which is omitted in the case of a machine code prefunction. This prefunction-postfunction distinction is more a convenience aimed at simplifing various subsequent expression modes than a basic logical necessity.

In order that the reader may have a definite set of conventions in mind, we specify the existence within our LISP system of the following set of item types and associated T-field flags:

T = 0	Immediate data item
1	List item
2	Atomic list item
3	Machine code function, type prefunction
4	Machine code function, type postfunction
5	Integer of machine word length
6	Real number of machine word length
7	BCD constant of machine word length.

Of course, in a more elaborate LISP system additional item types could readily be provided.

The basic LISP algorithms will of course have frequent occasion to refer to the type of an item which they are processing. In order that this be possible we assume that a number of elementary attribute functions are available. These attribute functions merely examine the type field of an item and return a value which is either zero or nonzero, depending on the contents of the T field, and on the particular

attribute which is required. The specific attribute functions with which we will be concerned will all occur in the basic LISP algorithms which are to be described later in the present section; the significance of each attribute will be explained in connection with the occurrence of the corresponding attribute function in an algorithm.

Two ideas going beyond the elementary notions presented so far play a fundamental role in LISP. The first is that of assigning a value to any list structure; this value will generally be a second, different, list structure. We assign such a value as follows. Consider a list structure, call it L. It consists of two parts; front(L) and back(L). If front(L) designates an item of machine code type, we obtain the value of L by applying the machine function defined by front(L) to back(L). More precisely, we take back(L) to be a list of the arguments to be supplied to front(L). The actual collection of arguments which will be supplied to front(L) is derived from the set of items on this list. If front(L) is designated as a prefunction, the items forming the list back(L) are submitted directly L. If, on the other hand, front(L) is designated as a machine code postfunction, each of its arguments is evaluated before being submitted for processing by the machine subroutine front(L). Next, we consider the procedure to be employed in case front(L) does not designate an item of machine function type. In this case, we require that front(L) be an element either of list type or of atomic list type. Then either front(L) is an atom or not. If front(L) is not an atom, we examine front(front(L)), front(front(front(L))) etc., iterating as often as necessary in order to reach either an item of machine code type or an atomic list item. Proceeding in this manner, we reduce the general case of evaluation to the particular case in which front(L) is an atom. In this case, i.e. when front(L) is an atome, we make a further reduction based upon a substitution procedure. This substitution procedure involves a second basic notion of LISP, the notion of atom value.

As has been mentioned above, an atomic list element is identical in structure and function with a normal list element, except that it is specially flagged. If 'a' is an atomic list item, then by convention front(a) is the 'value of a'. Whenever a value for a is required, this value will be substituted for a. Conversely, to establish a new value for an atomic list item a, we have only to change the front item field of the node referenced by a. The LISP evaluation algorithm makes use of this convention. If a situation is reached in the operation of the list evaluation algorithm in which a value is to be calculated for a list L, and front(L) is an item of list type rather than of machine code type, then front(L) is replaced by its value (in general a complex list structure) and the evaluation process proceeds iteratively to find an initial atom in this replacing structure. This process of replacement and initial atom finding will continue until a situation is reached in which the first item of a list to be evaluated is found to be a valid machine subroutine item, in which case the machine subroutine concerned will be applied directly to the remainder of the list being evaluated in the manner indicated above.

A precise specification of the basic evaluation procedure which has just been described informally is found in Table I.

TABLE I. The basic LISP evaluation algorithm.

	function value	
start:	if(atom(al))isat	x0=atom(al); if(x0)isat
	if(atom(front(al)))do	x1=front(al); x0=atom(x1)
		if(x0)do
	x1=join(back(front(al),	x0=back(x1); x1=back(al)
	back(al))	x1=join(x0,x1)
	al=join(front(front(al)),x1)	x0=front(al); x0=front(x0)
		al=join(x0,x1)
	go to start	go to start
isat:	return(front(al))	al=front(al); return (al)
do:	if(machsub(front(front(al))	x1=front(x1); x0=machsub(x1)
	domach	if(x0)domach

if(atom(front(front(al))) expatom	x0=atom(x1); if(x0)expatom
x0=errorroutine	x0=errorroutine
expatom: al=join(front(front(al)), back(al))	x0=back(al); al=join(x1,x0)
go to start	go to start
domach: x2=top; x3=back(al) if(prefunc(front(front (al))))dont	x2=top; x3=back(al) x0=prefunc(x1); if(x0)dont
valargs: x0=val(front(x3))	x0=front(x3); x0=val(x0)
push	push
execute: al=xeq(front(front(al)),x2)	al=xeq(x1,x2)
return(al)	return(al)
dont: x0=front(x3); x3=back(x3)	x0=front(x3); x3=back(x3)
push	push
if(x3)dont	if(x3)dont
go to execute	go to execute

The reader's analysis of Table I will be assisted by the following comments. The algorithm shown in Table I is written in RUDE. However, since the RUDE language as defined above does not permit the use of nested parenthetical expressions, and since the use of such expressions does serve to make the logical structure of an algorithm easier to follow, we employ the procedure of writing this RUDE algorithm in two forms, the first in a hypothetical extension of RUDE permitting nested parenthetical expressions, the second in the RUDE language as it has been defined above. The first version of the algorithm may be regarded merely as a set of comments intended to make the reading of the algorithm itself easier. In Table I, and in the similar tables which follow in the rest of the present section, we have arranged the two versions of the algorithm in parallel columns, with the parenthesized version on the left. Each line of the algorithmic form on the left corresponds to one or more similarly placed lines of formal RUDE statements on the right. The reader will probably find it most

convenient to examine the left hand or parenthesized version first, and to look at the expanded right hand version second.

A few detailed comments on the algorithm of Table I are in order. The basic list functions "front", "back" and "join" are used repeatedly in this algorithm. The attribute functions "atom", "machfnc" and "prefunc" are also used. Atom is assumed to return a non-zero value if and only if its argument is an item of any type other than list type. The attribute function machfnc returns a non-zero value if and only if its argument is an item of machine function type. The attribute function prefunc returns a non-zero value if and only if its argument is a machine function of prefunction type. The algorithm shown in Table I also makes use of an assumed machine function "xeq". This function is employed to set up and execute indirect machine function calls.

Finally, the subroutine 'errorroutine' is invoked at certain points in the algorithm shown in Table I. Note that this subroutine has no explicit arguments. We assume this subroutine to be a machine-written standard error procedure, which may be thought of as printing an appropriate diagnostic message, perhaps with additional diagnostic material, and returning control to the operating system through a call to thw system 'exit' routine.

LISP, in order to be of use as a programming language, must be supplied not only with internal conventions and basic algorithms like those outlined above, but also with a set of conventions for list input and output and for the initiation of processing action. Such an input scheme, taken in connection with the fundamental list evaluation algorithm described above, defines a programming language. Such a programming language will have the following general form and usage. A sequence of lists will be specified to the system, on whatever input medium the system uses, and in some particular form. Each list, after it is read in, is evaluated, i.e. the above "value" function is applied to it. This evaluation procedure requires a quantum of processing which may be small or large, i.e. which may consist on the one hand of a relatively simple initialization operation and on the other hand of a calculation leading at its conclusion to the printing of some desired final result. As a general rule, however, when the evaluation of one list read off the input medium is complete, the LISP processor will return control to its input routine and attempt to read yet one more list from the input medium. Only when an illegally formed data structure or an end-of-record mark is found on the input medium will control be returned to the operating system through a call to "exit."

Almost any context-free language may be interpreted as a LISP input language in the following way. Parse a sentence of the language; the result is a parse tree whose twigs represent the various lexical atoms occuring in the sentence. Now, a parse tree is a structure having almost exactly the form of a LISP list. We may therefore take each input sentence to represent its own parse tree. Lexical atoms are to be regarded as representing small lists, of some standard form depending on the lexical type of the atom, on which all relevant attributes of the atom are represented. Initially these attributes may merely be the print name and the lexical type of the atom; however, provision for the subsequent attachment of additional attributes, and, in particular, for attachment of the atom

value required by the fundamental evaluation algorithm should be made from the start. From the above point of view, the role of the special syntactic conventions which a particular input language will embody is merely that of facilitating the representation of list structures in common use (and of course, representing these in a manner having mnemonic value). Lists input in a given input language may, if desired, be transformed in any appropriate way before their submission to the basic evaluation algorithm; thus, the connection between an input form and the list which it represents can be made as flexible as necessary. LISP itself serves very well as a language for the expression of semmantic pre-transformations of this sort. The use of a powerful and general parser as a"front end" for a LISP system allows us to make input to the system in convenient, concise, and mnemonic form. On the other hand, if one is willing to write lists for input in a restricted and somewhat clumsy way, the parse aspect of LISP input can be reduced to a process which is almost trivial; LISP input is, as a matter of fact, more often accomplished by the use of the trivial conventional list notation which we are about to describe, than by the use of any more adequate input language.

The straightforward input scheme ordinarily employed by LISP makes use of a list notation which we will also find useful in defining other basic LISP functions. A list to be input is represented on the system input medium as a parenthesized structure built up out of symbolic names, constant data items (e.g. integer and real numbers), and parenthesized substructures. The occurrence of a name in the input string results internally in the construction of a list item element of the form shown in Figure 2.

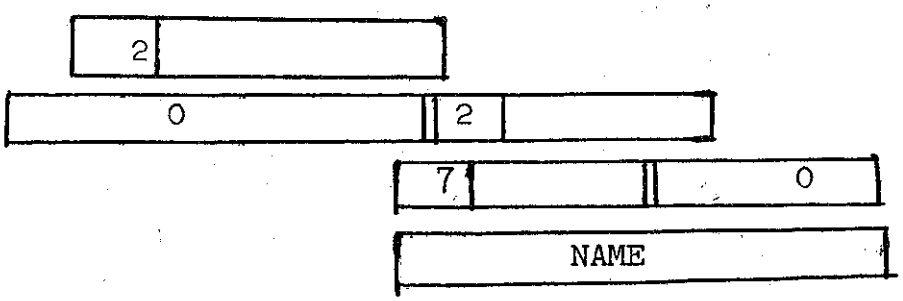


Figure 2. Atomic List Item Generated by a Symbolic Name.

Note that the left half of the node referenced by the top atomic list item in Figure 2 is the value cell of the atomic list item; this value cell may either be initialized to the value NIL, or, still better, to a reserved value 'empty' district from NIL, this latter procedure permitting subsequent detection of illegal occurences of unbound variables. The right hand side of the same node consists initially of an atomic list item which references a further node whose front part is of BCD type and references the symbolic name encountered in the input string, transformed to its internal BCD representation. The list structure referenced by the top atomic list item in Figure 2 is often called the property list of the atomic list item. During subsequent processing this property list is normally used for the storage of attributes of the atomic item as these may be developed, with the consequence that the structure of the property list may change considerably. Only two conventions concerning the structure of the property list must be observed; in the first place, the front item referenced by an atomic item will always be used to supply its value in the sense of the preceding algorithms; in the second place, we require as a standard output convention that the BCD name of a named atom always be the first item on its property list.

Each time that a new symbolic name is encountered in the LISP input string not only will an internal structure of the type shown in Figure 2 be created, but a reference to the top item of Figure 2 will also be entered into a comprehensive symbolic atoms table. This table may appropriately be maintained as a hash table. On encountering a symbolic name in its input stream, the LISP input processor will check the symbolic atoms table to see if the newly encountered name has been encountered before, If this is the case, the symbolic name will be identified with its prior occurrence and the internal item corresponding to the symbolic name will reference the old property list, so that no new property list will be constructed.

Three particular reserved names, NIL, T and F are treated specially by the input processor. On encountering one of these names the input processor constructs not a symbolic atom item of the type shown in Figure 2, but an immediate data item, zero in the case of NIL, and 1 in the case of T. The reader will observe that these two distinguished immediate items play a special role in the fundamental LISP algorithms represented above. The name F is treated as a synonym for the name NIL.

The occurrence in the LISP input stream of a data item other than a name produces an atomic item of the simple form shown in Figure 3.

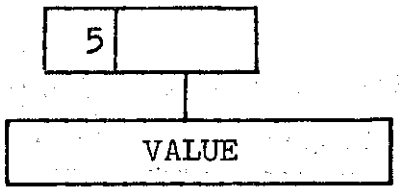


Figure 3. Internal Item Corresponding to Integer Data Item.

The most general allowable input list will be made up out of names, data items, and subroutines in the following manner. A structure having the form

(11) (a1 a2 ... an) ,

where a1, a2, ...,an are all either names, integer or real numbers, consistitutes an elementary input list. A structure of this same form in which a1,a2,...,an are either names, real or integer constants, or composite input list structures, is a composite input list structure. All input to the LISP system is required to have the form of a composite list structure. Moreover, the composite list structures which may be used as LISP input are required to have the particular form shown in

(12) f(a1 a2 ... an)

In (12), we require that f be a simple name, and that a1,a2,...,an be composite input list structures. The input routine automatically transforms the list structures shown in (12) above into the modified input structure shown in (13) below.

(13) (f (quote a1) (quote a2) ... (quote an))

In (13), "quote" is a predefined atom, whose value is the identity machine code prefunction with the trivial algorithm shown in Table II.

Table II. Algorithm for the "quote" function.

function quote	x1=front(a1)
return(front(a1))	return(x1)

The input structure shown in (13) is then transformed into an internal list structure, as follows. Each atomic name and constant occurring in structure (13) is translated into internal form in the manner explained above. The whole structure is then translated into internal form by a recursive procedure which may be described as follows. Designate the internal form of a structure by placing a bar over the structures, so that $\overline{a}$ designates the internal form of the input structure a. Then we define the internal form of the general composite input structure

(14) (a1 a2 ... an)

by the rule

$$(\overline{a1}) = \text{join}\ (\overline{a1},\ \text{nil})$$

$$(\overline{a1\quad a2\quad \ldots\quad an}) = \text{join}\ (\overline{a1},\ \overline{(a2,\ldots,an)}) \tag{15}$$

If, as is sometimes done, we write the join operation more compactly with an infixed dot, the rule (15) may be written in the more customary form

(16) $\overline{(a1 \quad a2 \quad \ldots \quad an)} = \overline{a1} \cdot (\overline{a2} \cdot \ldots (\overline{an} \cdot nil))$

Note that in terms of this notation we have

(17)
$$\overline{(a)} = (\overline{a} \cdot nil),$$
$$\overline{((a))} = ((\overline{a} \cdot nil) \cdot nil),$$
$$\overline{(((a)))} = (((\overline{a} \cdot nil) \cdot nil) \cdot nil), \ldots ;$$

whereas

(18) $(((a\ b))(c\ d)) = ((a \cdot (b \cdot nil)) \cdot nil) \cdot ((c \cdot (d \cdot nil)) \cdot nil)$,

and so forth.

The LISP input process concludes by applying the function "value" to the internal form of the input structure (13). To repeat: input is presented to the LISP system in the form (12); it is at once transformed by the LISP system from the form (12) to the form (13); the structure (13) is converted to its internal representation according to the above rules, and the function 'value' applied to it. Application of 'value' in this manner may of course initiate a considerable amount of processing. Moreover, since this processing may include calls on routines like 'setval' which affect permanent changes in the data structures maintained by the LISP system, the internal list structure of which the LISP processor is aware may be considerably changed by the operations set in train by input of a LISP input item.

Output from the LISP system is accomplished by a programmed 'print' function. This function normally produces output identical in form with what has above been called the composite inuut structure. Where output in this form is impossible, output is normally printed in a notation derived from the 'dot' notation used on the right hand sides of (16), (17) and (18).

Once the input and output conventions outlined above are established, LISP becomes a usable programming language, all but one which is unpleasantly homogeneous in appearance.

In what follows, we shall make use of the standard expository convention in terms of which internal LISP lists are represented in the above "input" form, or, as it is sometimes called, the list notation. This will at least give us a notation in which lists may be written. Note however that this notation is incomplete: it shows the manner in which a list is put together in terms of atoms, but does not give any information concerning the property lists of the atoms which occur in the list.

Concerning the list notation we observe in the first place that

(19) join((a b c d), (e f g h)) = ((a b c d) e f g h) .

Let us, in order to explore our notational conventions, calculate the value of a list of the form

(20) (((a b))) ,

where, for the sake of definiteness, we suppose that "a" has the value "quote". The list (20) may be written in dot-notation as

(21) (((a · (b · nil)) · nil) · nil) .

According to the algorithm of Table I, the value of (21) is the same as that of each of the sequence of lists

(22) ((a · (b · nil)) · (nil· nil)) ,
(a · ((b · nil) · (nil · nil))) ,

i.e., in list notation

(23) ((a b) (nil)) and (a ((b) nil)) .

Since a has the value "quote", the value of (23) is calculated as that of

(24) (quote ((b) nil))

according to the algorithm of Table I; since quote is a prefunction with the algorithm shown in Table II, we obtain

(25) ((b) nil)

or equivalently

(26) ((b · nil)· (nil · nil))

as the value of (20).

The function lambda, an algorithm for which is given in Table III, is almost as basic to the operation of the LISP system as the value function defined by Table I. This function performs the basic operation of attaching temporary values to atoms for use by the value function of Table I. Since the LISP system is required to be fully recursive, lambda also saves the old value of every atom for which it supplies the new value, and restores these old values before it returns. The specific conventions used by lambda are as follows. It receives a single argument which is required to be a list item referencing a list of two parts, whose front part is however, also a two-part list. The 3 items mentioned above are all used by lambda in an essential way. Let al be the argument supplied to lambda: write x1 = front((front)al), x2 = back(front(al)) and x2 = back(al). Then x1 is required to be a list item referencing a list of atoms while x2 is required to be a list item referencing a list of expressions. Lambda saves the values of all the atoms on the list x1 and supplies to each of these atoms as its new value the value of the corresponding item on the list x2. After this, "atom value substitution" operation has been performed, lambda calls the value function of Table I, supplying the item x3 as its argument and obtaining a

```
          function lambda
          x2=back(a1)                          | x2=back(a1)
          a1=back(front(a1))                   | x1=front(a1);a1=back(x1)
          x1=front(front(a1))                  | x1=front(x1)
          x3=front(x1);x4=front(x2)            | x3=front(x1); x4=front(x2)
          x5=nil                               | x5=nil
bind:     x0=front(x3); push                   | x0=front(x3); push
          x0=ident(x3); push                   | x0=ident(x3); push
          x0=repfront(x3,val(front(x4)))       | x0=front(x4); x0=val(x0)
                                               | x0=repfront(x3,x0)
          x3=back(x3); x4=back(x4)             | x3=back(x3); x4=back(x4)
          x5=incr(x5)                          | x5=incr(x5)
          if(x4) bind                          | if(x4) bind
          x0=ident(x5); push                   | x0=ident(x5); push
          a1=val(a1)                           | a1=val(a1)
          x3=valtop; pull                      | x3=valtop; pull
unbind:   x1=valtop; pull                      | x1= valtop; pull
          x2=valtop; pull                      | x2=valtop;  pull
          x0=repfront(x2,x1)                   | x0=repfront(x2,x1)
          x3=decr(x3); if(x3)unbind            | x3=decr(x3); if(x3)unbind
          function(a1)                         | function(a1)
```

Table III. Algorithm for the function 'lambda'.

2. The Snobol String Manipulation Language.

SNOBOL is an interesting string processing language organized around a concept of 'pattern matching'. The language has developed through several stages, in consequence of which various more or less general versions exist. The particular version of SNOBOL that will be described in the present section is the SNOBOL-3 language as specified by D. J. Farber, R. E. Griswold and I. P. Polonsky, in "The SNOBOL-3 Programming Language", Bell System Tech. Jour., Vol. XLV, No. 6 (1966) 895-944.

SNOBOL differs considerably from the languages of the FORTRAN-ALGOL-PL/1 algebraic class both in the nature of the data with which it is concerned, and in the nature of the statements out of which complete programs are composed.

In the SNOBOL language every datum and the value of every variable is a character string. Only one basic statement, the string matching-replacement statement, is provided in the SNOBOL language. The language provides this statement both in a full and in several degenerate versions which will be described in detail in the present section. The main data structures maintained by the SNOBOL processor may be described as follows. All string data known to the SNOBOL processor is collected into one general body of BCD text; this data body may be called the comprehensive text array. Each individual string known to the processor is represented as a substring of this text array. A named string of the SNOBOL language is defined by a pair of substrings of the text array, the first of these substrings being the name of the string, and the second of these substrings being the value of the string. Thus a named string is represented to the SNOBOL processor by the set of four fields indicated below:

(1)

name 1	name 2	value 1	value 2

A set of four fields as shown in (1) together constitute a "string descriptor". In (1), 'name 1' is a pointer to the first character of the name string of the descriptor, and 'name 2' is a pointer to the last character of the name string of the descriptor. Similarly, 'value 1' and 'value 2' are pointers to the first and last characters respectively of the value string of the descriptor. All of these pointers of course reference particular characters in the comprehensive text array. If, at a given point in a SNOBOL computation, any of this information has not yet been established, the corresponding field is set equal to zero. Thus, for example, if no specific value has yet been assigned to a named variable in a SNOBOL program, the variable is considered to have a nominal null string value, indicated by setting both fields 'value 1' and 'value 2' in its descriptor to zero.

As the statements in a SNOBOL program are successively executed, new values may be assigned to a previously named SNOBOL variable. This is accomplished by changing the 'value' fields of the unique string descriptor of the variable. New variables may also be defined and assigned values. This is accomplished by setting up a new string descriptor with the proper 'name' and 'value' fields. Finally, additional text may be generated and added to the comprehensive text array.

A SNOBOL program is organized as a sequence of statements, each of which specifies the execution both of a basic string operation and of a (possibly degenerate) conditional transfer. The fundamental SNOBOL string operation is that of matching a reference string to a pattern. Any SNOBOL string may serve as the reference string for such a matching operation. String constants in SNOBOL are written by enclosing their literal string in quotations. Thus,

(2) "THISISASNOBOLSTRINGCONSTANTWITHOUTSPACES"

is an example of a SNOBOL string constant, as is

(3) "THIS IS A SNOBOL STRING CONSTANT WITH SPACES"

The simplest form of SNOBOL pattern is an interspersed sequence of string constants and string variables. A string constant in a pattern is written, just as any other string constant, by enclosing its value in quotation marks. Several different types of variables are provided in the SNOBOL language. The simplest of these is the named general variable, written simply by enclosing the name of the variable in a pair of asterisks. Thus *X*, *Y* ,*Z*, *XX*, *YY*, etc., are SNOBOL named general variables. A SNOBOL pattern involving the first three of these variables is shown in (4).

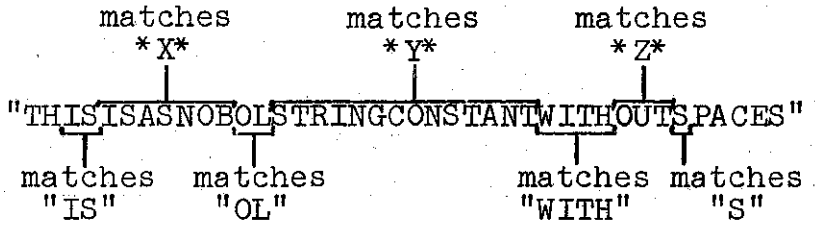
(4) "IS" *X* "OL" *Y* "WITH" *Z* "S"

In matching a pattern like (4) to a reference string like (2), one attempts to match every part of the pattern to a part of the reference. To match a constant occurring in the pattern requires that a precisely identical portion of the reference be found. A variable in the pattern, on the other hand, is taken to match any null or non-null string of successive characters in the reference string; if several alternative matches are possible, a variable is always taken to match the shortest possible substring of the reference string. If a match of every element of a pattern to a corresponding part of a reference string is possible, the pattern match is successful, and a system success-failure flag is set to indicate this fact. Moreover, in case of a successful match, every variable in the successfully matched variable is assigned a new value, namely, the part of the reference string which

it matches. If, on the other hand, not every element of the pattern can be matched to a part of the reference string, the attempted match is said to fail, and the SNOBOL system success-failure flag is reset to a "fail" condition. In this case, no new value is assigned to the variables occurring in the pattern.

As an example, we may consider the application of the SNOBOL matching process to the reference string (2) and the pattern (3). The reader may verify that in this case a match is possible. The details of this match are shown in (5):

(5)



The successful match shown in (5) would assign the string value "ISASNOB" to the name "X", the string value "STRINGCONSTANT" to "Y", and the string value "OUT" to the name "Z".

In addition to the fundamental matching operation, whose main features are described above, the SNOBOL language admits string expressions and operations described by such expressions. String expressions may be formed from string names and quoted string constants using either concatenation, which applies to arbitrary strings, or using the arithmetic operations +, -, *, /, ** which apply only to string variables whose values represent valid integers. (Note in particular that arithmetic is handled on a character string basis.) Concatenation is signified by the successive placement, separated by blanks, of the string variables and string constants whose values are to be concatenated. A SNOBOL expression may also involve a call to a string function of one or several variables written in the ordinary prefix style. The following is a typical SNOBOL expression:

(6) "HØUSE" "AND" JIM ("-3" * "2" + "7") F(I,J)

In the above expression 'JIM' is a variable and 'F(I,J)' is a function. By concatenating the value of the variable 'JIM' and the value of the function F(I,J) with the preceding constants and with the value of the arithmetic subexpression "-3" * "2" + "7" (this value is of course the string '1'), we obtain the value of the whole expression.

We have now described all the principal elements of the SNOBOL language, and can go on to describe the general SNOBOL matching-replacement statement.

The language provides this statement both in a full and in several degenerate versions. The full matching-replacement statement consists of six parts: a label part, a string reference, a pattern, an equals sign, a replacement part, and a routing indicator part.

1) The first of these six parts, the label, is optional. If present, it labels the statement in which it occurs, so that transfer may be made to the statement from other points in a SNOBOL program.

2) The string reference part is a string expression, evaluation of which is the first step in execution of a nondegenerate matching-replacement statement. Evaluation of the reference expression yields the string to which the remainder of the SNOBOL matching-replacement statement will refer for matching and for replacement; this string may appropriately be called the reference string.

3) The pattern part of a matching-replacement statement consists of a sequence of string expressions and string variables. In executing a matching-replacement statement, the SNOBOL processor attempts, in the manner indicated above and described in more detail below, to match the pattern against the reference string. A match attempt will either succeed or fail. Depending on

the outcome of an attempted match, a success-failure flag is set. If the match succeeds, the match processor also assigns new values to any variables occurring in the pattern. Additionally, a successful pattern match defines a particular substring of the reference string, to wit, that part of the reference string which matches the pattern. This substring, defined at the conclusion of a successful pattern match, may be called the designated substring of the reference string.

4) The replacement portion of a matching-replacement statement is an expression which is evaluated following a successful pattern match and whose value replaces the designated substring of the reference string.

5) The final part of the full SNOBOL matching-replacement statement is the routing part. This part is delimited by a prefixed slash following which one or two labels indicating the points to which transfer is to be made on success or on failure of the preceding steps of calculation and matching.

Some of the parts of the complete matching-replacement statement described above are optional; by omitting the various optional parts we obtain the various degenerate versions of the statement.

i) The label part of a SNOBOL statement may be omitted.

ii) The reference part of a statement may be omitted, but only if the following pattern, equals sign, and replacement part are also omitted. In this case the matching-replacement statement consists only of a routing portion and acts as a conditional GO TO statement.

iii) If the reference is not omitted, the pattern, the replacement part, and the routing part may each either be omitted or be present, except that if the equal-sign separating the pattern and replacement part is omitted, the replacement part must also be omitted. If the equal-sign is present and no explicit replacement part occurs, the

SNOBOL processor will assume that the null string is intended as the replacement part. Omission of the pattern reduces the matching-replacement statement to a string assignment statement followed by a conditional transfer statement.

iv) If the replacement part is omitted, pattern matching will still be carried out, and, on successful match, assignment of new values will be made to the variables occuring within the pattern. However, no replacement of the designated substring of the reference string will be made.

v) If the routing part of the SNOBOL statement is omitted the SNOBOL processor will, on completing execution of the matching and replacement parts of the statement, pass immediately to execute the next following SNOBOL statement.

The routing portion of a SNOBOL statement begins with a slash which separates it from the earlier portions of the SNOBOL statement. One or two routing expressions will follow this slash. The allowable routing expressions have the form

(a) S(<expression>)
(b) F(<expression>)
(c) (<expression>)

The letter "S" has the semantic significance: transfer if the system success-failure flag is in the success condition. Similarly, "F" requests transfer if the success-failure flag is in the failure state. The presence of parentheses but the absence of either of the letters S or F requests unconditional transfer. The expression occurring in parentheses in either (a), (b) or (c) above may be an arbitrary string expression.

The following sample SNOBOL statements will illustrate the above discussion.

(7) HERE TEXT " " *WØRD* " " = " " /S(GØT)

Statement (7) is a labeled SNOBOL statement. The value of the named string TEXT will be the reference string during

the execution of statement (7). In case of successful match, the designated substring will be replaced with a single blank, and the statement labeled with the label GØT will be executed next. If the match in (7) fails, the next following statement in the program containing statement (7) will be executed.

(8) STR = "FØUL" " " "BALL"

Statement (8) contains no pattern, and hence reduces to a string assignment statement. The value of the reference string STR becomes "FØUL BALL".

The syntax and the semantics of the SNOBOL expression are both straightforward. As noted above, the only infix operations provided are the basic arithmetic operations and the string concatenation operation. During the action of the SNOBOL processor, every operation not only returns a value but sets a success-failure flag which is available for subsequent use by the SNOBOL processor. Thus, for example, an attempt to apply an arithmetic operation to a pair of strings, one of which has a value which is not a valid integer, as, for example, "A23456" will set the failure flag, terminate evaluation of the expression and cause the SNOBOL processor to transfer to another SNOBOL statement. Arithmetic expressions may be enclosed in parentheses. A left parenthesis beginning an arithmetic expression must always be preceded by a blank space.

The prefixed monadic dollar sign operator ($) is used to signify indirect reference within a SNOBOL expression, i.e., $JIM designates the value of the variable whose name is equal to the value of the variable JIM. The treatment of this "indirection operator" is best understood by reference to the underlying operation of the SNOBOL processor. At any given moment the SNOBOL interpreter is aware both of a body of accumulated text contained in the text array, and of a comprehensive collection of string descriptors of the form shown in (1), contained in a dynamically varying string

descriptor table. In evaluating string expressions, and for other purposes as well, the SNOBOL interpreter makes use of a basic location function DESCRIPT, which, given a string value, finds a descriptor in the descriptor table whose name string is identical to the given string; if none such exists, DESCRIPT returns an indication of this fact. (The descriptor table is most appropriately maintained as a hash table; a hashing technique increase the efficiency of the basic DESCRIPT function above what it would otherwise be). The dollar sign represents externally what internally is a call on the basic function DESCRIPT.

SNOBOL expressions may also contain function calls, written in the ordinary prefix style. A function call in a SNOBOL expression consists of a name, followed without any intervening space by a left parenthesis, followed by a string of arguments, followed by a terminating right parenthesis.

Function calls are recursive, and involve the following conventions. A call on a function must be preceded by a call to the / SNOBOL 'DEFINE' function. This function call is required to have the form

(9) DEFINE(<namer>,<label>,<local variable definer>)

In (9), 'namer' must be a string expression whose value has the form: name(argument 1, argument 2, ...). The string 'name' is the assigned name of the function being defined; argument 1, argument 2, etc. are the names assigned to the argument variables of the function. The entry 'label' in (9) is a string expression whose value determines the entry point to the code sequence which is to be used for calculating the function. The 'local variable definer' which forms the final portion of the DEFINE statement must, if present, be a string expression whose value is a list of variable names whose values are to be

saved on entry into the function and restored on return from the function. However, the local variable definer may be omitted. The SNOBOL DEFINE function is dynamic in the sense that when executed it either causes a new entry to be made in a function table (c.f. below) or causes an old entry in this table to be modified. The use of the general form (9) is illustrated by the following particular examples:

(10) DEFINE ("REVERSE(X)","REV")

(11) DEFINE("JØIN(X,Y)","JINE","A,B,C,") .

When a function is invoked, the address to which return is to be made on completion of the function calculation is saved on a pushdown stack. The value of all the function arguments and of all the local variables belonging to the function are also stacked. The value of each of the declared arguments of the function is set to the value of the corresponding actual argument; the value of each of the declared local variables of the function is set to the null string. The function name is treated as a variable name, and its value set equal to the null string. The last value assigned to the function name before return from the subroutine determines the value returned by the function. On completion of whatever internal calculation the function is to perform, return to the calling point is accomplished by a nominal transfer to one of the two dummy labels RETURN and FRETURN. Either of these nominal transfers returns to the last previously stacked return address. Nominal transfer to the nominal RETURN label sets the system success-failure flag to the success condition; corresponding transfer to the FRETURN label sets the system success-failure flag to the failure condition.

Three types of variables are provided in the SNOBOL language: the general variable, the parenthesis-balanced variable and the fixed-length variable. Any or all of these variable types may be used in forming a SNOBOL pattern. The form of a general variable is

(12) *<expr>* or **

In (12) 'expr' denotes a string expression which is to be evaluated and whose value defines the name of the named variable. The second form shown in (12) is the form of a null-named ordinary variable. The form of a parenthesis-balanced variable is

(13) *(<expr>)* or *( )*

Here again 'expr' is an expression whose evaluation determines the name of the named variable; the second form shown in (13) is the form of a null-named, parenthesis-balanced variable. The form of a specified length variable is

(14) *<expr 1> / <expr 2>* or * /<expr 2>*

In (14) 'expr 1' is an expression whose evaluation determines the name of the specified length variable, while 'expr 2' is an expression, whose value must be a string denoting a valid positive integer, which integer determines the length of the specified length variable. The second form of the specified length variable shown in (7) is that in which the name of the variable is null.

The following example of a SNOBOL pattern illustrates various of the possibilities discussed above.

(15) "A" * * $BUMP *(JIM JØE + '1')* *JIM/JØE + "1"*

The pattern (15) consists of the coded string constant A, followed by a null variable, followed by the expression $BUMP which may be evaluated by a single indirect reference, followed by a parenthesis balanced variable whose name is to be calculated by evaluating a SNOBOL expression, followed by a single specified length variable.

The matching of a pattern to a reference in the execution of a SNOBOL statement proceeds as follows. The expression constituting the reference is evaluated. If this evaluation fails at any point, all further execution of the matching-replacement statement is suppressed and transfer is

immediately made to whatever point is indicated by the routing portion of the replacement statement. If, on the other hand, the reference can be successfully evaluated, its value determines the name of the string against which the pattern is to be matched. All expressions included in the pattern or in a variable of the pattern are then evaluated. If any one of these evaluations fails, all further matching is broken off and transfer is made as indicated by the routing portion of the statement. If, on the other hand, all the individual parts of the pattern are successfully evaluated, a matching process is invoked: the processor employed to carry out the matching attempts to match the pattern to the calculated string reference.

Pattern matching then proceeds as follows. The values of all the variables occurring in the pattern are first saved. This initial step is necessary since, on the one hand, if the match which is to be attempted turns out to be successful, new values will be assigned to all of the variables occurring in the pattern; if, on the other hand, the match turns out to fail after a particular success, the initially saved values of all the variables in the pattern must be restored.

Pointers indicating the number of elements of the pattern which have been successfully matched and the part of the reference string which has been matched are initialized. A length variable is set equal to the length of the reference string. The first pattern element is then taken up, and an attempt made to match it to the shortest possible section of the reference string beginning at the first character of the reference string and extending as far to the right in the reference string as is necessary to secure a match. The precise conditions for a match depend on the nature of the pattern element. A non-variable element will only match a substring of the reference pattern which is identical to it, character for character. A variable occurring in the pattern will match either an arbitrary substring of the reference string, or, if the variable is specified as being

parenthesis-balanced or fixed length, the variable will match only such substrings of the reference string as have the specified property.

If the first element of the pattern is successfully matched to a substring of the reference string an attempt is immediately made to match the next pattern element; this next pattern element is required to match a part of the reference string immediately following the part matched by the preceding pattern element, and extending to the right only as many characters as are necessary to secure a match. Proceeding in this way, the match processor attempts iteratively to match every element of the pattern to a part of the reference string. As soon as a variable is successfully matched, the substring of the reference string which it matches is assigned to the variable as its new provisional value. If every pattern element is successfully matched, the match as a whole is taken to be successful. On successful completion of the matching operation the SNOBOL success-failure flag is set to the success condition; this flag is used subsequently in interpreting the routing portion of the matching-replacement statement to determine the next SNOBOL statement to be executed. The successful completion of a match also defines the designated substring of the reference string as that subpart which is matched by all the pattern elements together. If the matching-replacement statement contains an equal-sign, the replacement expression following the equal-sign is then evaluated. If the system success-failure flag is set to 'failure' at any point during the evaluation of this replacement expression, the replacement operation is broken off and the interpreter proceeds at once to execute the routing portion of the matching-replacement statement. If, on the other hand, evaluation of the replacement expression is successful, the string value of the replacement expression replaces the designated substring of the reference string, thereby assigning a new value to the reference variable. In this case the success-failure flag is left set to the success state

and the routing portion of the matching-replacement statement is executed.

We have next to consider the procedure employed by the SNOBOL interpreter when it finds that a part of the pattern which it is attempting to match to a substring of the reference string cannot be so matched. When this occurs, the match processor returns to the last preceding pattern variable and attempts to expand the substring of the reference string to which this last preceding variable has been matched. Such extension is logically possible if the variable is a general or a parenthesis-balanced variable; the general variable can be matched to a longer substring of the reference string simply by matching it to a substring one character longer than has been used for its preceding match; a parenthesis-balanced variable may have to be matched to a substring which is considerably longer in order that the condition of parenthesis balance be respected. (Of course, the substring to which a fixed length variable is matched cannot be extended in length). Extension of the substring matched by a general variable will be impossible only if the present match of the variable extends all the way to the end of the reference string. If the match of a preceding variable can be extended in the above manner the match processor again attempts to match the following elements of the pattern. In this way successful match of every element in the pattern may be attained after a number of initial failures. If, on the other hand, extension of the match of the last preceding variable is impossible, the pattern processor retreats yet further back and attempts to extend the match of a still earlier variable of the pattern. If the match processor is in this way driven back to an attempt to rematch the very first pattern element, or if no further extension of the match of the first pattern element is possible, the match processor will attempt to rematch the first pattern element by starting the match of this element one character further to the right than has been previously been attempted.

Only when it has been determined that no possible initial match position for the first pattern element can possibly lead to a successful match of the whole pattern will the match processor finally decide that it is impossible for the pattern to be wholly matched to any portion of the reference string. In this case, the interpreter success-failure flag is set to 'failure' and the SNOBOL interpreter proceeds at once to interpret the routing portion of the matching-replacement statement.

It is to be noted that the SNOBOL system is highly interpretive, in the sense that its basic operations are performed by complex system routines rather than by short sequences of machine operations, and in the sense that certain classes of logical objects (e.g., transfer labels) which are determined at compile time in a language like FORTRAN may be dynamically varied during execution in the SNOBOL language. This circumstance makes SNOBOL somewhat more flexible than a less interpretive language could be. For example, since arithmetic expressions are evaluated interpretively, software switches which vary the interpretation of the basic arithmetic constructions can be and are provided. This makes it possible to establish several SNOBOL arithmetic modes, which will be discussed in more detail below.

A mode switch controlling an aspect of the matching process is also provided within the system. This switch may be called the anchor-unanchor mode switch; it controls the procedure employed by the pattern matching processor in attempting to match the first element of a pattern. In the "unanchor" mode, the first element of the pattern is allowed to match a substring of the reference string beginning at any character of the reference string. In the "anchored" mode the first pattern element is required to match a portion of the reference string beginning at its first character. This mode switch is toggled by a pair of dummy string function calls, MODE("ANCHOR") and MODE("UNANCH"). In view of the frequency with which

transitions between anchored and unanchored mode are required within SNOBOL programs, an alternative version of the anchored mode, valid only during the execution of a single statement, following which the anchor-unanchor flag is returned to its previous setting, is provided. A temporary matching mode of this sort is established by one of the two pseudo-function calls, ANCHOR( ) and UNANCH( ), which have null string arguments and return the null string as a dummy value.

Table I below gives a formal algorithm for the match procedure described informally in the preceding paragraphs. The algorithm is divided into two parts, a principal subroutine "MATCHER" controlling the overall flow of the match process, and a secondary subroutine "MATCHES " which tests a single section of a reference string for match with a single pattern element.

The following comments will assist the reader in comprehending the algorithm shown in Table I. The matching processor makes use of a match pointer stack and of a pattern parts stack during the process of matching. These stacks have the form indicated diagrammatically in Figure 1.

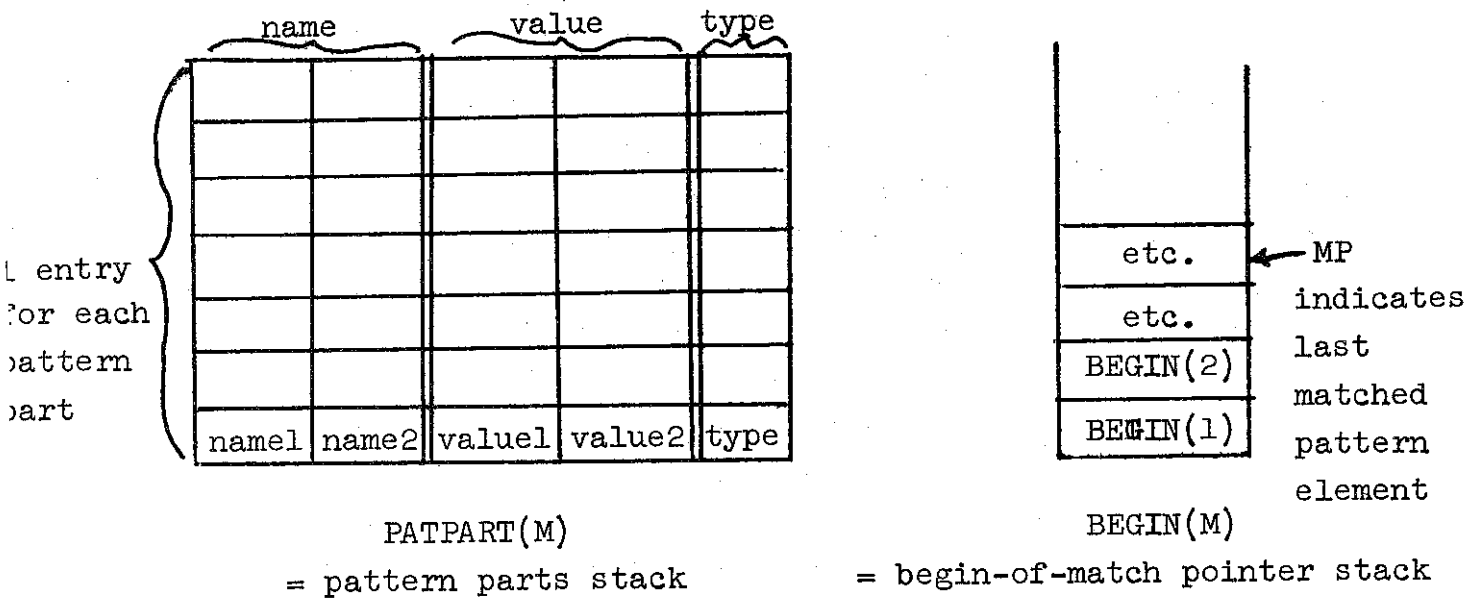


Figure 1. Information stacks used by matching process

The array PATPART is initialized to reference all the pattern elements to be matched, and to give the type (constant, general variable, balanced variable, etc.) of each pattern part. The array BEGIN is initialized to zero. As the match proceeds, BEGIN(J) is set to indicate the beginning of the section matching the J-th pattern part. When the match processor falls back from an attempted match, the corresponding element BEGIN(J) is set to zero. On the other hand, if the (J-1)-st element is a variable for which a prior match has been attempted, the non-zero value BEGIN(J) is saved to indicate the character from which the earlier variable match must be extended. The variable MP defines the next pattern element for match; NOPAT is the total number of elements in the pattern. MODE is the permanent match-mode switch; TEMPMODE is a temporary (one-statement) match mode switch; and TEMODE is a flag indicating whether or not the TEMPMODE switch is effective. TYPE(J) is the type of the J-th pattern element as taken from the appropriate field of the array PATPART. BALANCED is a computed attribute which holds if the reference string section which its arguments reference is parenthesis-balanced.

Table I: SNOBOL match process algorithm

```
            Subroutine MATCHER(Ref,Nopat,Patpart)
            Save values of all variables in Patpart
            (Nopat = Number of pattern elements)
            MP = 0
            Zero all entries in Begin stack; Begin(1) = 1
            Len = length of reference string
Suc:        MP = MP + 1
            If MP > Nopat go to Okdone
Match:      If (MATCHES(MP)) go to Suc
Fail:       MP = MP - 1.  Begin(MP+1) = 0
            If (MP < 0) go to Nokdone;
            If (Type(MP)=const. or flv.) go to Fail
            If (MP > 0) go to Match
            If (Temode = 1) and (Tempmode = Anch) go to Nokdone
            If (Mode = Anch) go to Nokdone
            Else Beg = Begin(1) + 1
            If (Beg > Len) go to Nokdone
            Else Begin(1) = Beg; go to Match
Okdone:     Matchedsec = [Begin(1), Begin(Nopat)]
Droptem:    Temode = 0.  Return
Nokdone:    Restore values of all saved variables
            Go to Droptem.
```

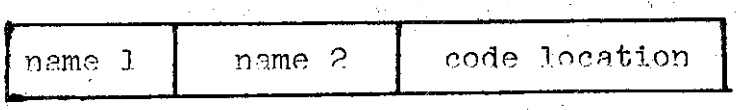
```
            Function MATCHES(MP)
            Low = Begin(MP)
            Go to (Const,Flv,Ordv,Balv) by type(MP)
Const:      Top = Low + Constlen; If top ≤ Len go to Idchek
Nomatch:    Matches = No; return
Idcheck:    Check for identity with following string.
            Go to Nomatch if no.
Setnext:    Matches = Yes; Begin(MP+1) = TOP+1
Flv:        Top = Low + Varlen
Over:       If (Top > Len) go to Nomatch
Setvl:      Value.Patpart(MP) = [Low,Top]; go to Setnext
Ordv:       If Begin(MP+1)=0, Top = Begin(MP) and go to Setvl
            Else Top = Begin(MP+1)+1; go to Over

Balv:       If(Begin(MP+1)=0) Begin(MP+1) = Begin(MP)
Bigger:     Top = Begin(MP+1)+1.  If (Top > Len) go to Nomatch
            If (BALANCED(Low,Top)) go to Setvl
            Else go to Bigger.
```

The SNOBOL interpreter handles the final or routing part of a matching-replacement statement as follows. Any string expression occurring in the routing part is evaluated. A failure condition arising during this evaluation is considered to be a programmer error and leads to the printing of a diagnostic message and the termination of SNOBOL execution. If, on the other hand, all expressions occurring in the routing portion of a SNOBOL statement are successfully evaluated, a search is made of a label table (see below) to find a label matching the appropriate transfer label. Failure to find such a label is again treated as a fatal programmer error. If, on the other hand, a matching label is found, the SNOBOL interpreter continues execution from the transfer point corresponding to this label.

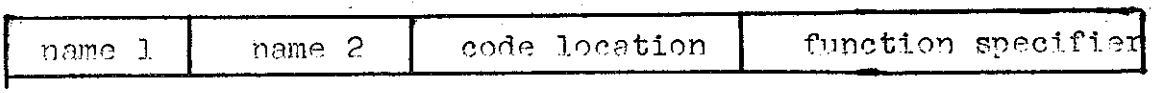
As indicated above, the two main data structures used by the SNOBOL interpreter are the text array and the descriptor table. For the treatment of transfers and of subroutine calls, the SNOPOL processor requires two additional tables, a label table and a function table. The label table consists of a set of entries of the form shown in (3L).

(3L)



The function table consists of a set of entries of the form shown in (3F).

(3F)



In (3L) and (3F), 'name 1' and 'name 2' are pointers to the first and last character of the label or name respectively. Moreover, the third field, 'code location', indicates the location within the fixed collection of SNOBOL instructions of the point referenced by the given label or function calls. The fourth field occurring in (3F), i.e. the "function specifier" field, is a pointer to the location of additional information concerning the function; this additional information includes, for example, a list of arguments for the function. Two basic SNOBOL processor functions LDESCRIPT and FDESCRIPT are used to locate entries in the label table and the function table respectively. These two routines operate in much the same manner as the DESCRIPT routine discussed above but reference the label table and the function table rather than the variable descriptor table. A flexibly expendable collection of built-in functions is provided within the SNOBOL system by allowing an initializing program to preset positions in the function table which reference the entry points to built-in routines.

7.3 SIMSCRIPT - An Algebraic language adapted for simulation.

SIMSCRIPT is a special purpose algebraic language with added list processing and statistical features intended to facilitate the writing of simulation programs.

The original SIMSCRIPT system for the IBM 7094 was developed by Markowitz, Hausner, and Karr of the Rand Corporation. In what follows, we shall describe a version of the SIMSCRIPT language agreeing with this original system in most essential respects but differing in the direction of generality and simplicity in some details.

SIMSCRIPT is organized around an internally maintained list of timed future events. The simulated occurrence of successive events drives the SIMSCRIPT processor to its various actions. In addition exogeneous events collectively constituting an exogeneous event file may be referenced during simulation and effect the flow of the simulation process.

The basic structure of the main SIMSCRIPT simulator routine is as follows. The ordered list of future events is referenced and the time of the next event, either exogenous or endogenous, is determined. An internal simulation clock is then set to the time of this next event. If the event is endogenous it must be described by an endogenous event routine of the sort which will be described below. If the next event is exogenous the details of the event are determined by reading the exogenous event file. Any action corresponding to the next event, whether it be exogenous or endogenous, is taken, following upon which the same basic simulation step is repeated iteratively until no further endogenous or exogenous events are found.

The events with which the SIMSCRIPT main routine deals belong, from the viewpoint of the general structure of the SIMSCRIPT language, to a more general class of temporary entities. A SIMSCRIPT entity is a data element with a declared subfield structure. The necessary storage for an

entity is dynamically allocated when the entity is created within the SIMSCRIPT system, and is released for re-allocation when the entity is abolished.

In more detail: the SIMSCRIPT system is organized around a collection of basic data types and logical entities. These include:

(a) the simulated clock - each event taking place during a SIMSCRIPT simulation is associated with a value of the special system variable TIME, which represents a simulated internal clock. The value stored in the clock location is automatically updated by the system before each simulated event.

(b) entities - a SIMSCRIPT entity is a structured collection of named data fields, forming a contiguous block of words beginning at a base address; this collection of words may be called the entity descriptor, while the individual fields in the block are called the entity attributes. Entities are divided into three sub-types: temporary entities, permanent entities, and event notice entities. The manner in which these three types of entities are distinguished, and the slight variations in the manner of their use, will be described in detail later. Event notice entities are treated in much the same way as temporary entities; each such entity is referenced via the address of its descriptor block. Permanent entities are handled more like elements of FORTRAN arrays, and are referenced by their name and number.

(c) The SIMSCRIPT system incorporates sets, actually linearly structured lists, of three kinds. The three types of sets provided are distinguished by their use and structure, as follows:

(i) Stacks or LIFO lists. Such a set is maintained as a unilaterally linked list, and the system maintains a reference to the first element of the list only.

(ii) Queues or FIFO lists. SIMSCRIPT queues are maintained as unilaterally linked lists; the system maintains a reference

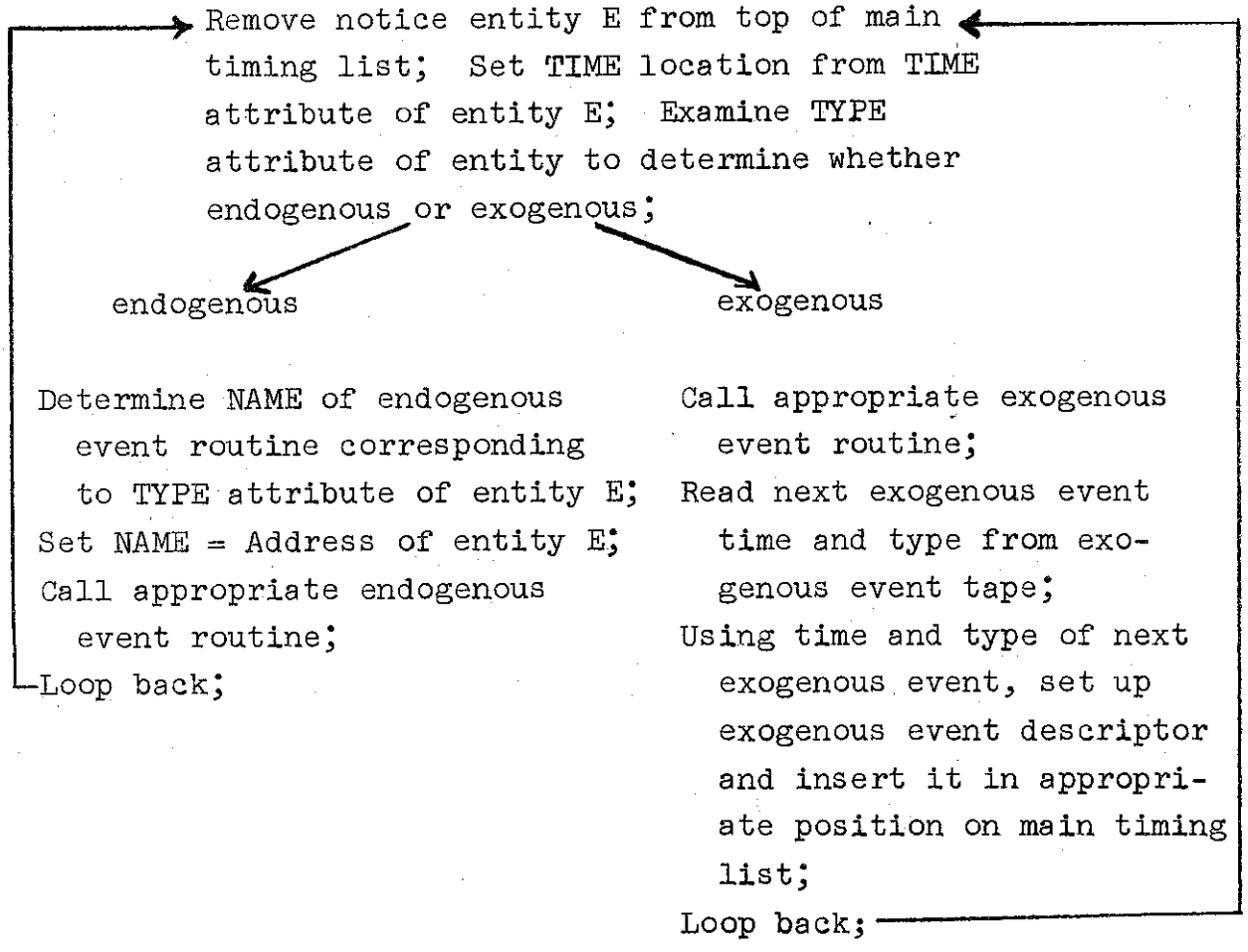
to the first and to the last element of every queue.
(iii) Ranked lists. Ranked lists are maintained as bilaterally linked lists; the system maintains a reference both to the first and to the last element of every ranked list.

Elements are entered into stacks at their top, and also removed from the top of stacks. Elements are entered into queues at their bottom, but removed from the top of queues. Entities are entered into ranked lists at a position determined by the numerical value of a ranking attribute, in a manner described in more detail below. The SIMSCRIPT system provides for the removal of elements from ranked lists either at the top of the list or at any intermediate position in the list; it is for this reason that ranked lists use bilateral linkages.
(d) The SIMSCRIPT system maintains one special ranked list for system use: the main timing list or endogenous event list, which is an ordered list of event notice entities. Each of these entities describes a future event, to occur at a definite time. The time at which an event is to occur is determined by its TIME attribute; this attribute is maintained by the system as one of a number of special system attributes pertaining to event notice entities and recorded in the first two words of the descriptor of such an entity (which words are consequently unavailable for programmer use). Except for the special reservation of the first two words of its descriptor block by the SIMSCRIPT system, an event notice entity has the same structure and usage as does an ordinary entity.

When, in the course of a simulation run, control returns to the main scheduling routine of the SIMSCRIPT system, this routine examines the main timing list to determine the next event to be simulated. A notice entity describing this event will always be found at the top of the main timing list. This entity may either describe an endogenous or exogenous event; in which case its data subfields will contain all

necessary parameters of the event, or may describe an exogenous event, in which case it will merely give the time and name of the exogenous event. In this case, parameters of exogenous events are read subsequently from an external exogenous event tape. During its operation, the master scheduling routine uses the TIME attribute of the event notice entity which it finds at the top of the main timing list to update the simulated clock location TIME, and then transfers control to whichever event routine corresponds to simulated event. (Cf. below for additional details concerning these routines.) Table I below describes the structure of the SIMSCRIPT main scheduling routine in more detail.

Table I. Structure of the SIMSCRIPT main scheduling routine.



(e) The SIMSCRIPT system allows use of an exogenous event tape. This tape contains a sequence of records, each such record describing a single exogenous event with all its parameters. The information on this tape is maintained in standard FORTRAN card image form, and is read both by the SIMSCRIPT system routines and by simulation-programmer written routines with the use of normal FORTRAN conventions involving READ statements with FORMATs etc. The first card of the record describing a particular event must contain the time of the simulated event and an integer describing the type of the simulated event, both in standard format. This card is read by the SIMSCRIPT main scheduling routine. The remaining cards are to be read by user-written routines. The association between the exogenous event numbers found on the exogenous event tape and the particular exogenous events to which they refer is established by a specific declaration of a form described somewhat below.

(f) Events, represented within the SIMSCRIPT system by event notice entities, are simulated by a collection of endogenous and exogenous event routines and their subroutines. One event routine must be provided for each type of endogenous and exogenous event which is allowed to occur during a given simulation run. When the main scheduling routine decides that a particular event is to be simulated, it transfers control to the associated event routine. Event routines are notified, on being called, of the location of the notice entity describing the event which they are to simulate; by reading appropriate fields of this notice entity, they are able to determine all the parameters of the event. (Cf. Table I above.) Exogenous event routines find the necessary parameters of the exogenous event which they are to simulate by reading the exogenous event tape.

(g) The SIMSCRIPT system is based upon FORTRAN. Any legal FORTRAN statement may occur in a SIMSCRIPT program; the SIMSCRIPT statements themselves have a distinguishing syntax which enables the SIMSCRIPT compiler to recognize them separately from FORTRAN statements among which they may be embedded. The subroutine description-and-call-conventions in SIMSCRIPT are identical with those in FORTRAN. In particular, any legal FORTRAN library subroutine may be called for use within a SIMSCRIPT program.

(h) The SIMSCRIPT language allows the use of local and global variable names for the storage of a single machine word of information; these variable names have much the same use and are subject to much the same conventions as in FORTRAN. A local variable is used as if it were a FORTRAN variable local to the particular subroutine in which it is mentioned; a global variable is accessible from all parts of a SIMSCRIPT program, and has much the same usage as a FORTRAN Common variable.

The fields of a (temporary) SIMSCRIPT entity are referenced by using a variable (or more generally expression) as an index, and prefixing it with the name of the field in question. Thus, e.g., if I is a local variable containing the address of an entity E of a type for which Weight and Volume are declared attributes, then WEIGHT(I) and VOLUME(I) are expressions whose values are determined by contents of the appropriate data fields of E. An attribute of one entity may be a reference to another entity, and, accordingly, nested expressions referencing some ultimate data field are allowed as in COST(HOUSE(OWNER(DØG))). In the above example, the variable DOG must contain a valid entity address.

Permanent SIMSCRIPT entities are handled in a slightly different way. When a SIMSCRIPT permanent entity type is declared (see below for details) the block size of the permanent entity's descriptor block, together with the manner

in which this block is structured into data fields, is stated. At the same time the number of permanent entities of each type is stated (as in FORTRAN Dimensioning). Storage for permanent entities is allocated statically at compile time (whereas storage for temporary entities is handled dynamically). Thus permanent entities, as well as the data fields which they contain, are referenced by numerical indices, and treated in much the same way as FORTRAN arrays would be treated (except that in the case of permanent entity attributes these arrays are arrays of part words whose members are regularly but not necessarily contiguously arranged in memory). A permanent SIMSCRIPT entity may be declared to have one or two indices; a doubly indexed permanent entity is rather similar to a FORTRAN array of two dimensions.

The general SIMSCRIPT construct used for reference to a data field is called a field expression. The syntactic definition of this notion is as follows:

```
<entity locator>  =  <field expression>
<integer quantity>=  <integer algebraic expression>
<algebraic expression>  =  <term><addop><term>...<addop><term>
                        |-<term><addop><term>...<addop><term>|<term>
<addop>  =  +|-
<term>   =  <factor> * <factor> * ... * <factor> | <factor>
<factor> =  <element> / ... / <element> | <element>
<element>=  <field expression>|(<algebraic expression>)
                        |<function call>
<field expression>  =  <simple variable name>
          |<temporary attribute name>(<entity locator>)
          |<permanent single-indexed attribute name>
                        (<integer quantity>)
          |<permanent double-indexed attribute name>
                        (<integer quantity>, <integer quantity>)
<integer algebraic expression> = <algebraic expression>
<real algebraic expression>    = <algebraic expression>      .
```

Function calls, which may occur in algebraic expressions, have the following syntax.

<function call> = <function name>(<algebraic expression>,
..., <algebraic expression>) ,

resembling that used in FORTRAN rather closely. Real expressions are distinguished from integer expressions by a lexical first-letter-of-name convention identical to that used in FORTRAN.

The most novel features of SIMSCRIPT come from the interaction between entities and sets; we now turn to describe the logical relations involved. Each type of set to be used in a given simulation must be declared, and its basic properties specified; whereas the actual declaration style used in SIMSCRIPT employs a fixed-format, tabular style of declaration, a logically equivalent linear declaration form might be described as follows:

<set declaration> = DECLARE SETS <set descriptor>
,..., <set descriptor>
<set descriptor> = <name> (<set type descriptor>)
<set type descriptor> = LIFO, <integer>| FIFO,<integer>
|<ranked set descriptor>
<ranked set descriptor> = RANKED,<integer>,<ranking
attribute name>, <order keyword>
<order keyword> = INCREASING | DECREASING .

In <set type descriptor> and <ranked set descriptor> above, the <integer> which occurs may be either 0, 1, or 2, and describes the number of indices required to identify a set of a particular type. A set with no subscripts will always be referenced through a global variable having the same name as the set and addressed by this name throughout the whole of a SIMSCRIPT code. Such a set is therefore used in "global" fashion. A set with a single subscript will always be referenced either through a field (LIFO case) or

through a pair of fields (FIFO and RANKED cases) belonging to some other entity as attributes. Such a set is therefore attached to some other entity (temporary or permanent); this entity is the owner of the singly indexed set. A set with two subscripts will always be referenced through either a field (LIFO case) or through a pair of fields (FIFO or RANKED case) belonging to a permanent entity as attributes; note that, since attributes of permanent entities, like permanent entities themselves, are addressed by numerical subscripts, a doubly indexed set is addressed using a pair of integer subscripts. If a temporary or permanent entity type is ever to be the owner of an indexed set, the entity type must include among its declared attributes an attribute called FNAME, which must be assigned a field long enough to contain an address; this field is used by the system to reference the first element of the set NAME owned by the entity. (F is used conventionally for "first element.") In addition, if the set in question is either an FIFO queue or a ranked set, any entity owning the set must include among its attributes a second field called LNAME (last element reference). These attributes of entities owning sets should normally not be used explicitly by the SIMSCRIPT programmer, but only implicitly via the entity-in-set operations which SIMSCRIPT provides.

The ranking attribute declared for a ranked set is used in a manner to be described in more detail below in order to determine the positioning of entities within a set. If a ranked set is declared as INCREASING, its elements will be arranged in increasing order of this attribute, and consequently elements with smaller values of the ranking attribute will be placed toward the beginning of the set. If the set is declared as DECREASING, its elements will be arranged in the opposite order.

A ranked set declaration must always specify the attribute according to which the elements of the set are to be ranked; every entity-type which in the course of a simulation may be assigned to the ranked set as a member must have an attribute of this name. When an entity of appropriate type is entered into a ranked set as a member (cf. "FILE" statement below), it will be placed in the position defined by its rank.

The specific syntactic form in which SIMSCRIPT makes reference to a set is defined as follows.

```
<set expression> = <unindexed set expression>|<singly
                   indexed set expression>|<doubly
                   indexed set expression>
<unindexed set expression>  =  <set name>
<singly indexed set expression> = <set name>(<entity locator>)
                   |<set name>(<integer quantity>)
<doubly indexed set expression> = <set name>(<integer quantity>,
                   <integer quantity>)
```

Each of the three types of entities provided by the SIMSCRIPT system, i.e., temporary entities, event notice entities, and permanent entities, must be declared together with its attributes. The actual form of declaration used in the implemented SIMSCRIPT system is fixed format and tabular; a logically equivalent linear declaration form is as follows.

```
<temporary entity declaration> = DECLARE ENTITIES <temporary
              entity descriptor>,...,<temporary entity descriptor>
<event notice entity declaration> = DECLARE EVENT ENTITIES
              <event entity descriptor>,...,<event
              entity descriptor>
<permanent entity declaration> = DECLARE PERMANENT ENTITIES
              <permanent entity descriptor>, ...,
              <permanent entity descriptor>
```

[continued]

<temporary entity descriptor> = <name>(<record size>,
<attribute descriptor list>)|<name>(<record
size>,<satellite pair list>,<attribute
descriptor list>)
<record size> = 1|2|4|8
<satellite pair list> = <satellite pair>,...,<satellite pair>
<satellite pair> = (<satellite block number>,<record size>)
<satellite block number> = 1|2|3|4|5|6|7|8
<attribute descriptor list> = <attribute descriptor>,...,
<attribute descriptor>

If an entity type is ever to be assigned as a member of a unilaterally linked set (i.e., a FIFO or LIFO queue) called NAME, it must include among its declared attributes an attribute called SNAME, which must be assigned a field long enough to contain an address; this field is used by the system in order to contain the successor links structuring the set NAME (S is used conventionally for "successor link"). If an entity is ever to be assigned as a member of a bilaterally linked set called NAME (i.e., a ranked set) it must include among its attributes a pair of fields, one called SNAME and the other called PNAME ("predecessor name"). These attributes of entities belonging to sets should normally not be used explicitly by the SIMSCRIPT programmer, but only implicitly via the entity-in-set operations which the SIMSCRIPT system provides.

The basic SIMSCRIPT commands relating entities and sets are as follows: A command

CREATE <entity type name> NAMED <field expression>

is provided. This statement calls the dynamic space-allocator which is built into the SIMSCRIPT system to allocate one or several blocks of storage for an entity of the type named. The address of the newly allocated entity is placed in the data field addressed by the field expression occurring

as the last part of the statement. Another action, which may be explained as follows, takes place at the same time. The SIMSCRIPT system associates, with each declared temporary entity type, a simple variable of the same name accessible from all parts of the SIMSCRIPT program within which this entity type is declared. Thus, for example, if an entity type named XXX has been declared, a cell also named XXX is set aside. Whenever the CREATE statement is invoked to create an entity of type XXX, the address of the newly created entity is placed in the cell named XXX. This circumstance allows use of the CREATE statement (as well as a number of other SIMSCRIPT statements to be described below) in the short form

CREATE <entity type name> .

This statement calls the SIMSCRIPT space allocator to allocate one or several blocks of storage for an entity of the type named, and places the address of the newly created entity in the variable having the same name as the entity type, as just explained. Since no field expression occurs in the short form of the CREATE statement, no attempt is made to place a reference to the newly created entity in any other place. That is, CREATE XXX is logically equivalent to CREATE XXX CALLED XXX. A similar remark applies to the short form of various other of the statements to be described in the next few paragraphs.

The operation inverse to the SIMSCRIPT CREATE operation is the DESTROY operation, which can be called either in long or in short form, these two forms being as follows:

DESTROY<entity type name> NAMED<field expression>
DESTROY <entity type name> .

Invocation of either of these statements returns the space occupied by the referenced entity for reuse by the SIMSCRIPT space allocator. If the DESTROY statement occurs in its

long form, the address of the entity to be destroyed is taken from the field expression occurring at the end of the statement; if the DESTROY statement occurs in its short form, the address of the entity to be destroyed is taken from the variable having the same name as the entity type, as explained above. It is the SIMSCRIPT programmer's responsibility to be sure that he never DESTROYs an entity to which subsequent reference is directly or indirectly made in the course of a simulation run. If such an error occurs in a program, invalid information may subsequently be supplied to routines referencing an entity after the space which it occupies has been released for reallocation; this may lead to various kinds of subsequent program failures, including transfers to irrelevant locations. In particular, an entity should never be destroyed while it is still a member of any set.

Entities are made members of sets by the SIMSCRIPT statement:

FILE <field expression> IN <set expression> .

Invocation of this command inserts the entity whose address is contained in the data field referenced by the <field expression> forming the first part of the statement into the set addressed by the <set expression> forming the second part of the statement. The entity in question is inserted in proper position into the set, all necessary pointers and references being automatically updated. Insertion will be at the top of a stack, at the bottom of a queue, and in rank order in a ranked set.

An action inverse to that of the FILE statement is provided by the following statement:

REMOVE FIRST <field expression> FROM <set expression> .

Invocation of this statement causes the first element of the set addressed by the <set expression> forming the latter

part of the statement to be removed from the set, and causes the address of this entity to be inserted into the <field expression> forming the first part of the statement. This allows the newly removed entity to be addressed for other purposes. For ranked sets, and for ranked sets alone, another related command having the following form is available:

REMOVE <field expression> FROM <set expression> .

This statement removes the specific entity addressed by the <field expression> forming its first part from the set addressed by the <set expression> forming its second part. Since this type of element removal is allowed to affect not only the first but an arbitrary element of the set to which it applies, its use is restricted to bilaterally linked, i.e., ranked, sets.

We have already noted that the SIMSCRIPT system maintains a master timing list of event notice entities. Special forms of the above commands are provided for entering event notice entities into, and deleting them from, this special list. These commands are as follows. The command,

CAUSE event type name> CALLED <field expression>
AT <real algebraic expression>

and its associated short form,

CAUSE <event type name> AT <real algebraic expression>

enters event notices into the main timing list. More precisely, the first of these statements, when invoked, adds an event notice entity of the type specified in the statement, and which is referenced by the contents of the data field located by the <field expression> which this statement contains, to the master timing list. The TIME attribute of the event notice entity is set equal to the value of the <real algebraic expression> forming the last part of the statement. If the statement is invoked in

short form it operates in much the same way, except that the notice entity added to the main timing list is located via the global variable having the same name as the entity type being added to the list. Of course, when an event notice entity is added to the main timing list it is placed in the position determined by its TIME attribute, earliest events being placed first.

The corresponding inverse command has the long and short forms

CANCEL <event type name> CALLED <field expression> ;

or

CANCEL <event type name> .

The first of these statements uses the field expression which it contains to locate an event notice entity of the specified type; this event notice entity is then removed from the bilaterally linked, ranked main timing list. The short form statement operates in much the same way, except that it uses the global variable having the same name as the type of the event entity to be removed to locate the entity.

In addition to the statements described above, which control the creation and destruction of entities, the insertion and removal of entities into and from sets, and the insertion into and removal of notice entities from the main timing list, SIMSCRIPT contains a variety of other statements which may be invoked to obtain effects more nearly resembling those familiar from experience with more common algebraic languages. Many of the statements in this latter group may be supplied with a postfixed repetition control list if iterative execution of a single statement is desired. Thus, for example, the SIMSCRIPT LET statement, whose significance is much like that of the FORTRAN assignment statement, except that it allows assignment not only to a full word but to a subfield of a word, has the following syntax.

```
LET <field expression> = <algebraic expression>
    |LET<field expression> = <algebraic expression>,
                             <repetition control>  .
```

The simple form of this statement sets the contents of the field addressed by its first part equal to the value of the <algebraic expression> forming its second part. The compound form of the LET statement performs precisely the same action, but performs it repetitively under the control of the <repetition control> element which forms the final part of a compound LET statement. The syntax of a repetition control element is as follows.

```
<repetition control> = <repetition phrase>,...,<repetition phrase>
<repetition phrase>  = <forpart>|<forpart>,<withpart>
<forpart>  =  FOR <variable name> = (<integer quantity>)
                                    (<integer quantity>)
 |FOR<variable name>  = (<integer quantity>)(<integer quantity>)
                                    (<integer quantity>)
 |<forwords> <single-subscripted permanent entity name>
                                    <variable name>
 |<forwords> <variable name> <ofwords> <set expression>
<forwords>  =  FOR EACH | FOR ALL | FOR EVERY
<ofwords>   =  OF | IN| ON | AT
<withpart> =   WITH <boolean expression>
<boolean expression>  = <boolean term>, OR <boolean term>,...,
                                    OR <boolean term> | <boolean term>
<boolean term>  =  <boolean factor>, AND <boolean factor>,...,
                                    AND <boolean factor> | <boolean factor>
<boolean factor> = <relation> | NOT < relation>
<relation>  = (<algebraic expression>) <relation operator>
              (<algebraic expression>)|(<boolean expression>)
<relation operator> = GR|GE|EQ|NE|LS|LE
```

A <repetition control> consists of a sequence of <repetition phrases> separated by commas. Each <repetition phrase> causes

a single "level" of repetition; thus, the occurrence of a whole sequence of repetition phrases causes a nested, iterated repetition, working out from the first repetition phrase to the last repetition phrase, as in a nested set of FORTRAN DO-loops. Each repetition phrase either consists simply of a FOR statement in one of four admissible forms, or is a compound consisting of a FOR statement to which a boolean condition has been attached. The syntax of the <forpart> constituting the first piece of every <repetition phrase> is given in the table above. Examples of the four options provided are as follows:

```
FOR I = (1)(N)
FOR I = (1)(N)(2)
FOR EACH PERMTHING I
FOR EACH I OF GROUP(J)  .
```

Each of these examples implies an iterative execution of whatever statement is a controlled statement by the FOR-statement in question, during which iteration the variable I is systematically varied. The first two forms shown above resemble the two forms of the FORTRAN DO statement, in that they vary I in arithmetic sequence from a lower to an upper limit; the second of these forms implies an increment different from 1. The third example shown above, in which the variable name occurring is required to be a zero-subscript permanent entry name as shown in the preceding syntax table, varies I by steps of one from an initial value of one to a final value equal to the total number of entities of type PERMTHING declared. The final line in the above group of four examples shows iteration over the members of a set; in this example, GROUP(J) is required to be a set expression, and I is successively set equal to each of the members of this set, in order, during the implied iteration. The iteration implied by a SIMSCRIPT FOR-clause can be conditioned if a WITH-clause is attached to the FOR-clause. As noted in the above syntactic

table, such a WITHPART consists of the key word WITH followed by a boolean expression built up from elementary comparisons using the bollean operators AND, OR, and NOT. Thus, for example, we may write

```
...,FOR EACH I OF GROUP(J), WITH(SIZE(I))GR(J),OR(2*SIZE(I))
    LT(J) ...
```

as a typical compound repetition phrase. The occurrence of such a repetition phrase will cause iterative execution of the statement which it controls, the variable of iteration running successively over the domain described by the <forpart> of the repetition control phrase, but all values of this variable not satisfying the appended WITH-phrase being omitted.

The remaining SIMSCRIPT statement forms are as follows. A STORE statement is provided, and has the form

```
STORE <algebraic expression> IN <field expression> |
    STORE <algebraic expression> IN <field expression>,
    <repetition control>  .
```

This command stores the value of the algebraic expression forming its first part into the field expression following the keyword IN; the presence of a repetition control part causes iterated execution of the basic statement in the manner already described. This statement has much the same semantic significance as the LET statement, except that, whereas the LET statement may imply conversion between real and integer values as determined by the declared or implicit quantity types in the two parts of a LET statement, no such conversion is implied by a STORE statement.

SIMSCRIPT incorporates all legal FORTRAN statements, and, in particular, incorporates the various forms of flow control statements provided by FORTRAN, i.e., the two-way IF, the three-way IF, the GO TO command, and the computed GO TO statement. It also incorporates the FORTRAN iterative DO

statement. In addition, a somewhat more flexible iteration control statement is provided, having the following syntax.

DO TO <statement number>, <repetition control> .

The scope of such a DO TO statement includes all statements which lie between the statement number which it contains and the next following SIMSCRIPT statement labeled with this statement number; as many repetitions of all these statements are executed as are specified by the repetition control part of the DO TO statement.

The following two statement forms are provided for testing the condition of a set.

IF <set expression> IS EMPTY <any SIMSCRIPT statement>

or

IF <set expression> IS NOT EMPTY <any SIMSCRIPT statement> .

The semantics of these statements is obvious.

To facilitate finding either the maximum or the minimum of a collection of numbers, SIMSCRIPT provides the two following statements.

```
FIND <field expression> = MAX OF <algebraic expression>,
                          <extended repetition control>, IF NONE
                          <any SIMSCRIPT statement>
FIND <field expression> = MIN OF <algebraic expression>,
                          <extended repetition control>,
                          IF NONE <any SIMSCRIPT statement> .
```

The syntax of the extended repetition control part forming the last section of a FIND MAX or FIND MIN statement is as follows.

```
<extended repetition control> = <extended repetition phrase>,
        ..., <extended repetition phrase>
<extended repetition phrase>  = <repetition phrase>
            <repetition phrase> <where clause>
<where clause>  =  WHERE <field expression> <comment>  .
```

Each <extended repetition control> is a list of <extended repetition phrases>; each such phrase causes a single level of repetition, and all together cause a nest of iterated repetitions working out from the first extended repetition phrase to the last extended repetition phrase. Each <extended repetition phrase> is either a simple <repetition phrase> of the type with which we have already become familiar, or is a compound consisting of a simple <repetition phrase> to which a <where clause> is appended. A <where clause> consists of the keyword WHERE and a <field expression>, to which an arbitrary sequence of SIMSCRIPT words may be appended. The occurrence of a WHERE clause in an extended repetition phrase within a FIND MIN or FIND MAX statement causes that value of an iteration variable belonging to the maximum or minimum eventually found to be saved, and to be inserted into the field expression occurring within the WHERE clause. This allows for the determination not only of the maximum of an algebraic expression, but simultaneously for a determination of the parameters for which this algebraic expression reaches its maximum. Thus, for example, if we write

```
FIND X = MAX OF CØS (Y1(I) + Y2(J)), FOR
    I = (1)(100), WITH (K(I))GT(MIN), WHERE IBIG
    IS MAXIMIZING I, FOR ALL J IN GROUP,
    WITH (TYPE(J))EQ(NEEDED), WHERE JBIG IS
    MAXIMIZING J, IF NONE GO TO 100
```

The indicated maximum will be calculated and the values of the parameters I and J for which this maximum is attained will be placed into the field, IBIG and JBIG. The comment provided as part of the WHERE clause has no program significance, but is only intended to improve program legibility. Every FIND MAX or FIND MIN statement occurring in a SIMSCRIPT program ends with the two key words IF NONE followed by some arbitrary SIMSCRIPT statement; if the iterative search implied by the FIND MAX or FIND MIN statement discovers no elements

satisfying the boolean conditions contained in the extended repetition control part of the statement, then the terminal <statement> which forms a part of the FIND MAX or FIND MIN command will be executed.

A modified form of the above statement is provided in the SIMSCRIPT language in order to make the iterative searching action of the FIND statement available in situations in which an iterative search is to be performed but in which no algebraic expression is to be minimized or maximized. This statement has the form

FIND FIRST <extended repetition control>,
IF NONE <any SIMSCRIPT statement> .

This statement searches iteratively over all the elements implied by the <extended repetition control> portion which it contains; the first time all the boolean conditions contained within this <extended repetition control> are satisfied, the iteration is broken off. At this point, the value of any iteration parameters mentioned in WHERE clauses will have been made available. If the iterative search implied by a FIND FIRST statement is unable to satisfy the set of boolean conditions which it contains, so that no element within the scope of the iterative search is actually found, the IF NONE clause which constitutes the final portion of a FIND FIRST statement is executed.

The SIMSCRIPT system incorporates a small statistical package which is accessed through a COMPUTE statement having the following form.

COMPUTE <field expression list> = <statistical quantity list>
OF <real algebraic expression>, <repetition control> .

The syntax of the various elements comprising a COMPUTE statement are as follows.

<field expression list> = <field expression>,...,
<field expression>

<statistical quantity list> = <statistical keyword>,...,
<statistical keyword>

<statistical keyword> = NUMBER|SUM|MEAN|SUM-SQUARES|
|MEAN-SQUARE|VARIANCE|STD-DEV

The first element of a COMPUTE statement is a list of <field expression>s, and the second element is an equally long list of <statistical key word>s. The two final portions of a COMPUTE statement are a <real algebraic expression> and a <repetition control> determining a set of values over which this <real algebraic expression> is to be calculated. By repeated calculation of a real algebraic expression, a finite collection of real numbers is determined; the statistical quantities listed in the COMPUTE statement are then calculated for this collection of real numbers, and the value of each statistical quantity inserted in the corresponding field as determined by the <field expression list> which the COMPUTE statement contains. The statistical quantities provided within the system are a number of basic linear and quadratic functions of a data-collection; the mean, variance, standard deviation, etc.

A statement convenient for the expression of groups of addition operations of a form common in simulation programs is provided by ACCUMULATE, whose syntax is as follows.

ACCUMULATE<field expression list> INTO<field expression list>
SINCE <algebraic expression list>
|ACCUMULATE<field expression list> INTO<field expression list>
SINCE <algebraic expression list>, <post-add list>
|ACCUMULATE<field expression list> INTO<field expression list>
ALL SINCE <real algebraic expression>
|ACCUMULATE<field expression list> INTO<field expression list>
ALL SINCE <real algebraic expression>, <post-add list>

The syntactic definitions of various of the sub-elements occurring in an ACCUMULATE statement are as follows.

<algebraic expression list> = <real algebraic expression>,...,
<real algebraic expression>

<post-add list> = <post-add clause>,...,<post-add clause>

<post-add clause> = POST <real algebraic expression>
| ADD <real algebraic expression> .

In its simplest form, the ACCUMULATE statement consists of three parts, all of which are lists required to be of equal length. The elements of the first and second lists are <field expression>s, the elements of the third list are <algebraic expression>s. Each <algebraic expression> is added to the value held in the field addressed by the corresponding <field expression> in the first list contained in an ACCUMULATE statement, and the results of this addition inserted into the field addressed by the corresponding <field expression> in the second list of the ACCUMULATE statement. If the key phrase ALL SINCE occurs in place of the phrase SINCE, then the ACCUMULATE statement contains a single <algebraic expression> rather than a list of algebraic expressions; in this case, the value of the <algebraic expression> occurring is added to the value of every field addressed by an element of the first <field expression> list contained in the ACCUMULATE statement, and the resulting value entered into the field addressed by the corresponding <field expression> in the second list. An ACCUMULATE statement may also be compound, and include a list of <post-add clauses> as its final part. If n such clauses occur, their effect is as follows. Each clause contains a real algebraic expression. If the n-th of these clauses contains the key word POST, the value of the real algebraic expression contained in the clause is calculated and inserted into the field addressed by the n-th member of

the first <field expression> list contained in the ACCUMULATE statement. If the n-th post-add clause contains the keyword ADD together with a real algebraic expression, the value of the real algebraic expression is calculated and added to the value contained in the n-th field addressed by the n-th element of the first <field expression> list contained in the ACCUMULATE statement; the sum is then inserted into this field.

As already noted, prime responsibility for the simulation of each exogenous or endogenous event belongs to a uniquely determined exogenous or endogenous event subroutine; it is the responsibility of the programmer to supply such an event routine for every event type which is allowed to occur during a given simulation run. The header card of an endogenous event routine has the form

ENDOGENOUS EVENT <name> ;

the header card of an exogenous event routine has the corresponding form

EXOGENOUS EVENT <name> .

Note that these routines are subroutines called by the SIMSCRIPT master timing routine; they have however no explicit parameters, since while an endogenous event routine finds its parameters in the various fields of a corresponding event notice entity whose address is supplied to it by the master timing routine, an exogenous event routine always reads any necessary parameters from the exogenous event tape. Exogenous event entities are set up using information contained on the exogenous event tape; the particular exogenous event which a given record on the exogenous event tape represents is always signaled to the main timing routine by a numerical parameter read from the exogenous event tape, as explained earlier in the present section. The correspondence between exogenous event subroutine names and the integer parameters found on the exogenous event tape is

established by an exogenous event declaration, whose syntax is as follows.

DECLARE EXOGENOUS EVENTS <expair>,..., <expair> ,
where <expair> = (<name>,<integer>)

This declaration contains a list of pairs, the first member of each pair being the name of some exogenous event, and the second member of the pair being the integer serial number assigned to this exogenous event in its representation on the external tape.

Input and output are provided in the SIMSCRIPT system through the FORTRAN READ, WRITE, and FORMAT statements which it contains; a special convention, according to which a READ statement with no tape number refers implicitly to the exogenous event tape, is employed. If an exogenous event routine is to read data from the first card of an exogenous event record, i.e., from the card containing the serial number and time of the exogenous event, which, as explained previously, is read also by the master timing routine, a special statement

SAVE

is provided. The effect of this statement is to readjust the pointers in the input-output buffer so as to make this first card available for re-reading by an exogenous event routine.

The original SIMSCRIPT system also incorporates a "report generator" feature allowing the setup of output in elegant tabular form and using a tabular format description. For simplicity, we shall omit this feature from the version of SIMSCRIPT described here.

4. The list processing language L^6.

L^6 is an elegant small list processing language developed by K. C. Knowlton of Bell Telephone Laboratories (cf. Knowlton, A Programmer's Description of L^6, CACM 9 (1966) 616-623). The basic data item in L^6 is the data block, consisting of a set of 2^n machine words, where n = 0,7. Thus, a data block consists of either 1,2,4,...,128 words. (Since the concept of "machine word" enters at so fundamental a level in its definition, the L^6 language is not machine independent.) A data block in L^6 may be divided into subfields of arbitrary size, except that each subfield must be a portion of a machine word. A subfield is defined by specifying a particular word in a block, and, within this word, the left-most and the right-most bit of the subfield. Each single character and single digit is treated by L^6 as the name of a field; thus, the L^6 processor is aware at any time of 35 different fields, one corresponding to each of the characters A,B,...,Z,0,...,9. That is, for each of these characters and digits a corresponding entry in a field definition table is maintained by the L^6 processor. The entries in this table have the form shown below.

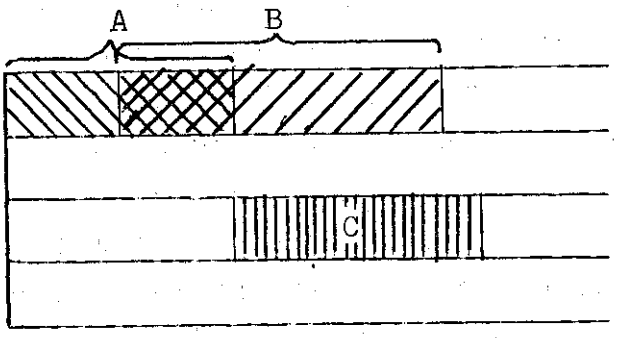
(1)

W	F	L	additional information

In (1), W references a certain word in a block; this word may be either the zero-th, first, ..., 127th word in a block. Thus the field W must contain 7 bits. The entry F in (1) references the first bit of a field; the entry L references the last bit of a field. The necessary size of each of F and L is of course determined by the size of the machine word in any given L^6 system. (The type of additional information which may usefully be contained in the extra field shown in (1) will be described below.) The entries constituting the field definition table are used to access the fields to which they refer. Each of the 35 field

definition table entries may be redefined dynamically during a run under the L^6 system. Thus, the field associated with every character may be redefined during the run. Fields are allowed to overlap. Suppose, for example, that the character A corresponds to the field in word zero of a block, bits 0-15; that the character B corresponds to the field in word zero of a block, bits 7-30; and that the character C defines a field in word three of a block, bits 16-32. Then the disposition of these fields in an L^6 data block consisting of 4 words appears as follows.

(2)



The contents of a field may be treated by the L^6 processor as alphanumeric information, as a non-negative integer, as a bit pattern to be manipulated by logical operations, or as a pointer to another L^6 block. The type of information contained in a field is defined implicitly by the way that an L^6 program uses it. Thus, there is no need to identify a field to the L^6 system as, for example, containing a pointer, containing an integer, etc.

Twenty-six logical base fields or bases are provided in the L^6 system. Each is associated with a given character A,B,...,Z. The instantaneous value of each base is determined by the contents of an associated machine word; the 26 machine words required to store the value of all the bases make up the base values table. The contents of a base are referenced in L^6 coding by writing the name of the base (consisting, of course, of a single character). More complex field reference is indicated by the concatenation of names. Thus, the

composite name AB1COD represents the contents of the field obtained as follows. First, get the contents of base A; treating this quantity as a pointer determining a certain block address, take the contents of the B field of this block. Treating this quantity, in turn, as a pointer determining a block address, take the contents of the 1-field of this block, ..., etc. Syntactically, a name or composite name in L^6 is any string of letters and digits beginning with a letter. Semantically, and in the sense explained above, any such name references the content of a certain field in some L^6 data block.

The field contents referenced by L^6 composite names are used in L^6 programming by tests and by operations. During the execution of an elementary L^6 operation, and depending on the particular operation in question, the field referenced by the operation may either be loaded only, be stored only, or be both loaded and stored. The L^6 language provides a variety of basic field comparison tests (equality, inequality, relative size, etc.) and basic field manipulation operations (copying, addition, subtraction, boolean operations, shifts, etc.). Operations and tests are assigned one or two character mnemonics and written in a primitive infix form. An exhaustive account of the L^6 operations and tests will be presented below. Here we mention only that, for example, (AX,E,BXY) is the form in which an L^6 test comparing the contents of the field AX and BXY for equality is written, while (AO,A,BXYZ) is the form in which we write an L^6 operation adding the contents of the field BXYZ to that of the field AO, and replacing the contents of the field AO by their sum.

The general L^6 statement is constructed of tests and operations of the above kind and has the following structure

(3) <label><condition-key><tests>THEN<operations><exit-label>

as in the following example:

(4) L2 IFNØNE(XD,E,Y)(XA,E,B) THEN (XD,A,C)(XA,A,D) L3

Various parts of the general L^6 statement may be omitted to form degenerate L^6 statements as explained in detail below.

The second argument of certain L^6 operations (specified in more detail below) can be a literal rather than a field name. Literals of the three types Hollerith, decimal, and octal are provided in the system. A Hollerith literal is any string of characters, digits, and periods. Operations and tests which permit a Hollerith second argument also require a Hollerith second argument, so that Hollerith literals are always distinguished from L^6 composite names by the context in which they occur. A decimal literal is a string of digits. An octal literal is a string of digits not including 8 or 9. The admissibility of a decimal or octal literal as the second argument of an L^6 argument is determined by the operation, these two possibilities excluding each other. Thus the interpretation of a numerical literal is always unique in context.

The code sequence generated by the occurrence in L^6 code of a composite name depends on whether the corresponding field is to be loaded or to be stored. The code sequence generated by the name ABC...DE in case of a load is shown in (5) below. In the code sequence (5) we assume the existence of two logical registers X1, X2 and three sets of additional logical registers Y00, Y01; Y1, Y11; Y20, Y21. The first of each of these pairs will be used to contain the location information for a given field; the second element of the pair will be used to contain the contents of that field. The particular pair to be used will be specified as a parameter in each call to the code generation routine which generates sequences of the form (5), and will be determined by knowledge of the particular (first, second, or third)

argument of the test or operation which is to be loaded or stored. (As indicated in more detail below, a few L^6 operations have three arguments.)

(5)
```
        X1 = A
        X2 = FD(B)
        call getfield
        X2 = FD(C)
        call getfield
        ...
        X2 = FD(D)
        call getfield
  ⟶    Y10 = X1  (or  Y20 = X1  or  Y30 = X1)
        X2 = FD(E)
        call getfield
        Y11 = X1  (or  Y21 = X1  or  Y31 = X1)
```

In (5) FD(J) is the J-th element of the field definition table described in the preceding paragraphs. The subouutine getfield fetches, into the logical register X1, the field contents specified by the block pointer contained in X1 at the time that getfield is called, and by the field definition table entry contained in X2 at the time that getfield is called.

When code for the field store operation corresponding to a given name is required, then the sequence (5) down to but not including the line in (5) marked with an arrow is generated, and placed exactly where the full code sequence (5) would otherwise be. Then, when the quantity to be stored has been calculated, the code sequence

(6)
```
        X1 = Y10  (or  X1 = Y20  or  X1 = Y30)
        X2 = FD(E)
        call setfield
```

is executed. By convention, all L^6 operations will leave their results

in a logical register known to the subroutine setfield; this routine finds a block pointer in X1 and the corresponding field definition table entry in X2. Using this information, it performs the required field store operation. The routines getfield and setfield may be coded very directly and simply; they require for their operation only the information contained in the fields W, F, L of (1). Alternately, and for the sake of efficiency, the addresses of efficient field extraction and insertion routines may be contained in the portion of the field definition item (1) marked "additional information". In this case, getfield and setfield may call directly into the load and store subroutines whose beginning addresses constitute this "additional information". If this procedure is followed, suitable field load and field insertion routines must be compiled dynamically by the L^6 processor every time a new subfield pattern is associated with one of the quantities A,...,9. Such dynamic compilation can produce highly efficient target code.

7.5. The APL or Iverson Language.

APL is an interesting and successful programming language developed at the IBM Corporation by Iverson and Falkoff. It is distinguished by the systematic and recursive use of arrays as basic data elements; these arrays may be of dimension 1, 2, 3, or more. APL is a basically interpretive system in which such fundamental attributes of data elements as their dimension are treated dynamically; this yields a system of great power which may however for certain problems be of low efficiency. As with LISP, the use of compound structures as basic data elements obviates the need to introduce functions or subroutines with large numbers of arguments, and in fact APL only provides functions with 1, 0, and 2 arguments. APL extends scalar operations in a systematic component-by-component manner to vectors, matrices, and higher dimensional arrays; vector operations are similarly and systematically extended to 2-dimensional matrices, etc. These implicit conventions give APL inherent power and conciseness. APL's conciseness is further enhanced by the systematic use of special symbols for built-in functions. A very remarkable economy of expression is ultimately attained, a fact especially welcome and impressive in connection with the use of APL from a console.

While it is our principal intent to discuss the APL language and the structure of its interpreter, it is worth mentioning some of the features of the supervisor under which APL runs which gives it much of its flavor. APL is available as an interpretive, time-sharing, remote terminal system. Supervisor commands are segregated from statements belonging to the APL language itself, so that the language maintains its simplicity unencumbered by system complications. The APL system may be considered to operate in response to invocation messages from the user; thus, a work session may be described as a series of invocation-response pairs. An

invocation will normally be an APL language expression; the response consists of the value of the APL expression. Alternatively, an invocation may be a stage in the definition of a function; in this case, the response consists merely in a request for the next line of the same definition. Finally, an invocation may be a supervisor command. Supervisor commands are distinguished from APL language commands by an initial right parenthesis followed by key words and identifiers.

The APL console executive associates a "library" with each system subscriber, and provides numbered "public libraries" in addition. Each library is divided into "workspaces," which may consist not only of symbolically stored user-defined functions but also of a complete set of tables describing the momentary status of an APL run. This allows APL sessions to be freely suspended and resumed. Within a given work space, defined functions are acessible through their symbolic names, in terms of which they may be copied from one work space to another. Defined functions and APL variables may also be united into symbolically named "groups," and entire groups may be copied using a single supervisor-level ACS command. At any time during execution of an APL program, control may be returned to the supervisor level by transmitting an appropriate break signal from the console; execution interrupted in this way may subsequently be resumed.

An APL console is always considered to be in one of two states, command state, in which statements are received for immediate execution, and function definition state, in which text is being received for subsequent execution as part of a function. The system is toggled from one state to another by the symbol ∇. When the console is in command state, entry of any valid APL expression causes the system to print out the value of that expression. Integer and/or real numbers are printed in a standard format; vectors

are printed by arranging their elements in standard sequence; two dimensional arrays are printed by arranging their elements in a standard two dimensional format; higher dimensional arrays are printed as a sequence of two dimensional arrays.

All simple APL statements are composed of data terms and function terms. The basic data forms are numeric constants (indistinguishably either fixed or floating point), boolean constants, and character constants. In output, the elements of boolean or numeric arrays are printed with separating blanks, the elements of character arrays without separating blanks. An APL name specifies either an array or a function name.

A large number of built-in system functions are provided; in addition, user defined functions are allowed. Functions are distinguished as monadic (one **argument**, always immediately to the left of the function term) dyadic (two arguments, one to the immediate right, the other to the left); ocassional no-argument functions are allowed. The simple function concept is applied within APL to a maximally wide variety of processes. Both to provide standardized treatment for a wide variety of cases and to provide simplified parsing, a strict right to left Polish convention is employed. Thus, for example, the APL expression

$$A \div B - FN\ C \times D \geq E$$

is equivalent to

$$A \div (B - FN\ (C \times (D \geq E))).$$

Wherever possible, function symbols do double duty as monadic and as dyadic operators. This is the case with - (minus) in many languages; in APL, this convention is systematically extended, so that for example, $\div P$ denotes the reciprocal of P, $\times T$ denotes the signum of T, etc. The following rule avoids ambiguity in this double use of function characters: If the lexical entity to the left of a function term in a statement is a datum, interpret the function

dyadically, otherwise interpret it monadically.

APL treats the assignment operator, designated by a leftward pointing arrow, as an ordinary binary operation having the value of its right hand side but having the "side effect" of giving its left hand side the attributes and value of its right hand side. This allows multiple assignment operations to occur as part of a composite APL expression; as usual for APL, operations are executed in a strict right to left fashion. Thus, for example, the expression

$$A \times A \leftarrow A \times A \leftarrow 3$$

is perfectly legal and has the value 81; it has also the side effect of assigning 9 as the final value of the variable A. If, however, the leftmost operation of an APL expression is an assignment operation, the value of the expression is not printed at the console when the expression is evaluated. Of course, in this case, the value may be obtained by separately entering the name of the variable occurring at the extreme left of the assignment expression. Thus, for example, if the expression

$$B \leftarrow A \times A \leftarrow A \times A \leftarrow 3$$

were entered at the console, no result would be typed. However, A would be assigned the value 9 as a side effect, and B the value 81. If the single symbol B were subsequently typed its value 81 would be printed.

The built in dyadic scalar functions provided in APL are addition, subtraction, multiplication, division, maximum, exponentiation, logarithm to a specified base, residue, binomial coefficient, a set of circular functions, the logical and, or, nand, nor operations, and the relational operators less than, not greater, equal, not less, greater, and unequal. Monadic operators are associated with many of these binary operators; this association is often defined very simply by substituting an appropriate "implied unit quantity"

in place of a missing left operand. This implied unit is 0 in case of addition and subraction, 1 in case of division, e in case of exponentiation, etc. In cases where this procedure is not appropriate, some other plausible significance is associated with the monadic form of a normally dyadic operator. Table 1 below, reproduced from the APL programmer's manual, gives detailed information on the built-in scalar operators provided in APL.

Monadic form fB		f	Dyadic form AfB	
Definition or example	Name		Name	Definition or example
$+B \leftrightarrow 0+B$	Plus	+	Plus	2+3.2 ↔ 5.2
$-B \leftrightarrow 0-B$	Negative	-	Minus	2-3.2 ↔ ¯1.2
$\times B \leftrightarrow (B>0)-(B<0)$	Signum	×	Times	2×3.2 ↔ 6.4
$\div B \leftrightarrow 1\div B$	Reciprocal	÷	Divide	2÷3.2 ↔ 0.625
B | $\lceil B$ | $\lfloor B$ 3.14 | 4 | 3 ¯3.14 | ¯3 | ¯4	Ceiling Floor	⌈ ⌊	Maximum Minimum	3⌈7 ↔ 7 3⌊7 ↔ 3
$\star B \leftrightarrow (2.71828..)\star B$	Exponential	⋆	Power	2⋆3 ↔ 8
$\circledast\star N \leftrightarrow N \leftrightarrow \star\circledast N$	Natural logarithm	⍟	Logarithm	$A\circledast B$ ↔ Log B base A $A\circledast B \leftrightarrow (\circledast B)\div\circledast A$
|¯3.14 ↔ 3.14	Magnitude	|	Residue	Case | $A\|B$ $A\neq 0$ | $B-(\|A)\times\lfloor B\div\|A$ $A=0, B\geq 0$ | B $A=0, B<0$ | Domain error
!0 ↔ 1 $!B \leftrightarrow B\times !B-1$ or $!B \leftrightarrow$ Gamma(B+1)	Factorial	!	Binomial coefficient	$A!B \leftrightarrow (!B)\div(!A)\times !B-A$ 2!5 ↔ 10 3!5 ↔ 10
$?B \leftrightarrow$ Random choice from ιB	Roll	?	Deal	A Mixed Function (See Table 3.8)
$\circ B \leftrightarrow B\times 3.14159...$	Pi times	○	Circular	See Table at left
~1 ↔ 0 ~0 ↔1	Not	~		
		∧	And	
		∨	Or	
		⍲	Nand	
		⍱	Nor	
		<	Less	
		≤	Not greater	
		=	Equal	
		≥	Not less	
		>	Greater	
		≠	Not Equal	

A	B	$A\wedge B$	$A\vee B$	$A\barwedge B$	$A\barvee B$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	0

Relations
Result is 1 if the relation holds, 0 if it does not:
3≤7 ↔ 1
7≤3 ↔ 0

$(-A)\circ B$	A	$A\circ B$
$(1-B\star 2)\star .5$	0	$(1-B\star 2)\star .5$
Arcsin B	1	Sine B
Arccos B	2	Cosine B
Arctan B	3	Tangent B
$(\bar{}1+B\star 2)\star .5$	4	$(1+B\star 2)\star .5$
Arcsinh B	5	Sinh B
Arccosh B	6	Cosh B
Arctanh B	7	Tanh B

Table of Dyadic ○ Functions

Table I: PRIMITIVE APL SCALAR FUNCTIONS

Any list of constants separated by spaces defines an APL vector. Thus, for example, if we write

A<--2 1 0

then when A is typed the value of A will be printed as

2 1 0.

Note that, typically for APL, the above assignment statement sets the value and the dimension of A simultaneously.

Character vectors are input by typing the individual characters which constitute their elements, in uninterrupted sequence and surrounded by quote marks. Thus, if we enter

A <-- 'ABCDEF'

A becomes a character vector, and if we then type A the response

ABCDEF

will appear.

Every scalar operator is at once generalized within the APL language to apply to arrays of arbitrary dimension and size on a component-by-component basis. This requires, however, either that the two operands of a dyadic operator be of the same size and dimension, or that one of the operands be either a simple constant or an array containing but a single element. Dyadic combination of a single element S' with an array combines S' separately with every element of the array. In terms of this convention and the standard APL conventions regarding assignments, we see that the input

A×A <-- 1 2 3 4

will produce the output

1 4 9 16;

the input

1 2 3 4 × 2

will produce the output

2 4 6 8 ;

while the input

1 2 3 4 × 5 6

will produce an error diagnostic.

Three other mechanisms by which scalar functions may be extended to arrays are provided. These are reduction, inner product, and outer product. If S is a built-in APL scalar function and V a vector of length m, then the formula f/V denotes the f-reduction of V, and has the value

V[1]fV[2]f...fV[n];

if n=1, then f/V has simply the value V[1]. If V is a two dimensional array, f/V is the vector U whose J^{th} element is

V[1;j]fV[2;j]f...fV[j;n],

and similarly for higher dimensional arrays. Reduction along other than the first dimension of a multi-dimensional array is specified by attaching an integer specifying the desired dimension of reduction enclosed in brackets as a so-called qualifier following the slash mark specifying reduction; thus, in the above circumstance f/[2]V is the vector whose j^{th} component is

V[j;1]fV[j;2]f...fV[j;n],

while f/[1]V has the same value as f/V.

The APL generalized inner product and other product provide two additional mechanisms for extending scalar functions to arrays. If f and g are two dyadic APL scalar functions, while A and B are arrays, then the outer product C←→ A∘.gB is the array whose typical element is

$$A(i_1,\ldots,i_m)gB(i_{m+1},\ldots,i_{m+n}).$$

If in this circumstance we assume that the last dimension of A is equal to the first dimension of B and let D be the array whose elements are $C(i_1,i_2,\ldots,i_m,i_m,i_{m+1},\ldots i_{m+n-1})$, the inner product Af.gB may be defined in terms of preceding concepts as f/[m]D. Note that A+.×B then denotes the ordinary matrix-matrix or matrix-vector product.

In addition to built-in scalar operations and their various extensionsto arrays, APL admits an interesting family of mixed operators, i.e., operators producing vectors from scalars, scalars from vectors, or more generally arrays of a certain dimension from arrays of a different dimension. One of the most basic of these is the indexing function. If C is a vector and I an integer constant, then C[I] denotes the I^{th} element of C. APL generalizes this familiar notion in various interesting ways. If I is a vector of integers rather than a single integer, the k^{th} element of I being i(k) while the i^{th} element of C is c(i), then C[I] denotes the vector whose components are c(i(k)). Similarly, if I is an n-dimensional array of integers whose elements are $i(k_1,\ldots k_n)$, C[I] denotes the array whose elements are $c(i(k_1,\ldots,k_n))$. If C is an m-dimensional array with elements $c(i_1,\ldots,i_m)$, and $I_1, I_2,\ldots,I_m$ are arrays with elements $i_j(k_1,\ldots,k_{\ell_j})$ then $C[I_1; I_2;\ldots;I_m]$ is the array whose elements are $c(i_1(k_1,\ldots,k_{\ell_1}),i_2(k_{\ell_1+1},\ldots,k_{\ell_1+\ell_2}),\ldots)$. Elision of a coordinate selects all indices along that coordinate; thus, for example, if A is a two dimensional array, A(I) is the vector whose J^{th} coordinate is A(I;J).

Besides its fundamental indexing operations, APL provides several additional built-in mixed functions. The monadic operator i converts an integer n into the vector consisting of the first n integers. Thus i5 has the value 1 2 3 4 5. The monadic operator ρ, applied to an array, gives the vector of dimensions of the array. The monadic ravel operator designated by a comma, makes any array A into the vector ,A of all the elements of A arranged in standard sequence.

The symbol ρ may also be used dyadically as the Vρa, where V is a vector of dimensions and A an arbitrary array; the value of VρA consists of the elements of A, taken in

natural order, but arranged as an array of the dimensions specified by V. If the elements of A do not suffice to fill out an array of the indicated dimension, they are cyclically repeated as often as necessary. Thus, for example, the value of

```
3 3⍴1 2 3 4 5 6 7
```

is

```
1 2 3
4 5 6
7 1 2.
```

The comma may also be used as a dyadic operator; as such, it denotes vector concatination. A dyadic index function ViS is provided, with the following significance. If V is a vector of n elements, then ViS yields the position of the earliest occurence of S in V, or n+1 if there is none such. If S is an array with elements S[I], then ViS is the array whose elements ViS[I]. The dyadic membership function A∈B gives 1 if the scalar A belongs to the array B, otherwise 0; if A is an array, it yields the obvious array of zeros and ones having the same dimension as A.

The take function V↑A is defined as follows. V is a vector of integers whose length m does not exceed the dimension of A. If the entries of V are positive, V↑A is the array whose entries are

$$A(i_1,i_2,\ldots,i_m,1,1,\ldots,1)\ ,\ 1\leq i_1\leq V(1),\ldots,1\leq i_m\leq V(m).$$

Any negative element V(i) selects the last |V(i)| rather than the first V(i) indices in the corresponding dimension. The drop function V↓A is defined analogously, except that an indicated number of elements are dropped rather than kept. If, for example, E is a 4 by 3 matrix, then 3 -2 ↑ E is the same as -1 1 E. The grade up and grade down functions, ⍋A and ⍒A respectively, produce the permutations which would order the vectur A in ascending (resp. descending) order. If A is an array of dimension higher than 1, they act on the first dimension of A unless written in qualifyed form as ⍋[N]A or ⍒[N]A, in which case they act on the N^{th} coordinate of A.

The compress function V/A is defined as follows. If V is a vector of length N, V/iN is the vector consisting of all integers J for which V[J] is non-zero; V/A is A[V/iN;]. The same "selection" action may be applied along the N^{th} dimension of A by writing V/[N]A. The related expand function V\A is defined by the condition that V/V\A is A, and that all the elements of V\A which do not correspond to elements of **A** are 0.

The following table, reproduced from the APL programmer's manual, summarises the APL operations. In the examples given in this table, the following arrays are used:

```
      P = 2  3  5  7

          1  2  3  4
      E = 5  6  7  8
          9 10 11 12

          A  B  C  D
      X = E  F  G  H
          I  J  K  L
```

Name	Sign[1]	Definition or example[2]
Size	ρA	ρP ↔ 4 ρE ↔ 3 4 ρ5 ↔ ⍳0
Reshape	VρA	Reshape A to dimension V 3 4ρ⍳12 ↔ E ⍳12ρE ↔ ⍳12 0ρE ↔ ⍳0
Ravel	,A	,A ↔ (×/ρA)ρA ,E ↔⍳12 ρ,5 ↔ 1
Catenate	V,V	P,⍳2 ↔ 2 3 5 7 1 2 'T',HIS' ↔ 'THIS'
Index[3,4]	V[A] M[A;A] A[A;...;A]	P[2] ↔3 P[4 3 2 1] ↔7 5 3 2 E[1 3;3 2 1] ↔ 3 2 1 11 10 9 E[1;] ↔ 1 2 3 4 ABCD E[;1] ↔ 1 5 9 'ABCDEFGHIJKL'[E] ↔ EFGH IJKL
Index generator[3]	⍳S	First S integers ⍳4 ↔ 1 2 3 4 ⍳0 ↔ an empty vector
Index of[3]	V⍳A	Least index of A in V, or 1+ρV P⍳3 ↔2 5 1 2 5 P⍳E ↔ 3 5 4 5 4 4⍳4 ↔ 1 5 5 5 5
Take	V↑A	Take or drop |V[I] first (V[I]≥0) or last (V[I]<0) elements of coordinate I 2 3↑X ↔ ABC EFG ¯2↑P ↔ 5 7
Drop	V↓A	
Grade up[3,5]	⍋A	The permutation which would order A (ascending or descending) ⍋3 5 3 2 ↔ 4 1 3 2
Grade down[3,5]	⍒A	⍒3 5 3 2 ↔ 2 1 3 4
Compress[5]	V/A	1 0 1 0/P ↔ 2 5 1 0 1 0/E ↔ 1 3 5 7 9 11 1 0 1/[1]E ↔ 1 2 3 4 ↔ 1 0 1⌿E 9 10 11 12
Expand[5]	V\A	A BCD 1 0 1\⍳2 ↔ 1 0 2 1 0 1 1 1\X ↔ E FGH I JKL
Reverse[5]	⌽A	DCBA IJKL ⌽X ↔ HGFE ⌽[1]X ↔ ⊖X ↔ EFGH LKJI ⌽P ↔ 7 5 3 2 ABCD
Rotate[5]	A⌽A	BCDA 3⌽P ↔ 7 2 3 5 ↔ ¯1⌽P ⍳1 0 ¯1⌽X ↔ EFGH LIJK
Transpose	V⍉A	AEI Coordinate I of A becomes coordinate V[I] of result 2 1⍉X ↔ BFJ CGK 1 1⍉E ↔ 1 6 11 DHL
	⍉A	Transpose last two coordinates ⍉E ↔ 2 1⍉E
Membership	A∈A	0 1 1 0 ρW∈Y ↔ ρW E∈P ↔ 1 0 1 0 P∈⍳4 ↔ 1 1 0 0 0 0 0 0
Decode	V⊥V	10⊥1 7 7 6 ↔ 1776 24 60 60⊥1 2 3 ↔ 3723
Encode	V⊤S	24 60 60⊤3723 ↔ 1 2 3 60 60⊤3723 ↔ 2 3
Deal[3]	S?S	W?Y ↔ Random deal of W elements from ⍳Y

Table II: PRIMITIVE APL MIXED FUNCTIONS

The binary deal function (whose name is taken from gambler's parlance) makes a stochastic element available within APL. It has the form M?N, and selects a vector of length M at random from iN, the vector being selected from iM without replacement. The monadic roll function ?N selects a random element from iN. The binary decode function V⊥U gives the value of the vector U in the number system whose radices are given by the components of the vector V; the encode function V⊤U gives the representation of a scalar U in the number system having the components of V as radices, and gives it to as many places as V has components. A binary rotate function K⌽X is provided, and is defined as follows. If K is a scalar or one element vector and X is a vector, then K⌽X is a cyclic rotation of X, rotating X K places to the left if K is positive and K places to the right if K is negative. If X is an m dimensional array, then the coordinate J along which rotation is to be performed may be specified by writing the qualifyed operator K⌽[J]X. Moreover, K may be an array whose dimensions must equal the dimensions of X less its J^{th} dimension; in this case, each vector along the J^{th} coordinate of X is rotated as specified by the corresponding element of K. K is also allowed to be a scalar, in which case every vector along the J^{th} coordinate of X is rotated K places. The monadic operator ⌽X corresponding to the dyadic rotation operator reverses the coordinates of a vector. It may be applied to an array of more dimensions by writing it in qualified form as ⌽[J]X, in which case X is reversed along its J^{th} coordinate. The generalized transpose operator V ⍉ X is defined as follows. V is a vector of length N equal to the number of dimensions of X; the coordinates of V constitute some permutation of the first m integers, with repetitions allowed. The value Y or V ⍉ X is the array of m dimensions defined by

$$Y(i_1,\ldots,i_m) = X(i_{q(1)},i_{q(2)},\ldots,i_{q(n)});$$

in this formula, each index i_j has the maximum range of

variation allowed by the expression on the right hand side of the formula. The corresponding monadic operator OX may be defined as

(1 2 . . . n-2 n n-1) OX,

where we assume that X is an array having n different dimensions; note that if X is an ordinary matrix this operation gives the transpose of X.

APL admits an interesting form of indexed assignment statement. The element immediately to the left of an assignment arrow may be an array A supplied with indices in square brackets; the pattern of indices which occurs must be such as to unambiguously designate some subarray B of A. The subarray B will have certain dimensions; the expression on the right of the assignment arrow must define either an array C of these same dimension or a scalar. This being the case, each element of the subarray B within A is replaced by the corresponding element of C (or by the value of C itself if C is a scalar).

APL makes use of a simple convention providing standard-format input and output. Numerical input required within an APL expression is indicated by using the "square box" symbol □ . On encountering this symbol, the APL interpreter will halt and require the input of a single numerical quantity from the keyboard. As soon as this is supplied, it will replace the occurrence of □ with the quantity supplied and the interpretation process will continue. If the symbol □ occurs to the left of an assignment arrow, it designates output. To request symbolic or alphabetic input, the symbol ⍞ is used instead of □ .

This completes our account of the APL interpreter in its execution mode. As has already been mentioned, APL admits input in a second mode, the so-called subroutine-definition mode. When in this mode, text is received for subsequent execution as part of a user defined subroutine. A definition is opened by typing the symbol ∇ . On comple-

tion of a subroutine defintion, the definition is closed by typing once more. The symbol beginning a new subroutine definition is followed immediately by a subroutine definition header. The header can have one of a number of forms. At its simplest, it consists solely of a name α, which subsequent to the completion of the subroutine definition body which follows the name becomes the name of the newly defined subroutine. If α is followed in the function header by a second name β, the subroutine is monadic (having a single argument, the argument being represented by β), and on each subsequent invocation of the subroutine, an argument is required. If three names occurred , α, β in sequence, the subroutine is recognized as having two arguments; and β are the argument names, and the middle name α is the subroutine name. If the header occurs in the form Z←A F B (this form may also occur without one of the arguments A or B), the subroutine is designated as a function; in this case, Z is the output value (possibly array) for the function. Variables occurring within a function definition will normally refer to the same logical object as identical symbolic names occuring elsewhere, which is to say that all variable names are ordinarily global. This is sometimes inconvenient and a mechanism is provided to designate names occurring within a function as being local to the function. To so name, one appends the name preceeded by a semicolon to the basic function header.

Successive statements in an APL subroutine are numbered and referred to by their nkmbers. APL statements in a subroutine may be the targets of transfers. A transfer operation in APL is indicated by an arrow (→) followed by an arbitrary expression. The expression is evaluated yielding an integer, and transfer is made to the statement having the appropriate number. Return from within an APL subroutine is indicated as a nominal transfer to any statement number not actually occurring in the subroutine. It is sometimes inconvenient to refer to a statement by its

explicit number, especially if the body of the function within which it occurs is to be modified. For this reason, a label option is provided within APL. If a statement occurring in the body of a function definition is prefaced by a name and a colon, then at the end of the definition the name is assigned a value equal to the statement number. Using this label name following a transfer arrow, even as part of some more complex expression, allows one to avoid the annoyances that might result from the direct use of numerical expressions. APL routines may be called recursively; the system uses an internal stack to handle this recursion.

Relevant details concerning the inner workings of the APL function-definition mechanisms are as follows. Individual statements are stored in a lexically analyzed but not syntactically reordered form. Statement bodies are stored singly in the APL array area. A subroutine is defined by an ordered vector V referencing the successive statements belonging to it. This vector V is established when the subroutine definition is closed; while it is open, pointers to the statements comprising a subroutine are maintained as the successive elements of a list, allowing easy insertion and deletion of statements. The vector V defining a subroutine is prefixed by a field giving such additional information concerning the subroutine as its trace and stop vectors (see below) and its modification lock. A stored subroutine R may be reopened for modification by typing ∇ R.

Some few words containing the name-scoping rules used in APL are in order. Names are global unless specified as being local to a subroutine. A name F used globally as a function may not be used globally as a variable, but may be used locally as a variable within a subroutine G. If this is done, the function F becomes unavailable once the subroutine G is entered. Labels are treated as local variables within the subroutines where they occur, but are specially flagged to prevent their values (established when the function definition is closed) from being changed.

The symbol table entry defining a subroutine contains fields specifying such information concerning the subroutine as its length, the variables local to the subroutine, the subroutine's arguments, etc. With each statement in a subroutine APL associates a stop bit and a trace bit, these bits taken for all the statementsof a subroutine collectively constituting the stop vector and trace vector for the subroutine. If the stop bit attached to a statement is on, execution will be suspended and control returned to the keyboard whenever interpretation reaches the statement in question. If the trace bit attached to a statement is on, a report giving subroutine name and statement number will by typed whenever interpretive control reaches the statement. The current value of the stop vector associated with the subroutine F may be accessed by using its special symbolic name SΔF in an APL expression; similarly, the stop bits may be modified by making a normal APL assignment to the pseudo-variable SΔF. The trace vector is referenced in similar fashion as TΔF.

Various other APL features used in the definition and debugging of APL subroutines and as general conveniences affect the detailed syntactic definition to appear below. If the definition of a function is opened by the special symbol $\nabla\!\!\!\!\!-$ rather than the normal ∇, or closed using $\nabla\!\!\!\!\!-$ in place of ∇, the function is locked and subsequently the function definition may not be displayed or modified, nor may the trace or stop vectors associated with the function be changed; this feature is useful in protecting programmed exercises intended for classroom use. A locked function may, however, be erased; but only by the owner of the library to which it belongs. APL assists the editing of unlocked functions in various ways. An existing function F may be reopened for modification by typing [N], where N is 1 more than the number of statements comprising F, indicating that it is ready to accept a new line to be appended at the end of F. Typing [$\square$] will display all of F and

cause the system to await entry of an additional line; typing [⍞ N] will display all statements from N onward and cause the system to await replacement of the last statement of F. Typing [N⍞] will display statement N and cause the system to await replacement of this statement. Typing [N⍞M] invokes a special mechanism for characer-by-character editing of statement N; the reader is refered to the APL Programmer's Manual for the details of this mechanism and the conventions associated with it. If [N] is typed, the system awaits insertion or replacement of line N; use of a decimal for N allows insertion. When a function is closed its statements are appropriately ordered and successively renumbered. Function definition or display may also be requested on a single line. Thus, for example, typing ∇ F[3]X<--X+1∇ opens the definition of F, enters a new third line consisting of the statement X<--X+1, and closes the function F, locking it.

The APL system monitors the activity of programs running under it, thereby accumulating information of potential dynamic interest to system users; various dynamic items of program status information may similarly be of use. Such information may be accessed using one of a limited number of system pseudo-variables having such forms as I19, I29, etc., in APL expressions. The information made available in this way includes such quantities as the time of day, the date, the amount of central processor time used in the current session, the vector of statement numbers belonging to the current status indicator, the number of the next statement to be executed, etc.

APL statements, whether belonging to prestored user-defined subroutines or received directly via keyboard transmission, are stored in a lexically analyzed but not syntactically reordered form. This assures compact storage and allows the symbolic version of any statement to be reconstructed from its stored form.

The APL interpreter embodies conventions facilitating the suspension and resumption of program runs. The execution of/function F may be stopped before completion in a variety of ways: by an error report, by an attention signal, or by the action of the stop bits comprising the vector SΔF mentioned above. No matter how stopped, status information on the function is still active and function execution can later be resumed. In this state the function is said to be suspended. Entering a transfer →K will cause execution of the syspended function to be resumed, beginning with statement K. The possibility of fully dynamic suspension and resumption is based upon the availability within the interpreter of complete dynamic program status information. (See below for a more detailed discussion.)

In the suspended state of any function, APL statements intended for immediate interpretation may freely be entered at the console; these statements may call on additional APL functions, which may in turn lead to new function suspensions and to the entry of additional statements from the keyboard. Statements α entered for execution from the keyboard are called keyboard statements. Note that on completing the interpretation of a keyboard statement α entered from the console during suspension of some particular function F, we return once more to the suspended state of F. Suspension of F will only be dissipated when a transfer is interpretively executed from the keybarod at the level of F.

As has been noted, pointers defining those portions of the control and argument stack that correspond to the interpretation of the most current APL statement are available at all times. If an error occurs during interpretation of this statement, the corresponding stack section is cleared, and execution suspended just prior to next following statement. In addition, the availability at all times of complete

status information allows a diagnostic trace of the set of subroutines momentarily activated to be given upon system-level request at any time during the suspended state of an APL run.

Expressions within APL are written in what is essentially right-to-left Polish form, a fact simplifying certain aspects of the APL syntax. Table III below gives the syntax of APL in the generalized Backus metalanguage, while Table IV contains a summary account of the lexical types and the generator routines occurring in Table III. In order to deal effectively with the right-to-left execution order of the APL expression, we have segregated the syntactic definitions comprising Table III into two logical sub-parts, a first part detecting system commands and handling subroutine definition and editing statements, and a second part defining the APL expression interpreter. We suppose the tokens for syntactic analysis to be provided by a simple token emmiter routine containing an emission-order switch; the first part of our syntax assumes a left-to-right emission order for tokens, while the epxression-interpreting portion of the syntax table assumes that the token emitter has been switched over to right-to-left emission. The necessary reversal of scan order is to be accomplished in the generator routines forward (see below) which switches to the keyboard-oriented forward scan, and reverse, which goes to the reverse scan. The routine forward serves in addition for initial keyboard input of a statement, unlocking the APL keyboard to permit reception of a line, waiting for this line to become available, and calling a lexical scanner to break a newly received line into tokens. Two principal syntactic entry points correspond to our division of the APL syntax into a forward-scanning keyboard-input section and a backward-scanning expression evaluator; these appear in Table III as the syntactic types <transmsn> and <expr>.

Table III. APL Syntax

```
                <-basic syntactic definition separating system commands,
                  function definitions, and statements>
<transmsn>    = .forward ')'<syscommand>..transmsn/..comnderr
              = '∇'.lock(o)<fhead>..fdef/.transmsn.synerr
              = '∇'.lock(1)<fhead>..fdef/.transmsn.synerr
              = <stmnt>/.synerr
                <-syntactic definitions applying to subroutine head>
<fhead>       = <*name>'←'. result(1)<*name>..ftail/.synerr..synerr
              = .result(0)..ftail/
<ftail>       = <*name><*name>.oklocal<locals*>.testlocs...synerr<*edge>
                  .post(2)...synerr..fdef/..dubfn.synerr
              = <-name>.newtest..oldedit.oklocal<locals*>.testlocs...synerr
                 <*edge>.post(0)...synerr.fdef/.synerr
<dubfn>       = <- name name>.oklocal<locals*>.testlocs...synerr<*edge>
                 .post(1)...synerr..fdef/.synerr
<oldedit>     = .open...synerr'['<distail>..fline/..synerr
                 <*edge>.setlength..fdefs/.synerr
 locals>      = ';' <*name>/.b.locerr
<locerr>      = .errnote/
                <-syntactic definitions applying to subroutine body>
<fdef>        = .startnmbr..fdefs
<fdefs>         .newnmbr.forward')'<syscommand>.numback..fdefs/..comnderr
                '[' <distail>..fline/..synerr
              = '∇'.numback.close..transmsn/.
              = '∇'.lock(1).close..transmsn/.fline
<fline>       = <nondel*> '∇'<*edge>.enter.close..transmsn/..synerr
              = <nondel*>'∇'<*edge>.lock(1).enter.close..transmsn/..synerr
              = <-nondel*><*edge>.enter..fdefs/.synerr
<distail>     = '□''['.display/.nobox.
              = <-□><*integer> ']'.lineset.display/.b.b
<nobox>       = <*integer>'□'<*integer>']'..linefix/.b.boxless..b
              = <-integer□>']'.lineset.disline.numback..fdefs/.b
<linefix>     = (We shall not show the action required here.  cf. APL
                 programmer's manual for details)
 boxless>     = <integer>']'.lineset..fline/.b
<nondel>      = <*name>/.
              = <*integer>/.
```

```
                = <*real>/.
                = <*quote>/.
                = <*fsymbol>/
                = <*sysname>/.
                  <-syntactic definitions defining the APL expression
                   interpreter>
<stmnt>         = .reverse..expr/
<expr>          = <*name>.testvar...fsymb..midexp/.
                = <vector>..midexp/.
                = <*quote>..midexp/.
                = ')'<expr>'('..midexp/..synerr.synerr
                = ']'<subs*>'['<*name>.subscript.domerr..midexp/..synerr.synerr
                = '⎕'.numin..midexp/.
                = '⍞'.symbin..midexp/.
                = <*sysname>.sysval..midexp/.expend

<subs>          = <expr>';'/..b
                = ';'.misdim/.

<fsymb>         = <-name> '△''⍫'.testfn(3)...synerr.tvector/.fcall.synerr
                = <-△name>'S'.testfn(3)...synerr.svector/.synerr
<fcall>         = .testfn(0)...synerr.call(0)...transmsn..expr/

<expend>        = <labpart >< edge>.clean.print.cleanup...transmsn..expr/.synerr
                = ':'< name>/..synerr

<midexp>        = < fsymbol>'.'< fsymbol><leftpart>.doinr...domerr..midexp
                   /.notsym.nodot..synerr
                = <-.fsymbol>'o'<leftpart>.doutr...domerr..midexp/.synerr.synerr
<nodot>         = <-fsymbol><leftpart>.dodyad...domerr..midexp/.
                = .domonad..domerr..midexp/

<notsym>        = ']'<expr>'['']/'< fsymbol>.doredn(1)...synerr..midexp
                   /..synerr.synerr.qualop.selecta
                = '/'< fsymbol>.doredn(0)...synerr..midexp/..selectb
                = < name>.testfn(2)..moncall <leftpart>.call(2)...transmsn
                   ..expr /..synerr
                  <-the next few lines describe the APL indexed assignment>
```

```
                = '←'< name>.testvar...mayfn.simpasin..midexp/.maygo.
                = <-←>']'<subs >'['< name>.indxasin...domerr..midexp
                  /..synerr.synerr.synerr
                = <-←>'  '.output..midexp/.synerr

                  < - the next line describes the APL transfer statement>
<maygo>         = '→'<labpart >< edge>.clean.dogo...transmsn.retval
                  ...expend..midexp/.expend.synerr

<mayfn>         = '∆''⊤'.testfn(3)...synerr.asintvect..midexp/.synerr
                = <-  name> 'S'.testfn(3)...synerr.asinsvert..midexp/.synerr

<moncall>       = .testfn(1)...synerr.call(1)...transmsn..expr/

<qualop>        = '⍒'.order(down)..midexp/.
                = '⍋'.order(up)..midexp/.
                = 'ø '.<lefpart>.dorot..midexp/.expand.
                = < -ø[q]val>.dorev..midexp/
<selecta>       = < -/[q]val><leftpart>.doselect...domerr/.synerr
<selectb>       = < -/val>.putslash<leftpart>.dodyad...domerr..midexp/.synerr
<expand>        = <- [q]val>' '<lefpart>.doexpand..domerr..midexp/.synerr

<lefpart>       = < name>.testvar..fsym/.
                = <vector> /.
                = < quote> /.
                = ')'<expr>'(' /..synerr.synerr
                = ']'<subs >'['< name>.subscript...domerr/..synerr.synerr
                = '⍞'.numin/.
                = '⍰'.symbin/.
                = <*sysname>.sysval/.

<vector>        = <num(1,*)>.makevect/.
<num>           = < integer>/.
                = < real>/.

<fsym>          = '∆''⊤'.testfn(3)...synerr.tvector/.backup.
                = <-∆name>'S'.testfn(3)...synerr.svector/.backup
<backup>        = .backl/
<syserr>        = .errout..transmsn/
<domerr>        = .errout..transmsn/
<syscommand>    = (The syntax of the APL system level commands is not shown.
                  See the APL programmer's manual for details.)
```

Implicit in the syntax shown in the above table is the structure of APL as an interpretive system. The APL features playing a central role in this regard have a significance for interactive interpretive systems extneding beyond the boundaries of the APL language itself, and are worth additional comment. The generator routines basic for this aspect of APL are dogo, cleanup, call, and errout, detailed algorithms for which are given below in Tables V through VIII. We note the following highlights.

i. Both a subroutine definiton mode and an execution mode are provided; the language interpreter gets its string of tokens indirectly either from the keyboard or from a prestored statement.

ii. A statement entered in execution mode from the keyboard is immediately classified either as a system-level command for immediate execution, as the beginning of a new function definition, or as an expression actually to be interpreted. On detecting the beginning of a new definition the system toggles into input status, in which status it remains until a special definition-close signal returns it to execution status.

iii. Transfers within and calls among pre-stored subroutines are executed interpretively, using a vector of statement addresses for transfers and using a hash-addressed master symbol table for calls. Transfers are handled simply by informing the system token emitter routine of the next statement to be executed, calls by stacking the system status information described below and passing the location of the first statement of a called subroutine to the token emitter.

iv. Interpretation of APL statements involves repeated parsing; the parser will use a recursion control stack and an argument-and-operator stack. The momentary state of any single-statement parse will be defined by the state of these two stacks and a small amount of additional information, such as the identity of the statement being executed, the last symbol in this statement which the parser has reached, etc. Since APL is interpreted rather than compiled, the most

Table IV. Lexical types and generator routines for APL syntax.

A. Lexical types.

<*name>, <*integer>, <*real>, <*quote>, <*edge>, <*fsymbol>, <*sysname>.

B. Generator subroutines associated with keyboard input and function definition.

forward- unlocks keyboard to receive transmission, awaits transmission, calls lexical scanner, signals token emitter for forward scan;

lock(yesno)- posts flag indicating lock/unlock of subroutine;

result(yesno)- posts flag indicating result/noresult for subroutine;

oklocal- drops local argument error flag;

testlocs- tests local argument error flag;

errnote - sets local argument error flag;

post(n) - posts definition of new subroutine with n arguments, checking on legality of name and correctness of local variable sequence;

newtest- returns 'no' if name designates existing subroutine;

open- opens subroutine for correction, tests for legality of opening;

setlength-sets input line number to end of current subroutine;

startnmbr-initialises input line number to 1;

newnmbr- prints and increments input line number;

numback- decrements input line number;

close- closes subroutine definition, posts lock;

enter- enter a new line into a function definition body;

display- display all lines from current position to end of subroutine, advance input line counter to end of subroutine;

lineset- establish specified input line number;

disline- display a single line of a subroutine;

C. Generator subroutines associated with the interpretation of APL expressions.

reverse- signal the token emitter to begin reverse scan;

testvar- test a name for validity as a variable;

subscript-carry out subscript expression interpretation, checking on validity of arguments;

numin-	await and accept numerical input;
symbin-	await and accept symbolic input;
sysval-	make value of system-dependent function available on argument stack;
tvector-	make value of subroutine trace vector available as argument;
svector-	make value of subroutine stop vector available as argument;
misdim-	post missing dimension pseudo-argument indicator on argument stack;
testfn(n)-	test name for validity as subroutine name, either with given number or with unspecified number of arguments;
call(n)-	subroutine call interpreter routine;
clean-	removes edge indicator and labels from top of argument stack;
print-	prints value of interpreted expression where appropriate;
cleanup-	terminates interperetation of prior statement, initialises for interpretation of next following statement, signals for keyboard or subroutine input through syntactic 'yesno' flag;
doinr-	executes inner product process, checking on argument validity;
doutr-	executes outer product process, checking on argument validity;
dodyad-	executes dyadic operation, checking on argument validity;
domonad-	executes monadic operation, checking on argument validity;
doredn(yesno)	executes reduction process in qualified or unqualified form, checking on argument validity;
simpasin-	execute non-indexed assignment operation;
indxasin-	execute indexed assignment operation, checking on argument and index range validity;
output-	print value of expression;
dogo-	interpreter of APL transfer statement;
retval-	check subroutine for returned value, setting 'yesno' flag and value-print flag accordingly;
asintvect-	make assignment to function trace vector;
asinsvect-	make assignment to function stop vector;
order(updown)	calculate permutation ordering quantities in specified manner;
dorot-	execute rotation operation in qualified form;
dorev-	execute reversal operation in qualified form;
doselect-	execute 'compress' operation in qualified form;
putslash-	put dyadic slash operator at top of argument stack;
doexpand-	execute 'expand' operation in qualified form;
makevect-	assemble string of numerical constants into a constant vector;
backl-	back up one token in input string;
errout-	issue diagnostic, terminate interpretation of current statement.

characteristic generative actions taken by the APL parser will consist rather in the interpretive execution of a local code fragment than in the emission of code. In connection with such an action one will typically remove an operator and a pair of arguments from the argument-and-operator stack, replacing them with a reference to an operation result. Calls on user-defined functions F, however, must be treated somehwat differently. In interpreting such a call, the following actions must be carried out.

a.) Record at the top of the control stack any information required by the interpreter which has not already been stacked. This will include:

ai.) The name of the subroutine containing the statement currently being executed;

aii.) The number of the currently executing statement within this subroutine;

aiii.)The position in the current statement of the last symbol scanned by the parser;

aiv.) The length of the control stack and argument stack segments associated with the currently executing statement.

This information may appropriately be called the invocation information belonging to a particular entry of control into a function F.

b.) On entering a subroutine F, reinitialize all non-argument variables local to F by stacking references to the prior values of these variables and by reflagging the variables as having undefined values. Initialize the arguments of F by stacking references to their prior values and causing each argument variable to reference a newly created copy of whatever argument values are transmitted to F from the point at which it is called.

c.) Establish new current sub routine and statement information, as follows:

ci.) Current subroutine is F;

cii.) Current statement number is 1;

ciii.)Last symbol scanned by parser is 0;

civ.) Current control stack top pointer and argument stack top pointer are recorded.

v.) On returning from a called subroutine F, one merely unstacks the previously preserved status information, returns any output argument, restores all F-local variables to their prior values, and recommences interpretation at the point from which the subroutine was originally called. A very similar mechanism is used to return control to the keyboard on a stop or error condition and to resume operation subsequently. The occurrence of a system error or stop signal stacks all necessary status information and returns control to the keyboard which is placed in execution status. Note that by treating calls on undefined functions as errors we allow partially programmed collections of subroutines to be executed and give the system user a chance either to define a missing subroutine when it is called or to provide the subroutine result manually by direct entry. This scheme for treating calls also forms the basis for the flexible and convenient diagnostic traces and pauses used in APL.

As has been noted, the syntactic signal which returns control from the keyboard to a suspended subroutine is the occurence of a transfer statement. Such a statement is in effect treated as a return from the keyboard to a suspended subroutine, followed at once by a transfer to the indicated destination within the formerly suspended subroutine. (cf. Table V for additional details of this mechanism.)

vi.) When the interpretation of a statement is either completed normally or terminated on the occurence of an error, all temporary information, whether contained in the parse-control stack, the argument stack, or elsewhere, is erased before control is passed either to the keyboard or to the next prestored statement to be interpreted.

Table IV lists the lexical types and describes the gen-

Note in connection with the above table that an attention signal transmitted from the keyboard will set a stop flag within the APL system. This flag is consulted whenever interpretation of a new statement is to begin; if it is found to be set, the action appropriate for a statement bearing a stop bit is taken, and the stop flag dropped.

erator routines which occur in the syntactic definition of APL given in Table III. The lexical types which occur are rather standard: Symbolic names, symbolic integers and real numbers, quoted character strings handled internally as character vectors, specially prefixed system function names, special symbols for built-in APL functions, and a special mark representing either the left- or right-hand boundary of a statement are admitted. The lexical analysis necessary to break character strings into these tokens is straightforward, and we shall not describe it in any additional detail. On the completion of lexical analysis, the tokens representing a lexically analyzed input string are either stored as part of a subroutine or passed directly to a token emitter for subsequent syntactic analysis.

Table IV groups the APL generator routines into two main subsets, namely those routines associated with subroutine definition and a forward scan, and those routines associated with statement interpretation and a backward scan. In many cases, the action of a generator will be clear from the brief description found in Table IV. In connection with the subroutine definition generator group, note that flags indicating whether a subroutine is to be locked on closing and whether the subroutine will return a value are set up by lock and result respectively; these flags are then used by close, and in the case of a new subroutine by post, which between them accomplish most of the labor of subroutine definition. The subroutine post is invoked just after the header line of a new subroutine has been syntactically analyzed; it checks the received information concerning subroutine name and local variables for semantic admissibility and goes on to establish a subroutine data structure into which the successive lines of a subroutine doby may be incorporated as received. Note that the syntactic as distinct from the semantic validity of a local variable list appended to a subroutine header is established by the elementary joint action of the three routines okaylocal, errnote, and testlocal. With the exception of the two key routines open and close,

both of which perform fundamental operations in connection with subroutine definition, most of the remaining generator routines of the first set are elementary and serve to display lines of a subroutine either singly or in groups, to manipulate a position counter defining the next position at which a subroutine line is to be received, etc. Additional details concerning the open and close subroutines are given below as part of an account of the two slightly different data structures used to represent subroutines during definition and during execution.

Many of the interpreter-associated generator routines shown in Table IV are of straightforward action. One subgroup, consisting of subscript, doinr, doutr, dodyad, domonad, doredn, dorot, dorev, doselect, doexpand, and order, implement the various basic dyadic and monadic operations of the APL system. Most of these routines also check their arguments for validity, and will set the syntactic error flag on encountering invalid arguments. Note that a variable whose value is undefined will normally be an invalid argument.

A second subgroup of generator routines, numin, symbin, tvector, svector, and sysval dynamically create argument values in slightly non-standard ways, either receiving keyboard input, supplying values from fields maintained within the interpreter itself, or making up bit vectors from the trace and stop information associated with subroutines. Note that the APL implementation attaches the trace and stop bits associated with the subroutine directly and individually to the subroutine statements, so that, for example, the insertion of additional statements into a subroutine will not change the meaning originally belonging to a particular trace bit. The numerical input routine numin will itself issue a diagnostic and wait for corrected input on receiving input in illegal format.

The generator routines simpasin, indxasin, asintvect

perform various types of value assignment operations; indxasin also checks its arguments for validity and sets the syntactic error flag if necessary. The routines asintvect and asinsvect set the trace and stop bits associated with the statements of a subroutine. The two routines print and output are responsible for APL standard form output. The second of these routines is unconditional and straightforward; the routine print, however, which is called on completing the interpretation of an entire APL line, is called only conditionally. It will output the calculated value of the line provided that the last executed operation in the line is not either an assignment or a function call not returning a value. The print condition flag necessary for this purpose is manipulated as follows. In beginning the interpretation of a new statement, the flag is set to its print condition, either by the routine forward, or by the routine testvar. The execution of any operation returning a value will again set the print flag; the execution either of an assignment operation or a function call not returning a value will drop the flag. The necessary flag manipulation action is accomplished by simpasin, indxasin, asintvect, asinsvect, and output.

The four routines call, cleanup, dogo, errout play basic roles in the logical flow of the APL interpretation; their action, represented in detail in the four tables which follow immediately below, has been survey in preceeding paragraphs. The generator misdim is called during the syntactic analysis of an APL subscripted expression to supply a marker to be stacked in place of a missing index when such cases as

A(;J;;).

are encountered. The interpreter subroutines subscript and indxasin will subsequently take appropriate interpretive action according to whatever pattern of explicit and nominal indices is supplied. The subroutine clean is used in auxiliary fashion to remove from the syntactic argument stack the edge

marker and a string of labels which may appear on the argument stack during the terminal scanning of the extreme left hand portion of an APL statement. The subroutine makevect assembles a string of numerical constants into a constant vector; note that in a stored APL statement the successive components of a constant vector are represented simply by the succession of integers representing these components. Note also that it is only within the body of a constant vector that a blank can be used to represent concatenation.

The few generator routines appearing in Table IV but not discussed in the foregoing brief survey all have elementary actions which are described sufficiently well by the summary given in Table IV.

We conclude the present section with an account of the principal internal data structures used by APL and their role in APL interpretation. Aside from the basic symbol table, these data structures are generally arrays. APL arrays are kept contiguously in storage blocks allocated dynamically; each array is prefixed by a descriptor word giving its size, its type (logical, numerical, character, or stored text), and relevant additional attributes, and containing a live-dead bit indicating whether or not the array has passed out of use. In addition, this descriptor word points to the symbol table entry of the temporary or programmer variable having the given array as its value. In those cases in which an array which is already the value of one programmer variable is assigned as the value of a second such variable, a second copy of the entire array is created. Thus, a single array is never the vlaue of more than one variable. This permits the following very simple garbage collection scheme to be employed. A pass over the whole array storage area is made. Space is repacked by moving arrays to eliminate dead arrays. When an array is moved, the unique symbol table entry referencing it (and which it references) is reset to show its new position.

Table V. Algorithm for the APL transfer interpreter.

subroutine dogo

```
        if current statement is at keyboard level, go to key;
insub:  if transfer target is not integer, go to error
        if transfer target is out of current subroutine range, go to
         popup;
        post number of next statement to execute;
        do current statement cleanup operations;
        if trace bit for next statement is on, go to trace;
tonext: if stop bit for next statement is on, go to stop;
        if stop flag is on, drop it and go to stop;
        transmit location of new statement to emitter routine;
        set syntactic yesno flag to 'yes' state;
        return;
stop:   set syntactic yesno flag to 'no' state;
        return;
trace:  print function name and statement number;
        go to tonext;
popup:  do current statement cleanup operations;
        do subroutine return unstacking operations;
        if subroutine returns a vlaue, place this value on top of stack;
        post flag to indicate whether subroutine returns a value;
        return;
key:    erase current statement;
        do current statement cleanup operations;
        if control stack is empty, go to error2;
        do control information unstacking operations;
        go to insub;
error1: issue diagnostic 'invalid transfer target';
        post zero as number of next statement to execute;
        do current statement cleanup operations;
        go to stop;
error2: issue diagnostic 'transfers illegal when no subroutine called';
        go to stop;
```

- end of subroutine algorithm -

Table VI. Algorithm for the APL end-of-statement interpreter.

```
        subroutine cleanup;
        if current statement is at keyboard level, go to key;
        construct statement number n by incrementing current statement
          number;
        place this statement number on stack top;
        call dogo;
        return;
key:    erase current statement;
        do current statement cleanup operations;
        set syntactic yesno flag to 'no' state;
        return;
```

- end of algorithm -

Table VII. Algorithm for the APL subroutine call interpreter.

```
        subroutine call (n);
        stack information on current state of parse;
        stack values of local variables;
        for each subroutine statement which contains a label, find the
          labels which it contains and set each label's value;
        bring in external values of any subroutine arguments;
        initialise all information descriptive of subroutine;
        post 1 as number of next statement to execute;
        if trace bit for first statement is on, go to trace;
tofirst: if stop bit for first statement is on, go to stop;
        signal emitter routine on location of new statement;
        signal emitter routine for reverse emission;
        set syntactic yesno flag to 'yes' state;
        return;
stop:   set syntactic yesno flag to 'no' state;
        return;
trace:  print function name, statement number is l;
        go to tofirst;
```

- end of algorithm -

Table VIII. Algorithm for APL error-handling subroutine.

```
subroutine errout

        issue appropriate diagnostic, indicating point of error
         within current statement;
        if current statement is at keyboard level, go to key;
        print current subroutine name and statement number;

        post current statement number as number of next statement
         to execute;
        do current statement cleanup operations;
        stack control information;
        return;
key:    erase current statement;
        do current statement cleanup operations;
        return;
```

- end of algorithm -

A symbol table entry has the form shown in Figure 1.

name pointer	name length	value pointer	dimension pointer	hash chain

Figure 1. APL symbol table fields.

The name pointer field in each symbol table entry references a location in a packed character array at which the external form of the symbol is stored; the length field defines the extent of the symbol. The value pointer locates that array, in the APL array area, containing the current value of a symbol, or, if the symbol has no current value, is blank; the dimension pointer similarly locates the array of dimensions of an APL quantity. A scalar quantity will have a blank dimension field. The hash chain field is used to chain together all symbols sharing a common hash address.

A symbol table entry representing an APL subroutine name rather than an array has similar fields, but these fields have a somewhat different significance. In this case, the value pointer references a vector defining the successive statements comprising a subroutine, while the dimension pointer references a vector defining the variables local to the subroutine.

An APL array is headed by a set of fields having the form shown in Figure 2.

type	length	symbol table entry	live bit	attributes
array word 1				
array word 2				
...				

Figure 2. An APL array with its header fields.

The live bit is used in the manner outlined above to control garbage collection. The array length is given in bytes. The type of an array will be one of the following: real, integer, logical, character, statement definer, local variable definer, set

of statements. Set-of-statement arrays and local variable definer arrays are referenced by symbol table entries representing user-defined subroutines; statement definer arrays are referenced through set-of-statement arrays. The symbol table entry field of an array header either defines the symbol table entry having the given array as value or dimension array, or in certain cases described in more detail below, some other field through which the array is located.

The attribute field associated with an array has a significance that varies with the type of the array. In the case of a logical (packed bit) array, the attribute field contains an integer between 0 and 7 describing the number of unused bits in the last byte of the array. An integer array of length 1 may have the special attribute "unmodifiable" indicating that it represents the value of a label.

The symbol table entry corresponding to an APL subroutine will reference an array of statement-set type and an array of local variable definer type. The first of these arrays will have an open-closed attribute and a locked-unlocked attribute; the local variable definer array will have a returned-value attribute, and an integer attribute describing the number of arguments of the associated subroutine. When a subroutine is closed, the associated statement set vector will have the form shown in Figure 3.

statement set	length	symbol table entry	live bit	closed
location of subroutine header				
location of statement number 1				
location of statement number 2				
...				

Figure 3. The statement set vector of a subroutine, closed form.

This form of the statement set vector is appropriate for the efficient interpretation of APL transfer statements occurring within a subroutine.

When the same subroutine is opened, either during its initial definition or for modification, the statement set vector is put into a variant form in which statement references and symbolic statement identifiers alternate. This open form of a statement set vector is shown in Figure 4.

statement set	length	symbol table entry	live bit	open
	line identifier 0			
	location of line 0			
	line identifier 1			
	location of line 1			
	...			

Figure 4. The statement set vector of a subroutine, open form.

As lines are added to an open subroutine, whether it be for insertion or as replacement lines, they are successively added, together with their labels, to the end of the open statement set vector. Thus, the logical body of an opened subroutine can be defined at each moment as the collection of its lines appropriately reordered in terms of their identifiers, and where the latest of any group of lines all having the same identifier is to be kept while the others are to be eliminated. The implied "sort and comb" action will be performed automatically whenever a whole sequential section of the subroutine is displayed; this same action is performed again when the subroutine is closed, at which time the statement identifiers occurring in the open form of a statement set vector are eliminated, putting the statement set vector into its closed form. The APL subroutine close

routine also examines all the lines comprising a subroutine, detecting all labels, adding them to the local variable definer vector for the subroutine, appending the appropriate value to each label, and checking the use of labels for semantic correctness. Figure 5 below shows the form of a local variable definer vector.

	local definer	length	symbol table entry	live bit	argument descriptor
0	symbol table pointer 1				
0	symbol table pointer 2				
0	...				
1	symbol table pointer n (for label)				
	local value				
0	symbol table pointer n+1				
	...				

Table 5. The local variable definer vector for a subroutine.

An ordinary local variable is flagged as such, and represented simply by a pointer to the variable's symbol table entry; a label is specially flagged, and the corresponding symbol table pointer is followed by field containing the value of the label.

An APL statement is represented by an array of the form shown in Figure 6.

statement	length	statement set entry pointer	live bit	attributes
symbol pointer number 1				
symbol pointer number 2				
...				

Figure 6. The APL statement-definer array.

Each statement definer points to the statement set entry referencing it. The two attributes of a statement definer are the stop bit and trace bit of the corresponding statement. Keyboard statements are referenced through a special statement set vector which may be considered to define the members of a nominal (keyboard) subroutine.

The syntactic argument stack is used to store the information which must be preserved during an APL subroutine call. Figure 7 shows the arrangement of information in the section of this stack corresponding to a partially interpreted statement S during a period when the interpretation of S has been interrupted by a call to a subroutine. All the various types of stack entries shown in Figure 7 bear distinguishing flags, allowing the routines which process them to take appropriate action in every case.

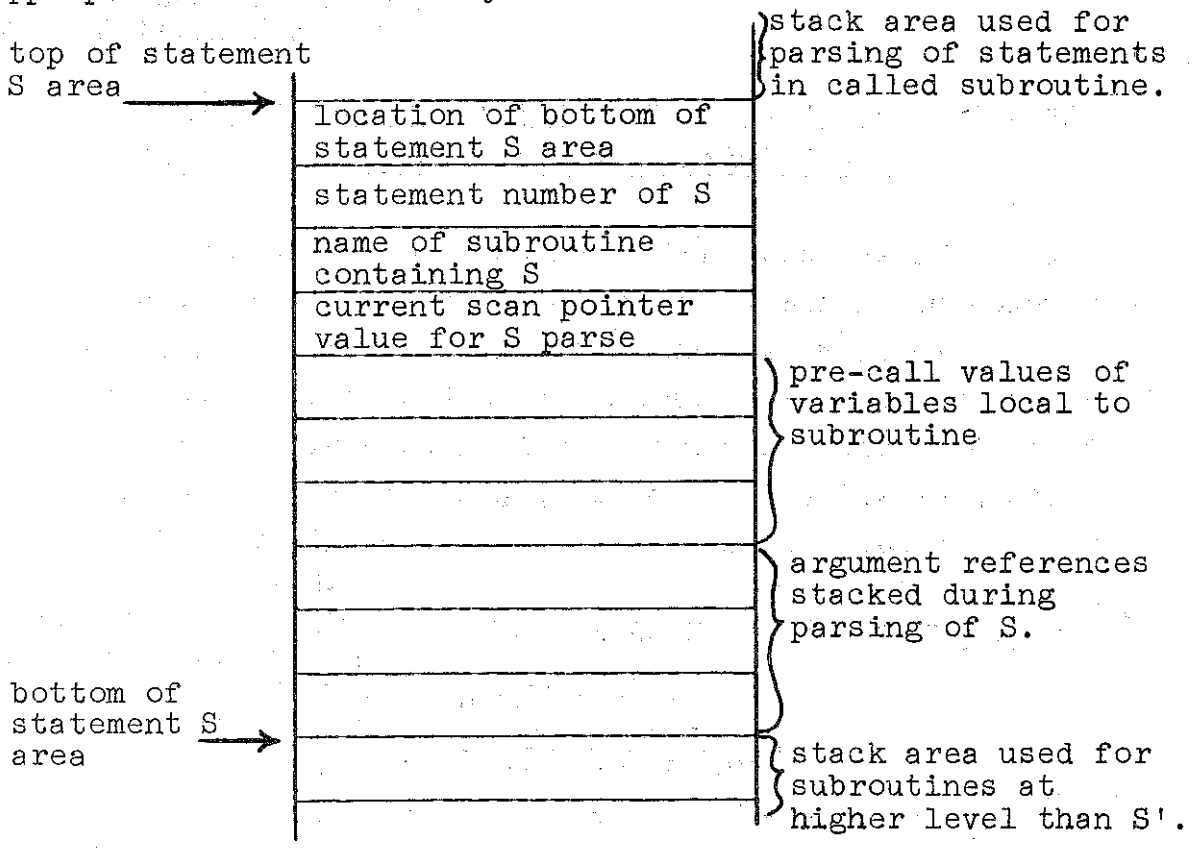


Figure 7. Layout of argument-stack information during a call from a statement to a subroutine.

The same remark applies to the entries in the corresponding control stack section, shown in Figure 8. Note that, since the call-associated information is kept in the argument rather than the control stack, the control stack contains a smaller variety of fields than the argument stack.

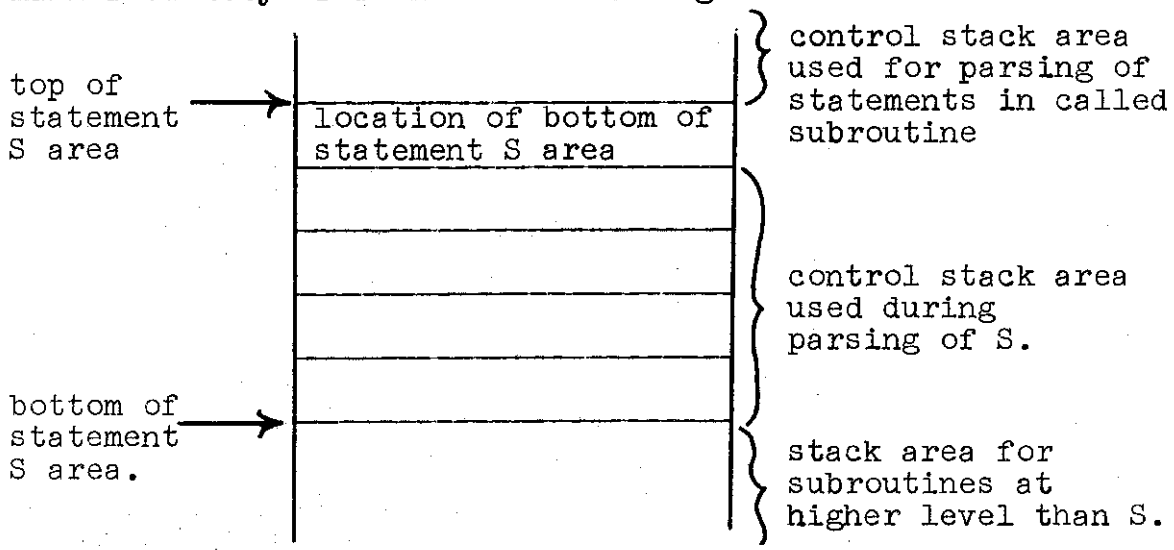


Figure 8. Layout of control-stack information during a call from a statement to a subroutine.

When the execution of a subroutine is suspended by the occurrence of an error, external signal, or stop bit and control is returned to the keyboard, information is stacked in a pattern logically resembling that shown in Figure 7. Since in this case no partial parse information or local variable pre-call value need be preserved, the argument stack area has the simpler form shown in Figure 9.

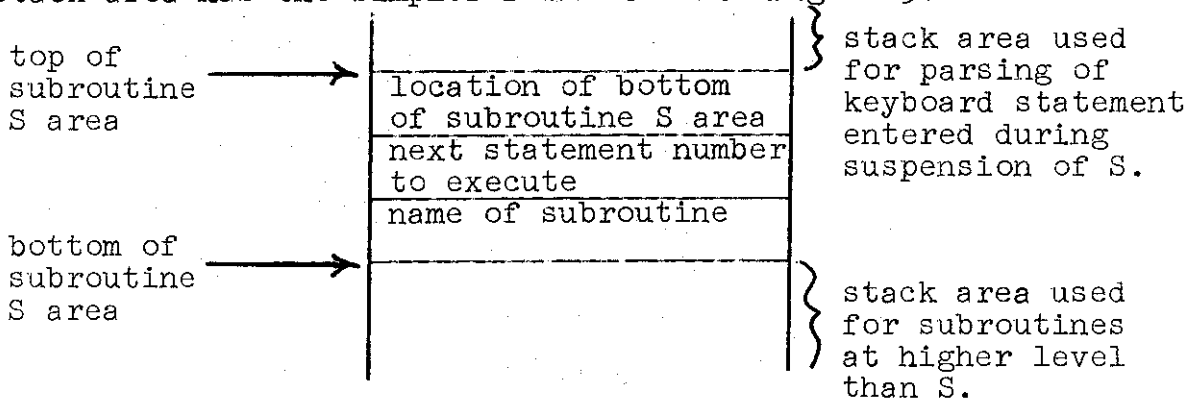


Figure 9. Layout of argument-stack information during suspension of a subroutine.

CHAPTER 8. THE SELF-COMPILING COMPILER.

One of the most serious long term problems in programming is the incessant and growing burden of reprogramming inevitably imposed by the continuing rapid development of physical hardware. The use, for the great bulk of programming of source language code reduces this problem by several orders of magnitude below the catastrophic proportions which it would have if machine language programming were still common. Nevertheless, even the small discrepancies in source language (say, FORTRAN) dialects available on different machines are productive of considerable difficulties. This difficulty, which in overall terms is proportional to the total accumulation of source code which must be carried from one machine generation to the next, is itself of growing significance and must, unless significant further steps of programming language standardization are taken, inevitably choke off the continued growth of society's library of effectively available programs. Dialectal variations in machine independent source languages like FORTRAN have one of their irreducible sources in the substantially different word lengths and the slightly different arithmetics with which machines are built. A more subtle but also quite significant source of dialectal variations lies in the tendency of even the most conscientious programmer to introduce small, possibly random variations in the action of a compiler being written de novo, especially in those inevitably numerous cases in which the "official" specification of the language is marginally incomplete. In the present chapter, we should like to propose a scheme for attaining a quite considerable measure of additional standardization in the definition of source languages, and moreover of making the process by which a source language is carried over to a new machine simpler and more efficient. The scheme to be described is one which has often been considered, and in some cases even implemented.

It involves the use of a compiler written in its own language, and capable therefore of compiling itself over to a new machine. Our effort will be to define and isolate to a maximum extent, the necessary machine dependencies residually inherent in such a compiler. By doing so, we aim to arrive at a system of programs which may be "bootstrapped" as smoothly and automatically as possible.

We should like, in order to explain our intent, to offer a suggestive biological analogy. The whole symbolic structure of programs written in a given source language, and ultimately carried by the compiler for the language, is capable (if it can run at all) of providing complex function and may be likened in biology to an extensive chromosome describing the multiple functions of a cell. Assuming that the few necessarily machine dependent parts of this possibly very large symbolic structure are carefully isolated into a small subsection of code, we may regard this part of the total symbolic structure as the description, to the symbolic entity (or chromosome), of its external environment (the machine). By changing only the machine dependent part of the larger structure we "mutate" the chromosome, producing a modified version of it which is capable of "infecting" a new environment. Thus, by proper design of our chromosome, we provide a highly "infectious" structure capable of spreading rapidly into a variety of environments. The presence, on the other hand, of a large number of implicit or hidden machine dependencies within a symbolic structure makes it highly environment-specific and therefore "uninefectious." Since technological development is bound to change the environment within which programming systems exist, it is clear that the uninfectious symbolic structures will be "killed"; only the most highly "infectious" structures can survive long enough to evolve to any very impressive level of function.

We may note, in analyzing the necessary parts of a self-compiling system, that

a) since it cannot safely be assumed that all necessary

information structures are available "in memory," an input/output function must be provided within the system.

b) This input/output function ought, for the sake of generality and convenience, involve the logical ability to reference arbitrary numbers of external files.

c) Such an ability implies the availability of at least a minimal operating system. For this reason, our discussion will, in the present chapter, go slightly beyond compiler theory proper and touch on some of the basics of operating system design.

The problem that must be faced is the following. We wish to define languages entirely in their own terms. For syntax, this is easy: we have only to use any reasonable metalanguage capable of describing code generation. This metalanguage can, as we have seen repeatedly in the preceding chapters, readily describe itself. The semantic side of our problem however, requires that we have available a language in which generator routines, symbol table manipulating routines, etc., as described in the preceding chapters, may be expressed. As seen in these chapters, this requires that we express a collection of algorithms more varied, less highly standardized, than those involved in syntax analysis. For this reason, we require for the expression of our generator algorithms a language going somewhat beyond what is appropriate for a specialized metalanguage. Such a language may appropriately be a language of algebraic type (built around a fairly standard assignment statement) which, on the one hand, omits those many features of algebraic languages not needed for the rather limited purpose intended, but on the other hand, includes some essential supplements. Foremost among these is the ability to define the bit-field-size of symbolic quantities, an ability that, as we have noted, is absolutely necessary to secure strict machine independence. Such a language will be described in the present chapter. Before doing so, however, we shall describe

the overall flow of the process by which we intend our symbolic structure (compiler) to pass from one machine to another.

Note first that, in order to isolate machine dependencies to a maximum degree, we insist that the compiler be written in its own language, and that it contain no scattered data statements descriptive of the particular machine on which it is running. Ideally the only machine dependent part of the compiler proper should be a single block of parameter statements defining such irreducibly necessary machine dependent facts as the machine word size, the size of the character field, the size of the symbol table and other main compiler tables appropriate in view of available memory size, etc. On the other hand, if the system is to run, connection with actual machine order codes must be established somehow, since the octal order codes for accomplishing addition, multiplication, etc., on the one hand, and the basic "driver" code sequences for I/O functions, etc. must be available somewhere. To narrow and standardize the symbolic code → machine code interface, we shall agree to place all this machine dependent code in a single binary file of data which we shall call the machine block. We assume that, whenever a program runs, this machine block is present in low core. We assume that the only information about the machine block available to the compiler, is a list defined by an initial parameter-type statement of the addresses, at which the machine language functions ultimately required for the running of symbolic programs are to be entered. Call the binary file of machine code constituting the machine block in the sense just explained. We assume, in accordance with the above, that with every compiled and in its low core locations the compiler loads the block MB; alternately, we could assume that the block MB is re-entrant, permanently resident in low core, and thus that it never requires special loading except at machine start-up. At any rate we shall always show the machine block MB as an explicit argument in a compilation.

Production of the machine block MB is, of course, the most laborious part of the work involved in transferring the self-compiling compiler to a new machine. MB will normally be produced from an assembly language written code. This code must be written with an eye to the machine's operation code set, channel configuration, external storage device configuration, etc. In order to facilitate the task of producing the block MB for a new machine, we shall always assume that a simple assembler, capable of assembling code for machines of arbitrary word length, is available within the general system containing the self-compiling compiler. We call the symbolic code for this assembler, written in the source language compiled by the self-compiling compiler, A. Intending also to imply that a rather minimal language is sufficient to express all necessary compiler functions, we call the language compiled by the self-compiling compiler LITTLE.

The total list of files, binary and symbolic, which therefore make up to minimal compiler system required for any given machine are as follows:

Binary Files: an executable LITTLE language compiler -- file C; an executable syntax expander code -- file E; a binary machine block -- file MB.

Symbolic Files: LITTLE-code for the assembler -- file SA; symbolic code for the compiler, written in a combination of syntax language and LITTLE-file SC; and symbolic code for the syntax expander, written in a combination of syntax language and LITTLE -- file SE.

The bootstrap procedure which takes us from an original machine M1 to a new machine M2 is then shown in Table I below. In preparing for the bootstrap procedure, we must

a) write an assembly language code SMB', which when assembled will produce the machine block MB' for the new machine.

b) hand-modify a few parameter statements in the symbolic code SC to produce a new symbolic code SC' which represents a LITTLE compiler for the machine M2.

c) hand-modify a few parameter statements in the symbolic code SE to produce a new symbolic code SE' which represents a syntax expander for the machine M2.

Note that, since the core programming system which we carry through the bootstrap procedure contains both a reasonably powerful systems programming language and an adequate syntax expander, the system is highly viable in the sense explained above.

TABLE I. BOOTSTRAP PROCEDURE FOR THE SELF-COMPILING COMPILER

A = C(SA,MB)	compile the assembler
MB' = A (SMB')	assemble the new machine's machine block
SC1 = E(SC')	produce a LITTLE-coded compiler, which will run on M1, but produce code for M2
C1 = C(SC1,MB)	produce an executable compiler, which will run on M1, but produce code for M2
C' = C1(SC1,MB')	produce an executable LITTLE-language compiler which will compile on M2 for M2
SE1 = E(SE')	produce a LITTLE-coded syntax expander for M2
E' = C1(SE1,MB')	produce a binary syntax expander for M2.

At this point, all the files MB', C', and E' are available to machine M2, and the LITTLE language has been carried to the new machine.

In Table I, we have assumed implicitly that the compiler system is capable of accessing a number of external files. On a large machine, provided with an adequate set

of external storage devices, these access operations might be accomplished by the automatic action of a family of storage device drivers, which, of course, form part of the machine block MB. On a small machine, where no such devices are available, the same goal might instead be accomplished by transmitting an operator console message requiring that a new paper tape be mounted. In either case, and of course neglecting speed considerations entirely, the logical structure of our boot-strap procedure remains the same.

It is now time to give a more detailed description of the LITTLE language. In defining this language, it is our intent to incorporate only that relatively minimal set of essential features which are truly necessary in the convenient expression of compiler function. As already noted, we shall build our language around an assignment of the standard form ⟨destination⟩ = ⟨expression⟩; this yields a convenient source form and has the second advantage of being close enough to the underlying machine load-store operations to facilitate the production of good target code.

Our compiler will have to perform arithmetic in bit fields of arbitrary length. Of course, it need provide only integer and not floating point arithmetic. The arithmetic operations provided will be: addition, subtraction, multiplication, and integer division, all of which are convenient auxiliaries in the expression of compiler functions. It is convenient and easy to provide a variety of additional boolean, comparison, and bit-field manipulating operators. These will be listed and described in more detail below.

In addition to the basic assignment statement, we require, of course, that a conditional transfer or "if" statement be available within our language, and that an unconditional transfer or "go to" statement, a "call" statement for the invocation of procedures of closed subroutine type, and a "read" together with a "write statement" be available also.

A "computed go to statement" is also provided; this statement gives access through the LITTLE language to the indexed transfer instruction available on most machines, and increases efficiency considerably in certain situations. Since the expression of a full compiler may require quite a bit of code, variable name conflicts can become a practical problem in writing a compiler. To avoid this, we find it appropriate to introduce a subroutine feature of the FORTRAN type, capable of providing name isolation. Accordingly, we introduce a "subroutine" statement, a "function" statement, and a "return" statement for returning from a called subroutine. This collection of nine members is our full list of executable statements, though, of course, we also require, as non-executable statements, and "end" statement for marking the end of a single subroutine, as well as a "fin" statement for indicating the end of all the statements of a program.

In addition to the executable statements described above, we include several essential declaration statements in the LITTLE language. In the first place, since we must provide variables representing bit fields of arbitrary size, we introduce a declaration statement SIZE which enables us to associate a bit field size with a variable name. This statement will have the form shown in the following example:

```
SIZE A(10), B(36), C(45)  .
```

A single index, which may be an arbitrary expression, may be attached to any variable name. Such an "indexed name" is taken to refer to the bit field obtained by offsetting, from the bit address belonging to the unindexed variable name, over the number

```
SIZ * (INDX - 1)
```

of bits; here, SIZ represents the declared size of the variable in question, while INDX represents the index value to be applied.

In order to provide for the storage of arrays on a reasonably rational basis, we include a "dimension" declaration of standard form in the LITTLE language. This enables us to declare the size of any array with which we are concerned, and enables the compiler to make corresponding storage space reservations.

To make it possible to initialize the values of variables, and to make it possible to collect all machine dependent program parameters into a single section of code, we provide both a "data" statement and a macro feature. The data statement allows data items of integer, octal, and alphabetic type, and has the form shown in the following example:

```
DATA  X=10/Y=B0775/JIM=3AJIM/STAR=1A* .
```

The more general form shown in the following example is also allowed.

```
DATA  ARRAY1=10,11,12,13,B16/ARRAY2=0,0,0/C=3AJIM .
```

Using such a data item, we can set the value of arbitrarily many locations of an array. The macro feature is explained below.

In order to improve the readability of LITTLE program listings, we include a "comment" statement in the language.

The LITTLE expression is composed in relatively standard fashion out of variable and function names, which are combined with members of a list of arithmetic, comparison, boolean, concatanation, bit-counting, and field extraction operators. With the exception of the triadic field extraction operator, all these operators are either diadic or monadic. The diadic operators are written in the usual infix notation, and the monadic operations are written as prefixes. Aside from the arithmetic operators, which are denoted by their usual special symbols, operators are designated by symbols obtained by enclosing a one, two, or three letter mnemonic between successive periods. More specifically:

1. The comparison operators are designated in the Fortran IV manner by .EQ., .NE., .GT., .LT., .GE., .LE.;

2. The boolean operators are designated in the Fortran IV manner by .OR., .AND., .EX., the last of these representing the "exclusive or", and by .NOT. which is a prefixed monadic operator;

3. A concatanation operator represented by the prefixed symbol .C. is provided, as is a monadic bit-count operator represented by the prefixed symbol .NB. A prefixed monadic "first non-zero bit position" operator .FB. is also provided.

4. The subfield extraction operator is triadic and is written in prefix form, as follows:

.F.⟨expr 1⟩,⟨*number⟩,⟨expr⟩.

This operator extracts, from the quantity obtained by evaluating the ⟨expr⟩ which it contains, a subfield whose size is given by the ⟨*integer⟩; the value of expr 1 determines the first position of this extracted field. The same form, written on the left side of an assignment statement, represents a field insertion operator (part word assignment). Accordingly, the four possible forms of LITTLE assignment statements are

⟨*name⟩ = ⟨expr⟩,
⟨*name⟩ (⟨expr 1⟩) = ⟨expr⟩,
.F. ⟨expr 1⟩, ⟨*number⟩, ⟨*name⟩ = ⟨expr⟩
.F. ⟨expr 1⟩, ⟨*number⟩, ⟨*name⟩ (⟨expr 2⟩) = ⟨expr⟩,

which represent respectively, the simple, the indexed, the partword, and the indexed partword forms of the assignment statement.

As in other languages, precedence rules are provided to reduce the number of parentheses needed to specify an expression. Table II below shows the precedence groups of LITTLE operators, arranged in order of increasing precedence.

Table II. The operators occurring in LITTLE expressions, arranged in order of increasing precedence.

1. (comparison operators) ..EQ., .NE., .GT., .LT., .GE., .LE.;

2. (arithmetic addition and subtraction) +, -;
3. (monadic arithmetic minus) -;
4. (arithmetic multiplication and division) *, /;
5. (boolean addition, concatenation) .ØR., .C.;
6. (boolean inversion, bit count, first bit operator)
   .NOT., .NB., .FB.;
7. (boolean multiplication, exclusive or) .AND., .EX.;
8. (subfield extract operator) .F. <expr 1> , <*number>, <expr>;
9. function references, array references.

Table III below gives a quick-reference list of all the allowed forms of LITTLE statements.

Table III. Quick-reference summary of LITTLE language.

Executable statements:

```
if (<expr>) go to label
go to <label>
goby <expr> (<label 1>,..., <label n>)
call <name> (<expr 1>,..., <expr n>)
return
continue
read <file>, <var>
write <file>, <expr>
a = <expr>
a (<expr 1>) = <expr>
.F. <expr 1>, <*const>, a = <expr>
.F. <expr 1>, <*const.>, a (<expr 2>) = <expr>
```

Declaration statements:

```
subr        <name> (<arg 1>, ..., <argn>)
fnct        <name> (<arg1>, ..., <argn>)
end
fin
data        <var1> = <const11>, ... <const1m> / <var2> = <const21>,
                  ..., <const2n>/.../<vark>=<constk1>...,<constkn>
dims        <var1> (<const1>), ..., <varn> (<constn>)
size        <var1> (<const1>), ..., <varn< (<constn>)
```

Table IV below contains a formal syntactic definition, written in the extended Backus meta-language, of the LITTLE language. In subsequent paragraphs of the present chapter, we shall describe the action of each of the generator subroutines appearing in Table IV; this will provide a very detailed account of the operation of the LITTLE compiler.

Table IV. Syntax and Generators for the LITTLE language.

```
<statement> = SUBR..subst/. = FNCT..funst/. = CØMM..comst/.
            = .chkend'/'<*name>.gengol(labop)'/'..unlab/.unlab.
              er(3).er(4) —
<unlab>   =  IF..ifst/.=GØ..gost/.=CALL..callst/.
          = GOBY..gobst/.=RETN..retst/. = END..endst/.
          = FIN..finst/.= DATA..datst/. = READ..readst/.
          = WRIT..writst/. = CØNT..cdend/. = DIMS..dimst/.
          = SIZE..sizest/. = <asgn> /.—
<ifst>    =  '(' <expr> ')' GØ TØ <*name> .genif..cdend/
             .er(5).er(6).er(7).er(9).er(9).er(3) —
<gost>    =  TØ <*name> .gengo..cdend/er(9).er(3) —
<callst>  =  <*name> '(' <plist> ')' .gencall(2)..cdend/
             er(10)..er(11).er(7)
          =  allri.gencall(1)..cdend/ —
<plist>   =  <expr> <exprc*> /. —
<exprc>   =  ',' <expr> /..er(6) —
<subst>   =  <*name> .gensub(no) '(' <arg> <args*> ')'
             ..cdend/.er(10)..er(11).er(7).
          =  .allri .gensub(no)..cdend/ —
<args>    =  ',' <arg> /..er(11) —
<arg>     =  <*name> .genarg/. —
<retst>   =  .genret..cdend/ —
<endst>   =  .genend..cdend/ —
<fin>     =  .terminate/ —
<data>    =  <ditem> <ditems*> ..cdend/.er(12) —
<ditems>  =  '/' <ditem> /..er(12) —
<ditem>   =  <*name> '=' <*const> <datents*> .gendat/.er(13)
              .er(8).er(13) —
<datents> =  ',' <*const> /..er(13) —
<readst>  =  <*name> ',' <expr> .genread..cdend/
             .er(14).er(14).er(14) —
<writst>  =  <*name> ',' <expr> .genwrit..cdend/
             .er(14).er(14).er(14). —
```

```
<comst>  =  <comp*> CMND..cdend/.er(12) —
<comp>   =  CMND.false/.
         =  <*eof>..er(15)/.
         =  <*any> .unstack / .er(12) —
<dimst>  =  <dimelt> <dimelts*> .. cdend/ .er(12) —
<dimelts>=  ',' <dimelt> / ..er(12) —
<dimelt> =  <*name> '(' <*integer> ')' .gendim/
            .er(11).er(5).er(11).er(7) —
<sizest> =  <sizelt> <sizelts*> .. cdend/.er(12) —
<sizelts>=  ',' <sizelt> /..er(12) —
<sizelt> =  <*name> '(' <*integer> ')' .gensiz/
            .er(11).er(5).er(11).er(7) —
<gobst>  =  <expr> '(' <*name> <namec*> ')'
            .gengoby/.er(6).er(5).er(3).er(7) —
<namec>  =  ',' <*name> /..er(3) —
<funst>  =  <*name> gensub(yes) '(' <arg> <args*> ')' ..cdend/
            .er(10).er(5).er(11).er(7) —
<asgn>   =  'left> = <expr> .genasin(AN)..cdend/
            ..er(8).er(6) —
<left>   =  '.F.' <expr> ',' <*integer> ',' <*name>
            '(' <expr> ')'.set(AN,4)/
            ..er(6).er(16).er(17).er(16).er(1),noinx.er(6).er(7)
         =  <*name> '(' <expr> ')' .set(AN,2)/.er(18)..er(6).er(7)
         =  <-name> .allri.set(AN,2)/ —
<noinx>  =  .allri.set(AN,3)/ —
<expr>   =  <arexp> <cop> <expr> .arith(AØ)/.b..b
         =  <-arexp> .allri/. —
<arexp>  =  <sterm> '+' <arexp> .arith(1)/.b..b
         =  <-sterm> '-' <arexp> .arith(2)/..b
         =  <-sterm> .allri/ —
<cop>    =  '.GT.' setop(AØ,3)/.
         =  '.LT.' setop(AØ,4)/.
         =  '.GE.' setop(AØ,5)/.
         =  '.LE.' setop(AØ,6)/.
         =  '.EQ.' setop(AØ,7)/.
         =  '.NE.' setop(AØ,8)/. —
```

```
<sterm>   =  -<factor> '*' <term> .arith(9).marith(1)/.term.b..b
          =  <--factor> '/' <term> .arith(10).marith(1)/..b
          =  <--factor> .allri. marith(1)/ —
<term>    =  <factor> '*' <term> .arith(9)/.b..b
          =  <-factor> '/' <term> .arith(10)/..b
          =  <-factor> .allri/ —
<factor>  =  <ℓsterm> '.ØR.' <factor> .arith(11)/.b..b
          =  <-ℓsterm> '.C.' <factor> .arith(12)/..b
          =  <-ℓsterm> .allri/ —
<ℓsterm>  =  '.NB.' <ℓterm> .marith(2)/..b
          =  '.FB.' <ℓterm> .marith(3)/..b
          =  '.NØT.' <ℓterm> .marith(4)/..b
          =  '.N.' <ℓterm> .marith(4)/..b
          =  <ℓterm>/.b —
<ℓterm>   =  <ℓfact> '.AND.' <ℓterm> .arith(13)/.b..b
          =  <-ℓfact> '.A.' <ℓterm> .arith(13)/..b
          =  <-ℓfact> '.EX.' <ℓterm> .arith(14)/..b
          =  <-ℓfact> .allri/ —
<ℓfact>   =  '.F.' <expr>, <*integer>, <expr>.genextr
             /..b.er(17).b
          =  '(' <expr> ')' /..b.er(7)
          =  <*name> '(' <expr>, <exprc*> ')' gencall(3)
             /.b..b.er(16).er(7)
          =  <-*name> .allri/ —
<er>      =  fetchp(EN).ermes(EN)..findend/ —
<cdend>   =  '.,' ..statement/ .er(2) —
<findend>=   '.,' ..statement/.
          =  <*eof> .terminate/.
          =  <*any> .nustack/.er(12) —
```

The general structure of the LITTLE compiler is as follows. A lexical scanner analyzes the successive characters of a source program to be parsed, dividing an input string into atomic words, calculating the values of constants as necessary, and initializing a hashed symbol table, i.e., a hashed array containing a unique reference to every variable name occurring in the given source program. This array is called HA; each item in it points to an attribute entry belonging to a second fundamental table VOA (variable and operator attribute array); entries in HA corresponding to variable names also point to a copy of the symbolic variable name to which they correspond. The lexical scanner provides successive lexical input atoms to the main parse routine. The main parse routine is a recursive syntactic recognizer written in the extended Backus metalanguage and calls various generator subroutines. Table IV lists the extended Backus metastatements defining the syntactic recognizer for the LITTLE language, and hence constitutes the definitive specification of the syntax of LITTLE. Table V lists the generator subroutines called directly from the main syntactic analysis program.

Table V. Generator subroutines called directly during parse of LITTLE programs.

A. Generator subroutines of elementary action.
allri, false, unstack, set, setop, fetchp, ermes

B. Generator subroutines corresponding to declaration statements.
gensub, genarg, genend, terminate, gendim, gensiz, gendat, (gengol).

C. Generator subroutines corresponding to executable statements.
genif, gengol, gencall, genret, genread, genwrit, gengoby, genasin, arith, marith, genextr

The generator routines occurring in Table V are classified according to the nature of the generative action they take. The "elementary" routines are those which are concerned neither with code generation nor with manipulation of entries referenced through the hash symbol table. The generator routines "corresponding to declaration statements" are concerned almost exclusively with manipulation of attribute entries referenced by name pointers in the hashed symbol table. These routines assign attributes by setting subfields of attribute entries; check on the consistency of name usage by examining the attribute entries attached to a given name; and, in a similar way, take account of the LITTLE name-scoping rules described below. The generator routines "corresponding to executable statements" also examine attribute entries and set relevant subfields of such entries. In addition, however, these generators or further subroutines called by them construct new entries in the VOA array which represent calls on machine-dependent code-selection routines. Note then that, as its name implies, the VOA array contains entries of two types: those representing variables and those representing operators. A new operation entry, when constructed by a generator subroutine corresponding to an executable statement, is placed at the top of the unused portion of the VOA stack, so that the order of operator type entries in the VOA stack also represents the order of the corresponding code fragments in the output code to be generated. Note also that the generator routines invoked during the parse of the component parts of the syntactic type ‹expr› (constituting expressions of quite general form) enter references to the entries which they construct into the hash array HA; using these entries, the same routines detect common subexpressions within basic program blocks and suppress logically redundant recalculations.

The occurrence within a LITTLE source program of either a label or a subroutine call terminates a basic code block within which formally redundant calculations are truly redundant and may be suppressed. For this reason, on encountering such a label we call a BLOCKEND

procedure, which assigns compiler-created "temporary" variables for storage of the result of each calculation in the compiler-generated macro-code stream, and which respecifies the inputs to a macro operation as a temporary variable rather than as the calculation whose result this temporary variable represents. This, on the one hand, reduces the logical context of each macro-operation in the code stream to something quite local, i.e., to a finite set of variables representing inputs, thereby transforming the code stream into a sequence of "productions" directly usable for final code generation. On the other hand, when proceeding in this way we break the logical links between an operation and the prior operations which prepare its inputs and make formal identities between distinct calculations unrecognizable. It is for the latter reason that temporary variable specification is postponed to the occurrence within our source-code of a block-terminating label or subroutine call. The BLOCKEND procedure also purges the hash-table HA of all entries representing operations rather than variables; this purging operation is done using an efficiently-aimed device implicit in the procedures described below.

Translation of LITTLE source-code into tabular macros representing eventual calls on code-selection routines continues in the manner described above until the occurrence within the source code of a FIN-statement terminating the source program to be translated, at which point code-selection action is commenced. The generated macros are processed, and the associated machine code produced. The code selection procedure is generally as follows. Each macro call in turn is examined by an ASSEMBLE routine which determines, by reference to the particular operation to be performed, and by reference to its input variables and their size, whether the operation is essentially elementary and to be performed by an open-macro-sequence of a few machine instructions, or whether the operation is non-elementary, in which case it is to be performed by a closed subroutine, written in LITTLE or directly in machine code, which performs the required complex operation by a programmed sequence of elementary steps.

BIBLIOGRAPHY

Adams, Charles W. and Laning, J. H., Jr. [1]
The M.I.T. System of Automatic Coding:
Comprehensive Summer Session and Algebraic,
in: Symposium on Automatic Programming for Digital Computers,
U. S. Dept. of Commerce.

Allard, Wolf, and Semlin [1]
Some effects of the 6600 Computer on Language Structures.
CACM 7 (2/) 112-118.

Allen, F. E. [1]
Program Optimization
IBM (Confidential) Research Paper RC-1599, Yorktown, 4/66.

Anderson, J. P. [1]
A note on some compiling algorithms
CACM 7 (3/64) 149.

Arden, B. W. et al. [1]
An algorithm for equivalence declarations.
Comm. ACM 4 (7/61) 310-314.

_____ [2]
An algorithm for translating Boolean expressions.
J. ACM 9 (1962) 222-239.

Arden, Galler, and Graham [1]
The internal organization of the MAD translator.
CACM 4 (1/) 28-31.

Arden, B. and Graham, R. [1]
On GAT and the construction of translators.
CACM 2 (7/59) 24-26. Correc., CACM 2 (11/59) 10-11.

Automatic Programming.
A Soviet algebraic language compiler.
in: Handbook of Automation, Computation and Control, V. 2.
ed. by E. M. Grabbe, Wiley, New York, 1959.

_____
The A-2 Compiler System.
Computers and Automation 4 (9/55) 25-31 and 4 (10/55) 15-23.

the IT Translator
in Handbook of Automation, Computation and Control, v. 2.
herausgegeben von E. M. Grabbe et al.
J. Wiley, New York 1959 2-200 - 2-228.

Backus, J. W. and Herrick, H. [1]
IBM 701 Speedcoding and other Automatic Programming Systems.
Symp. Automatic Programming Digital Computers, ONR 13-14 May 1954, PB 111 607, 106-113.

Backus, J. et al. [1]
The Fortran Automatic Coding System.
in S. Rosen op. cit. pp 29-47.

Banerji, R. [1]
Some studies in syntax-directed parsing.
in Computation in Linguistics (ed. by Garvin, P.) p. 76
1966 Indiana Univ. Press.

Barnett, M. [1]
Syntactic analysis by digital computer.
CACM 5 (10/62) 515-526.

Barron, D. W. et al. [1]
The main features of CPL.
Comp. J. 6 (2/63) 134-143.

Bastian, L. [1]
A phrase-structure language translator.
AFCRL Rep. 62-549, AF Cambridge Res. Labs., Bedford, 8/62.

Batson, A. [1]
The organization of symbol tables.
CACM 8 (2/65) 111.

Bauer, F. L. and Samelson, K. [1]
Maschinelle Verarbeitung von Programmsprachen.
in H. Hoffmann (ed.) Digitale Informationswandler,
Braunschweig: Vieweg Verlag, (1962) 227-268.

Bayer, R., Murphree, Jr., E., and Gries, D. [1]
User's manual for the ALCOR Illinois 7090 ALGOL-60 translator
2nd ed. U. of Illinois, Sept. 1964.

Belady, L. A. [1]
A study of replacement algorithms for a virtual-storage computer.
IBM Systems Journal 5 (2/66) 78.

Bennett, R. K. and Neumann, D. H. [1]
Extension of existing compilers by sophisticated use of macros.
CACM 7 (9/64) 541 [actually about assemblers]

Blair, C. R. [1]
A program for correcting spelling errors.
Information and Control (3/60) 60-67.

Bobrow, D. G. [1]
Syntactic analysis of English by a computer - a survey.
Proc. AFIPS Fall Joint Comp. Conf. 24 (1963) 365-387
Spartan Books.

Bobrow, D. G. and Weizenbaum, J. [1]
List processing and extension of language facility by embedding.
IEEE Trans. on Electronic Computers EC-13 (4/64) 395-400.

Bolduc, R. et al. [1]
Preliminary description of the TGS-II system.
Document CA-6408-0111, Computer Associates, Inc. Sept.'64.

Bottenbruch, H. [1]
Übersetzung von algorithmischen Formelsprachen in die
Programmsprachen von Rechenmaschinen.
Zeitschr. Math. Logik u. Grundl. Math. 4 (1958) 180-221.

Brooker, R. A. [1]
Top-to-bottom parsing rehabilitated?
CACM 10 (4/) 223-225.

Brooker, R. A. and Morris, D. [1]
An assembly program for a phrase structure language.
Comp. J. 3 (1960) 168-174.

_____ [2]
Some proposals for the realization of a certain assembly program.
Comp. J. 3 (1960) 220-231.

Brooker, R. A. and Morris, D. [3]
Trees and routines.
Comp. J. 5 (1962) 33-47.

_____ [4]
A general translation program for phrase structure languages.
J. ACM 9 (1/62) 1-10.

Brooker, R. A. and Rohl, J. S. [1]
Simply partitioned data structures: the compiler-compiler reexamined.
Machine Intelligence I, Collins and Michie (eds.), Oliver and Boyd, London (1967).

Brooker, R. A. et al. [1]
Compiler-compiler facilities in Atlas autocode.
Comp. J. 9 (1967) 350-352.

_____ [2]
Experience with the compiler compiler.
Comp. J. 9 (1967) 345-349.

_____ [3]
The compiler-compiler.
Annual Review in Automatic programming, 3 (1963) 229.

Buchholz, W. [1]
File organization and addressing.
IBM Sys. Jour. (6/63) 86.

Burkhardt, W. H. [1]
Universal programming languages and processors: a brief survey and new concepts.
FJCC (1965) 1.

Cheatham, T. E. Jr. (ed.) [1]
The introduction of definitional facilities into higher level languages.
Proc. FJCC (1966) 623-638

---------- [2]
Proceedings of a working conference on mechanical language structures.
CACM 7 (2/64), entire issue.

---------- [3]
The TGS-II translator generator system.
Proc. IFIP Congress (1965)
Spartan Books.

---------- [4]
The theory and construction of compilers.
Document CA-6606-0111, Computer Associates Inc. (6/66)

---------- [5]
The architecture of compilers
Document No. AD-64-2-R, Computer Assoc. Inc. Wakefield Mass.,64.

---------- [6]
Algol E-- an extendable version of Algol-60
In preparation (1966).

Cheatham, T. E. Jr. and Leonard, Gene F. [1]
Introduction to the CL-II programming system.
Document CA-63-7-SD, Computer Assoc. Inc. (11/63).

Cheatham, T. E. Jr. and Sattley, K. [1]
Syntax directed compiling.
Prod. SJCC (1964) 31-57; also Proc. AFIPS Spring Joing Comp. Conf. 25 (1964) 31-57, Baltimore: Spartan Books.

Cheatham, T. E. Jr. et al. [1]
Preliminary description of the translator generator system, II.
Document CA-64-1-SD, Computer Assoc. Inc. (1964).

Cheatham, T. E. Jr. [7]
The TGS-II translator-generator system.
Proc. IFIP Congress, N. Y. (1965) 592-593.

---------- [8]
The introduction of definitional facilities into higher level programming languages,
Proc. AFIPS (1966) FJCC 29 623-637.

Chomsky, N. [1]
On certain formal properties of grammars.
Inform. and Control 2 (2/59).

_____ [2]
Formal properties of grammars.
In Handbook of Math. Psych. II, R.D. Luce + R.R. Bush, eds.
Wiley, New York (1963).

Cohen, L. J. [1]
COMMEN: a new approach to programming languages.
AFIPS Conf. Proc. 30 (1967) 671-676
Spring Joint Computer Conference.

Conway,
Design of a separable translation-diagram-compiler.
CACM 6 (7/63) 396-408.

Conway, R. W. and Maxwell, W. L. [1]
CORC: the Cornell computing language.
CACM 6 (1963) 317.

Christensen, C. H. and Mitchell, R. W. [1]
Reference manual for the NICOL 2 programming language.
Computer Associates Inc., 1st ed., in preparation (1966).

Damereau, F. J. [1]
A technique for computer detection and correction
of spelling errors.
CACM 7 (1964) 171.

Dantzig, G. B. [1]
On the shortest route through a network.
Management Science (1960) 187.

Dantzig, G. B. and Reynolds, G. H. [1]
Optimal assignment of computer storage by chain
decomposition of partially ordered sets.
Univ. Calif., Berkeley Operations Research Centre Rep.
No. ORC-66-6 (3/66).

Dean, A. L. [1]
Some results in the area of syntax directed compilers.
Computer Assoc. Inc. Rep. CA-6412-0111 (12/64).

Dennis, and van Horne [1]
Programming semantics for multiprogrammed computations.
CACM 9 (3/) 143-155.

Dijkstra, E. W. [1]
ALGOL 60 translation.
Stichting Mathematisch Centrum, Amsterdam, Netherlands
ALGOL Bulletin Suppl. No. 10 (11/61).

———— [2]
Making a translator for ALGOL 60.
Annual Review in Automatic Programming, Pergamon Press,
The MacMillan Co. Inc. NYC Vol. 3 (1963) 347-356.

Earley, J. [1]
An efficient context-free parsing system.
Ph.D. Thesis (1968) Computer Science, Carnegie-Mellon, Pittsb.

———— [2]
An LR(K) parsing algorithm.
Carnegie Inst. Tech., Pittsb. Pa. (1967) (mimeo.).

———— [3]
An N^2 recognizer for context free grammars.
Dept. of Comput Sci. Report (1968) Carnegie-Mellon U.

———— [4]
Generating a recognizer for a BNF grammar.
Comp. Cr. Rep., Carnegie-Mellon U., Pittsb. (1965).

Eickel, J. [1]
Generation of parsing algorithms for Chomsky-2 type languages.
Math. Inst. Techn. Hochschule, Munich (1964).

Eichel; Paul; Bauer and Samuelson. [1]
A syntax-controlled generator of formal programming languages.
CACM 6 (8/) 451-455.

Englund, D. and Clark, E. [1]
The CLIP translator.
CACM 4 (1961) 19-22.

Ershov, A. P. [see also Yershov] [1]
ALPHA - an automatic programming system of high efficiency.
Jour. ACM 13 (1/66) 17-24.

——— [2]
On programming of arithmetic operations.
CACM 2 (8/58) 3-6.

Evans, A. Jr. [1]
An ALGOL 60 compiler.
Comp. Cr., Carnegie-Mellon U., Pittsburgh
Report No. CRO-4 (Aug. 27, 1963).
also Annual Review in Automatic Programming 4 (64) 87.

Evans, Perlis, and van Zoeren. [1]
The use of threaded lists in constructing a combined
ALGOL and machine like assembly processor.
CACM 4 (1/) 36-41.

Evey, J. [1]
The theory and application of pushdown store machines.
Math. Linguistics + Automatic Translation Rep. No. NSF-10,
Harvard Comp. Lab., Cambridge (1963).

Fabian, V. [1]
A recursive procedure for compiling expressions.
Chiffres 2 (4/63) 275-281.

Feldman, J. A. [1]
A formal semantics for computer languages and its
application in a compiler-compiler.
CACM 9 (1/66) 3-9.

——— [2]
A formal semantics for computer-oriented languages.
Carnegie-Mellon U. Report (1964).

Feldman, J. and Gries, D. [1]
Translator writing systems.
CACM 11 (2/68) 77-113.

Ferguson, H. E. and Berner, E. [1]
Debugging systems at the source language level.
CACM 6 (1963) 430.

Floyd, R. [1]
A descriptive language for symbolic manipulation.
J. ACM 8 (4) Oct. 61 pp. 579-584.

_____ [2]
Syntactic analysis and operator precedence.
J. ACM 10 (3) July 1963 pp. 316-333.

_____ [3]
Bounded context syntactic analysis.
CACM 7 (2/64) 62-65.

_____ [4]
The syntax of programming languages - a survey.
IEEE Trans. on Elect. Comp.v.EC-13, no. 4, (8/64).

_____ [5]
An algorithm for compiling efficient arithmetic operations.
CACM 4 (1/) 42-51.

Franciotti, R. G. and Lietzke, M. P. [1]
The organization of the SHARE ALGOL translator.
Proc. ACM 19th Natl. Conf. (1964) pp. D1.1.1-D1.1.10

Fredkin, [1]
Trie Memory.
CACM 4 ( ) 498-500.

Freeman, D. N. [1]
Error correction in CORC, the Cornell computing language.
AFIPS Conf. Proc. v. 26: 1964 Fall Joint Comp.Conf. pp 15-34.

Galler, B.A. and Fisher, M.J. [1]
An improved equivalence algorithm.
CACM 7 (5/64) 301-303.

Galler-Perlis [1]
A proposal for definitions in ALGOL.
CACM 10 (4/) 204-219.

Garwick, J.V. [1]
GARGOYLE, a language for compiler writing.
CACM 7 (1/64) 16.

_____ [2]
The definition of programming languages by their compilers.
In Formal Language Description Languages for Computer Programmang, ed. T.B. Steel Jr., North Holland, Amsterdam, 1966, p. 139.

Garwick, J.V. et al. [1]
The GPL language.
TER-05, Control Data, Palo Alto, California, 1966.

Gear, C.W. [1]
High speed compilation of efficient object code.
CACM 8 (8/65) 483-488.

_____ [2]
Optimization of the address field compilation in the ILLIAC II assembler.
Comput. Jour. 6 (Jan. 1964) 332.

Gilbert, P. [1]
On the syntax of algorithm languages.
JACM 13 (Jan. 1966) 90-107.

Gilbert, P., Hosler, J., and Schager, C. [1]
Automatic programming techniques.
TDR-62-632, Rome Air Development Center, Rome, N.Y., 1962.

Gilbert, P., and McLellan, W.G. [1]
Compiler generation using formal specification of procedure-oriented and machine languages.
Proc. AFIPS 1967 SJCC, Vol. 30, pp. 447-455.

Ginsburg, S. [1]
The mathematical theory of context free languages.
McGraw Hill, 1966.

Glennie, A. E. [1]
On the syntax machine and the construction of a universal compiler.
Carnegie Tech. Comp. Cr. Report No. 2 (7/60).

Goldfinger, R. [1]
New York University Compiler System.
Symposium Automatic Programming Digital Computers, ONR 13-14 May '54
PB 111 607, 30-33.

Graham, R. M. [1]
Bounded context translation.
Proc. SJCC (1964) 17-29.

——— [2]
Translation construction.
Notes of Summer Conf. Automatic Programming, U. of Mich.,6/63.

Grau, A.A. [1]
Recursive processes and ALGOL translation.
CACM 4 (1/) 10-15.

——— [2]
A translator-oriented symbolic programming language.
J.ACM 9 (4) (10/62) 480-487.

——— [3]
The structure of an ALGOL translator.
Oak Ridge Natl. Lab., Oak Ridge, Tenn.,Rep. ORNL-3054 (2/9/61).

Greibach, S. [1]
Formal parsing systems.
CACM 7 (8/) 499-504.

Grems, M. and Porter, R.E. [1]
A digest of the Boeing Airplane Co. algebraic interpretive
coding system.
Boeing Airplane Co., Seattle, Wash. (7/55).

Gries, D. [1]
The object program produced by the ALCOR Illinois 7090 compiler.
Rep. No. 6412, Rechenzentrum der Tech. Hochsch., München, 1964.

Gries, D. et al. [1]
Internal notes on the compiler writing system.
Computer Sci. Dept., Stanford Univ., 1967.

———— [2]
Some techniques used in the ALCOR-Illinois 7000.
CACM 8 (8/65) 496-500.

———— [3]
The use of transition matrices in compiling.
Tech. Rep. CS-57, Comp. Sci. Dept., Stanford U. (3/67);
also in: CACM 11 (1.68) 26-34.

Gries, D., Paul, M. and Wiehle, H. R. [2]
ALCOR Illinois 7090 - An ALGOL compiler for the IBM 7090.
Rep. No. 6415, Techenzentrum der Tech. Hochsch., München, '64.

———— [1]
Some techniques used in the ALCOR Illinois 7090.
CACM 8 (8/65) 496-501.

Griffiths, T. V. [1]
An upper bound on the time required for parsing by
predictive analysis with abortive path elimination.
Fourth Annual Meeting of the Assoc. for Machine Translation
and Computational Linguistics, UCLA, July 26-27, 1966.

Griffiths, and Petrick, [1]
On the relative efficiencies of content-free grammar recognizers.
CACM 8 (5/) 289.

Haines, L. H. [1]
Serial compilation and the 1401 Fortran compiler.
IBM Sys. Jour. 4 (1) 1965 73-80.

Halpern, M.I. [1]
XPOP: a meta-language without metaphysics.
Proc. FJCC 1964 57-68.

———— [2]
A manual of the XPOP programming system.
Electronic Sci. Lab. Lockheed Missiles and Space. Co.,
Palo Alto, Calif., (3/67).

———— [3]
Toward a general processor for programming languages.
CACM 11 (1/68) 15-26.

Hamblin, C. L. [1]
Translation to and from Polish notation.
Comp. J. 5, 3, 1962 210-213.

Hartman, P. H. and Owens, D.H. [1]
How to write software specifications.
AFIPS Conf. Proc. 31 (1967) 779-790
Fall Joint Computer Conference.

Hartmanis, J. and Stearns, R.E. [1]
Computational complexity of recurring sequences.
Proc. Fifth Ann. Symp. Switching Circuit Theory and
Logic Design 82-90. (1964).

Hawkins, E.N. and Huxtable,D.H.R.[1]
A multi-pass translation scheme for Algol 60.
Annual Review in Automatic Programming 3 (1963) 163.

Hays, D. [1]
Automatic language-data processing. in
Computer Applications in the Behavioral Sciences,
H. Borko, ed. Prentice Hall, Englewood, N.J. (1962).

Hendry, D. [1]
Provisional BCL manual.
Inst. of Comp. Sci., U. of London (1967).

Hennie, F.C. [1]
On-line Turing machine computations.
IEEE Trans. Electron. Comput. EC-15, (1966) 35-44.

Hill, U., Langmaack, H., Schwarz, H.R. and Seegmuller, G. [1]
Efficient handling of subscribed variables in ALGOL 60 compilers.
Proc. Symp. Symbolic Languages in Data Processing,
Rome, (1962) 331-340 Gordon and Breach, N.Y.

Hopgood, F.R.A. [1]
Compiling techniques.
American Elserier, N.Y. (1969).

———— [2]
A solution to the table overflow problem for hash tables.
Computer Bulletin 11 (3/68) 297.

Hopgood, F.R.A. and Bell, A.G. [1]
The Atlas Algol preprocessor for non-standard dialects.
Comp. J. 9 (2/67) 360.

Horwitz, L.P. et al. [1]
Index register allocation.
J. ACM 13 (1/66) 43.

Huskey, H. D.
Compiling techniques for algebraic expressions.
Comp. J. 4 (1/61) 10-19.

Huskey, H.D. and Wattenberg, [1]
A basic compiler for arithmetic expressions.
CACM 4 (1/) 3-9.

_____ [2] ???
Compiling techniques for Boolean expressions and
conditional statements in Algol 60.
CACM 4 (1/) 70-75.

Huxtable, D.H.R. [1]
On writing an optimizing translator for ALGOL 60.
in Introduction to System Programming, P. Wegner (ed.)
Academic Press, (1964) 137.

Huxtable, D.H.R. and Hawkins,E.N. [1]
A multi-pass translation scheme for Algol 60.
Annual Review of Automatic Programming 3

IBM Systems Manual for 704 FORTRAN and 709 FORTRAN. [1]
Appl. Programming Dept., IBM (4/60).

Iliffe, J.K. and Jodeit, J.G. [1]
Dynamic storage allocation.
Comp. J. 5 (10/62) 200.

Ingerman, P.Z. [1]
A syntax-oriented translator.
Academic Press (1966, rev. 1967), New York.

_____ [2]
A syntax oriented compiler...
Moore School of Elec. Engr., U. of Engr., U. of Pa., Phila., 4/63.

———————— [3]
A translation technique for languages whose syntax is expressible in extended Backus normal form.
Proc. Intl. Symp. Symbolic Lang. in Data Proces., Rome, Italy, (1962) 741-758 Gordon and Breach, New York.

———————— [4]
Thunks - a way of compiling procedure statements with some comments on procedure declarations.
CACM 4 (1/) 55-58.

Irons, E.T. [1]
A syntax-directed compiler for Algol-60.
CACM 4 (1/61) 51-55.

———————— [2]
The structure and use of the syntax-directed compiler.
Annual Review in Automatic Programming,
Pergamon Press 3 (1963) 207-228.

———————— [3]
An error correcting parse algorithm.
CACM 6 (11/63) 669-674.

———————— [4]
Structural connections. in
Formal Languages
CACM 7 (2/) 67-70.

———————— [5] [4 on card?]
PSYCO, the Princeton syntax compiler.
Inst. for Defense Analysis, Princeton, N.J. IDA Repoint, 1/61.

———————— [6] [5?]
The structure and use of the syntax directed compiler.
in Annual Review in Automatic Programming,
Pergamon Press, MacMillan Co., N.Y. 3 (1963) 207-227.

———————— [7] [6?
Towards more versatile mechanical translators.
AMS Symp. Appl. Math. 15 (1963) 41-50.

Irons, and Feuerzeig, [1]
Comments on the implementation of recursive procedures and blocks. in ALGOL 60.
CACM 4 (1/) 65-69.

Iverson, K. E. [1]
Formalism in programming languages.
IBM Research paper RC-992 (8/63).

_____ [2]
A programming language.
John Wiley, (1962) 144.

Jones, R. W. [1]
Generalized translation of programming languages.
AFIPS Conf. Proc. 31 (1967) 570-580 Fall Joint Comp. Conf.

Kanner, H. [1]
An algebraic translator.
CACM 2 (10/) 19-22.

Kanner, H. et al. [1]
The structure of yet another Algol compiler.
CACM 8 (7/65) 427-438.

Kasami, T. [1]
An efficient recognition and syntax analysis algorithm for context free languages.
U. of Ill. (1966).

_____ [2]
A note on computing time for recognition of languages generated by linear grammars.
Information and Control 10 (1967) 209-214.

Kasami, T. and Toril, K. [1]
Some results on syntax analysis of context free languages.
Record of Technical Group on Automata Theory of Inst. of Electronic Communication Engrs., Japan (1/67).

Keese, W. M. and Huskey, H.D. [1]
An algorithm for the translation of ALGOL statements.
Information Processing (1962) 498-500.

Kerr, R.H. and Clegg, J. [1]
The Atlas ALGOL compiler - an ICT implementation of ALGOL using the Brooker-Morris syntax-directed compiler.
Comp. J. (1967).

Knuth, D. E. [1]
Runcible - algebraic translation on a limited computer.
CACM 2 (12/59) 18-20.

---------- [2]
History of writing compilers.
Proc. ACM 17 Natl. Conf. (1962) 43, 126.

---------- [3]
On the translation of languages from left to right.
Inf. Contr. 8 (10/65) 607-639.

---------- [4]
PRUNCIBLE - algebraic translation on a limited computer.
CACM 2 (11/59) 18.

Korenjak, A. J. [1]
A practical approach to the construction of deterministic language processors.
RCA Labs., Princeton, N. J.

Kuno, S. [1]
The predictive analyzer and a path elimination technique.
CACM 8 453-463.

Kuno, S. and Oettinger, A.G. [1]
Multiple path syntactic analyzer.
Information Processing 62, Popplewell, C.M. (ed.)
North-Holland, Amsterdam (1962-63).

---------- [2]
Multiple path syntat

Landin, P. J. [1]
A formal description of ALGOL 60.
in Formal Language Description Languages for Computer Programming, Steel, Jr., T.B. (ed.)
North-Holland, Amsterdam (1966) 266.

Leavenworth, P. M. [1]
Syntax macros and extended translation.
CACM 9 (11/65) 790-793.

_____ [2]
Fortran IV as a syntax language.
CACM 7 (2/) 72-79.

Ledley, R.S. and Wilson, J.B. [1]
Automatic-programming-language translation through syntactical analysis.
CACM 5 (3/62) 145-155.

Leroy, H. [1]
A macro-generator for ALGOL.
AFIPS Conf. Proc. 30 (1967) 663-669 Sprint Joint Comp. Conf.

Lewis, P.M., Hartmanis, J. and Stearns, R.E. [1]
Memory bounds for recognition of context-free and context-sensitive languages.
IEEE Conf. Record on Switching Circuit Theory and Logical Design. IEEE Publ. 16C13 (1965) 191-202.

Lietzke, M. P. [1]
A method of syntax-checking ALGOL 60.
CACM 7 (8/64) 475-478.

Lin, C. L., Chang, G. D. and Marks, R. E. [1] and [2]
The design and implementation of a table driven compiler system.
AFIPS conference Proc. 30 (1967) 691-697 Sprint Joint Comp. Conf.
and Project MAC Report (1967).

Lucas, P. [1]
The structure of formula-translators.
Elektr. Rechen. 3 (8/61) 159-166;
Mailufterl, Vienna, Austria, Algol Bull. Suppl. 16 (9/61).

Luccio, F. [1]
A comment on index register allocation.
CACM 10 (9/67) 572-574.

McCarthy, J. [1]
Recursive functions of symbolic expressions and their computation by machine, Part I.
CACM 3 (4/60) 184-195.

——————— [2]
A formal description of a subset of ALGOL.in
Formal Language Description Languages for Computer Programming, Steel Jr., T.B. (ed.) North-Holland, Amsterdam (1966) 1.

McClure, R. M. [1]
TMG - a syntax directed compiler.
Proc. ACM Natl. Conf. (1965) 262-274.

McIlroy, M. D. [1]
Macro instruction extensions of compiler languages.
CACM 3 (4/60) 214-220.

McKeeman, W. M. [1]
Peephole optimization.
CACM 8 (7/65) 443-444.

——————— [2]
An approach to computer language design.
Tech. Rpt. CS 48 Comp. Sci. Dept., Stanford U. (8/66)

McKeeman, Horning, and Wortman [1]
A compiler-generator.
Comp. Sci. Dept., Stanford U., manuscript (to be publ.) 1970.

——————— [2]
The XPL compiler generator system.
Proc. Fall Joint Comp. Conf. (1968).

Martin, W. A. [1]
A left to right then right to left parsing algorithm.
Project MAC, M.I.T. (2/68).

Masterson, K. S. [1]
Compilation for two computers with NELIAC.
CACM 3 (11/60) 607-611.

Maurer, W. P. [1]
An improved hash code for scatter storage.
CACM 11 (1/68) 35.

Mealey, G. H. [1]
A generalized assembly system.
Rand Corp. RM-3646-PP (8/63).

Medlock, C.W. and Lowry, E.S. [1]
Global program optimization.
IBM (Confidential) TR 00.1330 (9/65).

_____ [2]
Object code optimization.
CACM 12 (1/69) 13-22.

Mehlan, W. [1]
A description of a cooperative venture in the production of an automatic coding system.
J. ACM 3 (1956) 266-271.

Mendocino, S.F. and Zwickenberg, R.F. [1]
A FORTRAN code optimizer for the CDC 6600.
Tech. Rep. (4/65).

Metcalfe, H. [1]
A parameterized compiler based on mechanical linguistics.
ACM Natl. Conf. (1963) Denver, Colorado
and Annual Review in Automatic Programming 4 (1964) 125-165.

Morris, R. [1]
Scatter storage techniques.
CACM 11 (1/68) 38.

Nakata, [1]
On compiling expressions for arithmetic expressions.
CACM 10 (8/)

Napper, R.B.E. [1]
The third-order compiler. A context for free man-machine communication.
Machine Intelligence I, Collins and Michie (eds.)
Oliver and Boyd, London (1967).

Naur, P. (ed.) [1]
Report on the algorithmic language ALGOL 60.
Numer. Math. 2 (1960) 106-136
and CACM 3 (5/60) 299-314.

——————— [2]
Revised report on the algorithmic language ALGOL 60.
CACM 6 (1/63) 1-17;
Numer. Math. 4 (1963) 420-452;
Comp. J. 5 (1963) 349-367.

——————— [3] and [4]
The design of the GEIR ALGOL compiler.
Nordisk Tidskrift for Information Behandling, Pt. I 3('63)124.
and Annual Review in Automatic Programming 4 (1964).

Nelson, H. L. [1]
Program optimizing techniques for the CDC 6600 central processor.
Tech. Rep. (4/6/65)

Nievergelt, J. [1]
On the automatic simplification of computer programs.
CACM 6 (6/65) 366-370.

Northcote, R.S. and Steiner, A.J.[1]
A syntax loader and subscanner for a compiler-compiler.
U. Ill. Digital Comp. Lab. Rep. No. COO-1018-1031 (1964).

Oettinger, A. G. [1]
Automatic syntax analysis and the pushdown store.
Proc. Symp. Appl. Math. 12 (1961)
Amer. Math. Soc., Providence, R.I.

Olper, A. [1]
"Tool" - a processor construction language.
Proc. IFIP Congress, Munich (1962) 513-514.

Paul, M. [1]
A general processor for certain formal languages.
Proc. Symp. Symb. Lang. in Data Proces. (1962) Rome
Gordon and Breach, N.Y.

———————— [2]
ALGOL 60 processors and a processor generator.
Proc. IFIP Congress (1962) 493-497
North-Holland, Amsterdam
and Munich.

Perlis, A. J. (ed.) [1]
Papers presented at the ACM compiler symposium, Nov. 1960.
CACM 4 (1/61) entire issue.

Perlis, A.J. and Samelson, K. [1]
Preliminary report - Intl. algebraic language.
CACM 1 (12/58) 8-22.

———————— [2]
Report on the algorithmic language ALGOL, etc.
Num. Math. 1 (1959) 41-60.

Perlis, A.J. and Smith, J.W. [1]
A mathematical language compiler.
Automatic Coding, Jour. Franklin Inst., Monograph 3, Phila. Pa. (4/57) 87-102.

Perlis, A.J., Smith, J.W. and van Zoeren, H.R. [1]
Internal translator (IT), a compiler for the 650.
Carnegie Inst. Tech. Comp. Cr., Pittsburgh (1956). Reproduced by Lincoln Lab. Div. 6, Document 6D-327.

Perlis, A.J. and Thornton, C. [1]
Symbol manipulation by threaded lists.
CACM 3 (4/60) 195-204.

Peterson, W. W. [1]
Addressing for random-access storage.
IBM Jour. Res. Devel. 1 (1957) 130.

Plaskow, J. and Schuman, S. [1]
The TRANGEN system on the M460 computer.
Comp. Assoc. Inc. Report (7/66).

Plath, W. [1]
Mathematical linguistics. in
Trends in European and Amer. Linguistics 1930-1960, Spectrum Publ., Utrecht, Neth. (1961) 21-57.

Pratt, T. W. [1]
Syntax-directed translation for experimental programming languages.
TNN-41 Comp. Cr., U. Texas, Austin (1965).

Pratt, T.W. and Lindsay, R.K. [1]
A processor-building system for experimental programming languages.
Proc. AFIPS 1966 FJCC 29 (1966) 613-621.

Programmers Reference Manual FORTRAN. [1]
IBM Corp., N.Y. (1956).

Prosser, R. [1]
Applications of Boolean matrices to the analysis of flow diagrams.
Proc. Eastern Joint Comp. Conf. (1959)
available from Spartan Books, Wash., D. C.

Pyle, I. C. [1]
Implementation of FORTRAN on Atlas.
Introduction to System Programming, Wegner, P. (ed.) 1964 86-100.

Randell, B. and Russell, L.J. [1]
ALGOL 60 implementation.
Academic Press, N. Y. 1964.

——— [2]
Single-scan techniques for the translation of arithmetic expressions in ALGOL 60.
J. ACM 11 (2) (4/64) 159-167.

Reynolds, J. C. [1]
A compiler and generalized translator.
Appl. Math. Div., Argonne Natl. Lab., Argonne, Ill. (undated).

——— [2]
An introduction to the COGENT programming system.
Proc. ACM Natl. Conf. (1965) 422.

Rosen, S. (ed.) [1]
Programming systems and languages.
McGraw-Hill, N. Y., 1967.

———— [2]
A compiler-building system developed by Brooker and Morris.
in S. Rosen, op. cit. 306-331.

———— [3]
Programming systems and languages: a historical survey.
Proc. SJCC (1964) 1-16.

———— [4]
PUFFT - the Purdue University fast FORTRAN translator.
CACM 8 (11/65) 661-665.

Ross, D. T. [1]
A generalized technique for symbol manipulation and numerical calculation.
CACM 4 (3/61) 147.

———— [2]
An algorithmic theory of language.
Electronic Systems Lab., MIT, Cambridge, Mass.
Rep. No. ESL-TM-156 (11/62).

———— [3]
On context and ambiguity in parsing.
CACM 7 (2/64) 131-133.

———— [4]
The AED free storage package.
CACM 10 (8/67) 481.

Ross, D. T. and Rodriguez, J.E. [1]
Theoretical foundations for the computer-aided design project.
SJCC (1963)

Rutishauser, H. [1]
Über automatische Rechenplanfertigung bei programmgesteuerten Rechenanlagen.
Z. Angew. Math. Mech. 31 (1951) 255.

———— [2]
Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen.
Mitt. Inst. f. Angew. Math. der ETH Zurich, Nr. 3 (1952) and Verlag Berkhäuser, Basel, (1952).

———— [3]
Some programming techniques for the ERMETH.
Jour. ACM 2 (1/55) 1-4.

Ryan, J. L., Crandall, R.L., and Medvedeff, M. [1]
A conversational system for incremental compilation and execution in a time-sharing environment.
AFIPS Conf. Proc. 29 (1966) 1-23 Fall Joint Comp. Conf.

Samuelson, K. et al. [1]
A syntax controlled generator of formal language processors.
CACM 6 (8/63) 451-455.

———— [2]
Programming languages and their processing.
Proc. IFIP Congress (1962) 487-492.

———— [3]
Probleme der Programmierungstechnik.
Inter. Kolloquium über Probleme der Rechentechnik, Dresden (1955) 61-68.

Samelson, K. and Bauer, F.L. [1]
Sequential formula translation.
in S. Rosen op. cit. 206-220.

———— [2]
The ALCOR project.
Proc. Symp. Symbolic Langs. in Data Proces. (1962) 207-217, Rome.
Gordon and Breach, N. Y. 1962.

———— [3]
Sequentielle Fromelübersetzung.
Elektron. Rechenanl. 1 (1959)

———— [4]
Programming languages and their processing.
Proc. IFIP Congress, Munich (1962) 487-492.

Savitt, D. A., Love, H. H. and Troop, R.E. [1]
ASP: a new concept in language and machine organization.
AFIPS Conf. Proc. 30 (1967) 87-102 Spring Joint Comp. Conf.

———— [2]
Association-storing processor study.
Tech. Rep. RADC-TR66-174, AD488 S-38
Defense Documentation Center, June 1966.

Shantz, P. et al. [1]
WHATFOR - the University of Waterloo FORTRAN IV.
CACM 10 (1/67) 41-44.

Shapiro, R.M. and Zand, L. [1]
A description of the input language for the compiler generator system.
Comp. Assoc. Inc. CAD-63-1-SD (6/63) [and CA-6306-0112 ?]

Schay, G. and Spruth, W. G. [1]
Analysis of a file addressing method.
CACM 5 (8/62) 459.

Schmidt, L. [1]
Implementation of a symbol manipulator for heuristic translation.
ACM Natl. Conf. paper (1963) Denver, Colorado

Schneider, F. W. and Johnson, G.D. [1]
A syntax-directed compiler-writing compiler to generate efficient code.
Proc. ACM 19 Natl. Conf. (1964) pp. D1.5.1-D1.5.8.

———— [2]
META-3; a syntax-directed compiler writing compiler to generate efficient code.
Proc. ACM 19th Natl. Conf. (1964) D1.5-1.

Schorr, H. [1]
Analytic differentiation using a syntax directed compiler.
Comp. J. 7 (1/65) 290-298.

Schorre, O. V. [1]
Meta II, a syntax oriented compiler writing language.
Proc. ACM 19th Natl. Conf., Phila (8/64) ACM Publ., p. 64.

Schützenberger, M. P. [1]
Context-free languages and pushdown automata.
Inf. Control 6 (9/63) 246-264.

Sheridan, P. B. [1]
The arithmetic translator-compiler of the IBM FORTRAN automatic coding system.
CACM 2 (2/59) 9-21.

Sherry, M. [1]
Syntactic analysis in automatic translation.
AFCRL-TR-61-100 Air Force Cambridge Res. Labs., Bedford,Mass.'61.

Sibley, [1]
The SLANG system.
CACM 4 (1/) 75-84.

Squires, B. E. [1]
Lexical analysis by a precedence grammar.
Univ. Ill. Dept. Comp. Sci. Report (1966).

Steel, Jr., T. B. [1]
A formalization of semantics for programming language description. in
Formal Lang. Descrip. Langs. for Comp. Program.
Steel, T.B. Jr. (ed.) North-Holland, Amsterdam (1966) 25.

Stone [1]
One-pass compilation of arithmetic expressions for a parallel processor.
CACM 10 (4/) 220-223.

Strachey, C. [1]
Towards a formal semantics.
IFIP Working Conf. Formal Lang. Descrip. Langs.,Vienna '64.
and in Formal Lang. Descrip. Langs. for Comp. Prog.
(ed.) T.B. Steel Jr., North-Holland, Amsterdam (1966) p. 198.

——— [2]
A general purpose macrogenerator.
Comp. J. (10/65) 225-241.

Strong, J. et al. [1]
The problem of programming communications with changing machines.
CACM 1 (8,9/58) 12-18, 9-15.

Tabory, R. [1]
Survey and evaluation of AEC system at MIT.
IBM Rep. No. TR 00.1383 (2/66).

Taylor, A. [1]
The FLOW-MATIC and MATH-MATIC automatic programming systems.
Annual Review in Automatic Programming, 1 (1960) 196-206
(R. Goodman, ed.) Pergamon Press, N. Y.

Tesler, L. G. and Enea, H. J. [1]
A language design for concurrent processes.
AFIPS Conf. Proc. 32 (1968) 403-408 Spring Joint Comp. Conf.

Thorlin, J. F. [1]
Code generation for PIE (parallel instruction execution) computers.
AFIPS Conf. Proc. 30 (1967) 641-642 Spring Joint Comp. Conf.

Torii, K., Kasami, T. and Ozaki, H. [1]
Recognition and syntax analysis of context-free languages.
Report of Inst. Elec. Communication Engrs., Japan (6/66)
in Japanese.

Trout, R. G. [1]
A compiler-compiler system.
Proc. ACM 22nd Natl. Conf. (1967) 317-322.

van Wijngaarden, A. [1]
Recursive definition of syntax and semantics. in
Formal Lang. Descrip. Langs. for Comp. Programming,
Steel Jr., T. B. (ed.) North-Holland, Amsterdam (1966) 13.

Waldburger, H. [1]
Gebrauchsanleitung für die ERMETH.
Inst. for angew. Mathematik der ETH. Zürich 1959.

Warshall, S. [1]
A syntax-directed generator.
Proc. FJCC (1961) 295-305.

Warshall, S. and Shapiro, R. M. [1]
A general-purpose table-driven compiler.
Proc. AFIPS 1964 SJCC 25 pp. 59-65.

Wegner, P. [1]
An introduction to stack compilation techniques.
in Introduction to System Programming, Wegner, P. (ed.)
(1964) 101-121

——— [2]
Introduction to systems programming.
Academic Press, 1962.

Willett, H. and Helwig, F. [1]
Syntax processor.
MITRE Working Pap. W-4677, Bedford, Mass. (1/62)

Wilkes, M.V. and Strachey, C. [1]
Some proposals for improving the efficiency of ALGOL 60.
CACM 4 (1/61) 488-491.

Wirth, N. [1]
A generalization of ALGOL.
CACM 6 (9/63) 547-554.

Wirth, N. and Weber, H. [1]
EULER: a generalization of ALGOL (Parts I, II)
CACM 9 (1,2/66) 13-25, 89-99.

Yamada, H. [1]
Real-time computation and recursive functions not real-time computable.
IRE Trans. Electron. Comp. EC-11,(1962) 753-760.

Yershov, A. P. (see also Ershov) [1]
ALPHA - an automatic programming system of high efficiency.
J. ACM 13 (1/66) 17-24.

——— [2]
Programming program for the BESM computer.
Pergamon Press, N. Y. 1959.

——— [3]
On operator algorithms (Russian).
Doklady Akademii Nauk USSR, N.S. 122 (6) 967-970.

Yershov, A.P., Kozhukhin, G.I. and Voloshin, U.M. [1]
Input language for automatic programming systems.
R.W. Hockney (trans.) N.Y. and London: Academic Press (1963).

Younger, D. H. [1]
Context free language processing in time n^3.
G.E. Res. Develop. Cr., Schenectady, N.Y. 1966.

_____ [2]
Recognition and parsing of context free languages in time n^3.
Info. and Contr. 10 (1967) 189-208.

Zand, L. J. [1]
A description of the input language for the compiler generator system, Vol. 2: MDL.
Comp. Assoc. Inc. Rep. No. CA-6404-0114 (4/64).

Zemanek, H. [1]
Die algorithmische Formelsprache ALGOL.
Elektr. Rechen. 1 (1959) 72-79, 140-143.

Alber, K. and Oliva, P. [1]
Translation of PL/1 into abstract syntax.
PL/1 Tech. Rep. TR25.086 (6/28/68) Uld Version II, IBM Lab. Vienna.

Alber, K., Oliva, P. and Urschler, G. [1]
Concrete syntax of PL/1.
PL/1 Tech. Rep. TR 25.084 (6/28/68) Uld Vers., IBM Lab.Vienna

Bandet, K. [1]
On the formal definition of PL/1.
AFIPS Conf. Proc. 32 (1968) 363-374 Spring Joint Comp. Conf.

Bandat, R. S. and Wilkins, R. L. [1]
An experimental general purpose compiler.
AFIPS Conf. Proc. 30 (1967) 457-462 Spring Joint Comp. Conf.

Bemer, R. W. [1]
PRINT 1 - an automatic coding system for the IBM 705.
Automatic Coding, Jour. Franklin Inst.,
Monograph No. 3, Phila. Pa. (4/57) 29-36.

Constantine, L. L. [1]
Control of sequence and parallelism in modular programs.
AFIPS Conf. Proc. 32 (1968) 409-414 Spring Joint Comp. Conf.

Donovan, J.J. and Ledgard, H.F. [1]
A formal system for the specification of the syntax
and translation of programming languages.
AFIPS Conf. Proc. 31 (1967) 553-569 Fall Joint Comp. Conf.

Elgot, C. C. and Robinson, A. [1]
Random access stored program machines. An approach to
programming languages.
J. ACM. 11 (4) 64, 365-399.

Fleck, M. and Neuhold, E. [1]
Formal definition of the PL/1 compile time facilities.
PL/1 Tech. Rep., Uld Version II, TR 25.080 (6/28/68)
IBM Lab. of Vienna.

Lucas, P. [1]
Two constructive realizations of the block concept and
their equivalence.
Tech. Rep. TR 25.085 (6/28/68), IBM Lab., Vienna.

Lucas, P., Lauer, P. and Stigleitner, H. [1]
Method and notation for the formal definition of programming languages.
PL/1 Tech. Report, Uld Version II, TR 25.087 (6/28/68) IBM Lab. Vienna.

Lucas, P., Alber, K., Bandat, K., Bekic, H., Oliva, P., Walk, K. and Zeisel, G. [1]
Informal introduction to the abstract syntax and interpretation of PL/1.
PL/1, TR 25.083, (6/28/68), Uld Version II, IBM Lab. Vienna.

Walk, K., Alber, K., Baudat, K., Bekic, H., Chroust, G, Kudielka, V., Oliva, P. and Zeilsel, G. [1]
Abstract syntax and interpretation of PL/1.
PL/1 Tech. Rep. Uld Vers. II TR 25.082 (6/28/68) IBM Lab Vienna

Appendix I

A BIBLIOGRAPHY OF FORMAL LANGUAGE THEORY
Draft I[1]

by
D. Wood[+]

Introduction.

This bibliography has been prepared as a basis for a "Notes and Comments" section within the book, "Programming Languages and Their Compilers" by J. Cocke and J.T. Schwartz. It must also be understood that this is Draft I of the bibliography, its most notable omissions being work carried out after 1968.

The sources are:

S. Ginsburg, "The Mathematical Theory of Context-Free Languages"

J. Hopcroft and J. Ullman, "Formal Languages and Their Relation to Automata"

J. Earley, "An Efficient CF Parsing Algorithm"

M. Fischer, "Grammars With Macro-like Productions"

Information and Control, 1965-1968

Journal of the ACM, 1966-1968, and miscellaneous reprints.

---

[1]Prepared at the Courant Institute under the National Science Foundation, Grant NSF-GJ-95.

[+]Present address: Courant Institute, 251 Mercer Street, NY, NY 10012

Aanderaa, S. and Fischer, P.C., 1967

The solvability of the halting problem for 2-state post machines

JACM 14 1,pp 677.

Aho, A.V., 1967

Indexed grammars--an extension of context-free grammars

Doctoral dissertation, Princeton Univ

excerpts in IEE Conf. Record 1967, 8th Ann. Symp. on Switching + Automata Theory, 10, pp. 21-31.

Aho, A.V., 1968

Indexed grammars - an extension of context-free grammars

JACM 15 1, pp. 647.

Aho, A.V., Hopcroft, J.E. and Ullman, J.D., 1968

On the computational power of pushdown store systems.

Inf. and control, to appear.

from Formal Languages and their Relation to Automata by J. Hopcroft, J. Ullman, Addison-Wesley 1969.

Altman, E.B., 1964

The concept of finite representability

Systems Research Cr. Rep SRC 56-A-64-20, Case Instit. Tech.

from biblio. Ginsburg 66, op. cit.

Altman, E.B. and Banerji, R.B., 1965.

Some problems of finite representability.

Inf. and Control 8, pp. 251-263.

from biblio. Ginsburg, 66.

Amar, V. and Putzolu, G., 1964.

On a family of linear grammars.

Inf. and Control 7, pp. 283-291.

from biblio. Ginsburg 66.

Amar, V. and Putzolu, G., 1965.

Generalizations of regular events.

Inf. and Control 8, pp. 56-63.

from biblio. Ginsburg.

American Standards Association Sectional Committee X3, 1964-65.
Fortran vs. Basic Fortran.
CACM 7, pp. 591-625. Appendices in 8, pp. 287-288

Arden, B., Galler, B. and Graham, R. 1963
The Michigan Algorithm Decoder.
Programmer's Manual, Univ. of Mich. Comp. Cr., Ann Arbor.

Axt, P. 1959.
On a subrecursive hierarchy and primitive recursive degrees.
Trans. Am. Math. Soc. 92, pp. 85-105.

Backus, J. W. 1959-60
The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference.
ICIP Proc., Paris, London: Butterworth's, pp. 125-132.

Backus, J. W. et al. 1960
Report on the algorithmic language ALGOL 60.
Num. Math. 2 pp. 106-136.

Baer, R. M. 1967
Certain direct Post systems and automata.
Zeitschrift für Math. Logik und Grundlagen der Math.,
Band 13, Heft 2, 151-174.

Banerji, R. B. 1963
Phrase structure languages, finite machines and channel capacity,
Inf. and Control 6, pp. 153-162.

Bar-Hillel, Y. and Shamir, E. 1959
Finite-state languages.
Jerusalem: Hebrew Univ.

Bar-Hillel, Y., Perles, M. and Shamir, E. 1961
On formal properties of simple phrase structure grammars.
Z. Phonetik, Sprachwiss. Kommunikationsforsch 14, pp. 143- 172.
also in: Bar-Hillel, Ch. 9, Language + Information, 1964.
Addison Wesley Publ. Co., Reading, Mass.

Bar-Hillel, Y., Perles, M. and Shamir, E. 1960.
On formal properties of simple phrase-structure grammars.
Tech. Rpt. No. 4, Hebrew Univ., Jerusalem.
also in: Zeit. für Phonetik, Sprachwissenschaft, und Kommunikationsforschung, 1961 14, pp. 143-172.

Bastian, A. L. 1962
A phrase-structure language translator.
Report No. 69-549, AF Cambr. Res. Labs., L. G. Hanscom Field, Bedford, Mass.

Beckenbach, E. F., Drooyan, I. and Wooton, W. 1964
College Algebra
Wadsworth Publ. Co.: Belmont, Calif.

Bellert, Irena 1965
Relational phrase structure grammar and its tentative application
Inf. and Control 8, p. 503.

Blum, M. 1964
A machine-independent theory of recursive functions.
Doctoral Thesis, M.I.T., Cambridge, Mass.
JACM 14 2, pp. 322-336.

Bobrow, D. G. 1963
Syntactic analysis of English by computer -- a survey.
Proc. AFIPS Fall Joint Comp. Conf. 24, pp. 365-387
Spartan Books, Baltimore.

Böhm, C. 1964
On a family of Turing machines and the related programming language.
ICC Bull. 3, 3, pp. 185-194.

Bohm, C. and Jacopini, H. G. 1964
Machines and languages with only two formation rules.
Proc. 1964 Intl. Coll. Algebra Ling. + Automata Theory, Jerusalem, in press.

Böhm, C. and Gross, W. 1964
Introduction to the CUCH.
Proc. Intl. Summer Sch. on Automata Theory, Ravello, in press.

Booth, T. L. 1967
Sequential machines and automata theory.
Wiley, New York.

Brainerd, Barron 1967-8
An analog of a theorem about context-free languages.
Inf. and Control 11, p. 561.

Brooker, R. A. and Morris, D. 1962
A general translator program for phrase structure languages.
JACM 9, 1, pp. 1-10.

Brooker, R. A., MacCallum, I. R., Morris, D. and Rohl, J.S. 1963
The compiler compiler.
in R. Goodman (ed.) Annual Review in Automatic Programming
Vol. 3, N.Y. and London: Pergamon Press, pp. 229-271.

Burge, W. H. 1964
Interpretation, stacks and evaluation.
in P. Wegner (ed.), Intro. to Sys. Prog.
N.Y. and London: Academic Press, pp. 294-312.

Burks, A. W., Warren, D. W. and Wright, J. B. 1954
An analysis of a logical machine using parenthesis-free notation.
Math. Tables and Other Aids to Comp., No. 46, pp. 53-57.

Burks, A. W. and Copi, I. M. 1956
The logical design of an idealized general purpose computer.
J. Franklin Inst., Vol. 261, pp. 299-314, 421-436.

Burks, A. W. and Wright, J. B. 1962
Sequence generators, graphs, and formal languages.
Inf. and Control 5, 3, pp. 204-212.

Burks, A. W. and Wang, H. undated
The logic of automata.
U.S.A.F. Off. Sci. Res. Develop., Wash., D. C.

Brzozowski, J. 1962
A survey of regular expressions and their applications.
PGEC 11, 3, pp. 324-335.

Brzozowski, J. A. 1964
Derivatives of regular expressions.
JACM 11, pp. 481-494.

Burks, A. W., Warren, D. W. and Wright, J. B. 1954
An analysis of a logical machine using parenthesis-free notation.
Math. Tables and Other Aids to Comp., 8, pp. 53-57.

Cantor, D. C. 1962
On the ambiguity problem of Backus systems.
JACM 9 4, pp. 477-479.
Caracciolo, A. 1963
Some remarks on the syntax of symbolic programming languages.
CACM 6, pp. 456-460.
Carnap, R. 1937
The logical syntax of language.
N.Y.: Harcourt, Brace and Co.
Carnap, R. 1958
Introduction to symbolic logic and its applications.
N.Y.: Dover Press.
Chaitin, G. J. 1966
On the length of programs for computing finite binary sequences.
JACM 13, 1, p. 547.
Chartres, B. A. and Florentin, J. J. 1968
A universal syntax-directed top-down analyzer.
JACM 15 1, p. 447.
Cheatham, T. E. and Sattley, K. 1964
Syntax-directed compiling.
Proc. AFIPS Spr. Joint Comp. Conf. 25, pp. 31-57
Baltimore: Spartan Books.
Cheatham, T. E. 1966
The theory and construction of compilers.
Computer Assoc., Inc., Wakefield, Mass.
Chomsky, N. 1953
Systems of syntactic analysis.
J. Symb. Logic 18, pp. 242-256.
Chomsky, N. 1955
Transformational analysis.
Thesis, U. of Pa., Phila., Pa.
Chomsky, N. 1956
Three models for the description of language.
IRE Trans. on Inform Theory, vol. IT-3, pp. 113-124
also in: PGIT 2,3, pp. 113-124 ??
Chomsky, N. 196_
The logical structure of linguistic theory.
Mimeographed, source unknown.

Chomsky, N. 1957
Syntactic structures.
s-Gravenhage: Mouton and Co.

Chomsky, N. 1959
On certain formal properties of grammars.
Inf. and Control 2, 2, pp. 137-167.

Chomsky, N. 1959
A note on phrase structure grammars.
Inf. and Control 2, pp. 393-395.

Chomsky, N. 1961
On the notion "rule of grammar".
Proc. Symp. Appl. Math. 12, AMS, Providence, R.I.

Chomsky, N. 1962
Context-free grammars and pushdown storage.
Quart. Prog. Dept. No. 65, MIT Res. Lab. Elect., pp. 187-194.

Chomsky, N. 1963
Formal properties of grammars.
Handbook of Math. Psych. 2, pp. 323-418. Wiley, N.Y.

Chomsky, N. and Miller, G. A. 1958
Finite state languages.
Inf. and Control 1, 2, pp. 91-112.

Chomsky, N. and Miller, G. A. 1963
Introduction to the formal analysis of natural languages.
in R. R. Bush, E.H. Galanter and R. D. Luce (eds.)
Handbook of Math. Psych 2, pp. 269-322. J. Wiley, N.Y.

Chomsky, N. and Schutzenberger, M. P. 1963
The algebraic theory of context-free languages.
Computer Programming + Formal Systems, North Holland,
Amsterdam, pp. 118-161.

Church, A. 1937
Combinatory logic as a semigroup, preliminary report.
Bull. AMS 43, p. 33.

Church, A. 1943
Review of Post's 'Formal reductions of the general
combinatorial decision problem'. (Am.J.Math. 65, pp. 197-215).
J. Symb. Logic 8, pp. 50-52. Erratum ibid., p. iv.
See also Note theorem by Post, Bull. AMS 52, p. 264 [1946]

Clifford, A. H. and Preston, G. B. 1961
The algebraic theory of semigroups.
AMS Math. Surv. 7.

Cohen, J. M. 1967
The equivalence of two concepts of categorial grammar.
Inf. and Control 10, 5, p. 475.

Cole, S. N. 1964
Real-time computation by iterative arrays of finite-state machin
Doctoral thesis, Harvard U., Cambridge, Mass.
IEEE Conf. Record of 7th Ann. Symp. on Switch. + Automata Theory
Berkeley, Calif., pp. 53-77.

Conway, M. E. 1958
Proposal for an UNCOL.
CACM 1, 10, pp. 5-8.

Conway, M. E. 1963
Design of a separable transition-diagram compiler.
CACM 6, 7, pp. 396-408.

Cook, S. A. 1966
The solvability of the derivability problem for one-normal syste
JACM 13, 1, p. 223.

Copi, I. M., Elgot, C. C. and Wright, J. B. 1958
Realization of events by logical sets.
JACM 5, pp. 181-196.

Church, A. 1936
An unsolvable problem of elementary number theory.
Amer. J. Math. 58, pp. 345-363.

Cudia, D. F. and Singletary, W. E. 1968
Degrees of unsolvability in formal grammars.
JACM 15, 1, p. 680.

Culik III, K. 1965
Axiomatic system for phrase structure grammars. I.
Inf. and Control 8, 5, p. 493.

Culik,III, Karel 1967
On some transformations in context-free grammars and languages.
Czech. Math. Jour. 17, pp. 278-310.

Culik III, K. 1961
Some notes on finite-state languages and events represented by finite automata using labeled graphs.
Cas. pro Pest. Mat. 83, pp. 43-55.

Culik III, K. 1962
Formal structure of ALGOL and simplification of its description.
Proc. ICC Symp. Symb. Lang. in Data Proces.
N.Y. and London: Gordon and Breach, 1pp. 75-82.

Culik III, K. 1963
On some axiomatic systems for formal grammars and languages.
Proc. IFIP Congress, Munich, '62.
Amsterdam: North-Holland, pp. 313-317.

Culik III, K. 1964
Applications of graph theory to mathematical logic and linguistics.
The theory of graphs and its applications. Prague: Czech. Acad. of Sci., pp. 13-20.

Curry, J. B. and Feys, R. 1958
Combinatory logic, I.
Amsterdam: North-Holland.

D

van Dalen, Dirk 1968
Fans generated by nondeterministic automata.
Zeitschr. f. math. Logik und Grundlagen d. Math. 14, pp. 273-278

Davis, M. 1950
On the theory of recursive unsolvability.
Thesis, Princeton U., Princeton, N. J.

Davis, M. 1958
Computability and Unsolvability.
McGraw-Hill, New York.

Dijkstra, E. W. 1960
Recursive programming.
Num. Math. 2, pp. 312-318.

Dijkstra, E. W. 1961
On the design of machine-independent programming languages.
Rep. No. MR-34, Math. Centrum, Amsterdam.

Dijkstra, E. W. 1961
ALGOL 60 translation.
ALGOL Bull. Suppl. No. 10, Math. Centrum, Amsterdam.
Dijkstra, E. W. 1962
An attempt to unify tee constituent concepts of serial program execution.
Proc. ICC Symp. Symb. Lang. in Data Proces.
N.Y. and London: Gordon and Braach, pp. 237-251.
Dijkstra, E. W. 1964
A simple mechanism modeling some features of ALGOL 60.
in F. Duncan (ed.) ALGOL Bull. No. 16, The Hague, pp. 14-23.

Earley, J. 1967
An n^2-recognizer for context free grammars.
Research Rep., Carnegie-Mellon U., Pittsburgh, Pa. Sept.
Earley, Jay 1968
An efficient context-free parsing algorithm.
Computer Sci. Dept., Carnegie-Mellon U., Pittsburgh.
Eickel, J. 1964
Generation of parsing algorithms for Chomsky-2 type languages.
Math. Inst. Techn. Hochschule, Munich.
Eickel, J., Paul, M., Bauer, F. L. and Samelson, K. 1962-3
A syntax-controlled generator of formal language processors.
Inst. für Agnew, Math., U. of Mainz, also in CACM 6, 8,pp.451-55.
Eilenberg, S. and Wright, J. B. 1967
Automata in general algebras.
Inf. and Control 11, p. 452.
Elgot, C. C. 1961
Decision problems of finite automata design and related arithmet:
Trans. AMS 98, pp. 21-51.
Elgot, C. C. 1962
Review of 'A remark on finite transducers'.
IRE Trans. Electron. Computers, Vol. EC 11, p. 802.
Elgot, C. C. and Mezei, J. E. 1965
On relations defined by generalized finite automata.
IBM J. Res. Devel. 9, pp. 47-68.

Elgot, C. C. and Robinson, A. 1964
Random-access, stored-program machines, an approach to programming languages.
Rep. No. RC-1101, IBM Res. Lab., Yorktown Hgts. N.Y.
also in JACM 11, 4, pp. 365-399.

Elgot, C. C. and Rutledge, J. D. 1961
Operations on finite automata.
Proc. AIEE Sec. Ann. Symp. Switching Circuit Theory Logic. Des., Detroit, Mich, pp. 129-132.

Ershov, A. P. 1959
Programming programme for the BESM computer.
N.Y.: Pergamon Press.

Evans, A. 1964
An ALGOL-60 compiler.
Annual Review in Automatic Programming 4, pp. 87-124. Pergamon Pre

Evans, A., Perlis, A. J. and Van Zoeren, H. 1961
The use of threaded lists, etc.
CACM 4,1,pp. 36-41.

Evey, J. 1963
The theory and application of pushdown store machines.
Ph.D. Thesis, Harvard U., Cambridge, Mass.
also in Math. Ling. + Autom. Transl., Harvard U.
Comp. Lab. Rep. NSF-10.

Fabian, V. 1963
A recursive procedure for compiling expression
Chiffres 2, 4, pp. 275-281.

Fabian, V. 1964
Structural unambiguity of formal languages.
Czech, Math. J. 14, 89, pp. 394-430.

Feldman, J. A. 1966
A formal semantics for computer languages and its application in a compiler-compiler.
CACM 9, 1, pp. 3-9.

Feldman, J. and Gries, D. 1968
Translator writing systems.
CACM 11, 2, pp. 77-113.

Fenichel, R. R. 1964
On the nonexistence of a type 1 grammar for FAP.
Math. Ling. + Automatic Transl. Harvard U. Comp. Lab.
Rep. NSF-13 March.

Fischer, M. J. 1968
Grammars with macro-like productions.
Aiken Comp. Lab., Harvard U., Rep. No. NSF-22 to NSF- .

Fischer, P. C.
On computability by certain classes of restricted Turing machines
Proc. 4th Ann. Symp. on Switching Circuit Theory + Log. Des.,
Chicago, Ill. pp. 23-32.

Fischer, P. C. 1965
Multitape and infinite state automata -- a survey.
CACM 8, 12, pp. 799-805.

Fischer, P. C. 1966
Turing machines with restricted memory access.
Inf. and Control 9, 4, pp. 364-379.

Fischer, P. C. 1968
Turing machines with a schedule to keep.
Inf. and Control 11, p. 138.

Fischer, P., Meyer, A. and Rosenberg, A. 1967
Real time counter machines.
IEEE Conf. Rec. of 8th Ann. Symp. on Switch. + Automata Theory
pp. 148-154 October.

Floyd, R. W. 1961
A descriptive language for symbol manipulation.
JACM 8, 10, pp. 579-584.

Floyd, R. W. 1962
On ambiguity in phrase structure languages.
CACM 5, 10, pp. 526-534.

Floyd, R. W. 1962
On the nonexistdnce of a phrase structure grammar for ALGOL 60.
CACM 5, 9, pp. 483-484.

Floyd, R. W. 1963
Syntactic analysis and operator precedence.
JACM 10, 3, pp. 316-333.

Floyd, R. W. 1964
Bounded context syntactic analysis.
CACM 7, 2, pp. 62-67.

Floyd, R. W. 1964
New proofs and old theorems in logic and formal linguistics.
Computer Assoc., Inc., Wakefield, Mass.

Floyd, R. W. 1964
The syntax of programming languages -- a survey.
PGEC 13, 4, pp. 346-353.

Floyd, R. W. 1967
Nondeterministic algorithms.
JACM 14, 1, p.636.

Fris, I. 1968
Grammars with partial ordering of the rules.
Inf. and Control 12, 5,6, p. 415.

Fris, Ivan 1965
On stop-conditions in the definitions of constructive languages.
Zeitschr. f. math. Logik und Grundlagen 11, pp. 61-73.

Fris, Ivan 1968
Grammars with partial ordering of the rules.
Inf. and Control. 12, 1, pp. 415-425.

Gaifman, H. 1965
Dependency systems and phrase structure systems.
Inf. and Control 8, pp. 304-337.

Galler, B. A. and Perlis, A. J. 1967
A proposal for definitions in Algol.
CACM 10, 4, pp. 204-219.

Gilbert, P. 1966
On the syntax of algorithmic languages.
JACM 13, 1, p. 90.

Gilbert, P., Hosler, J. and Schager, C. 1962
Automatic programming techniques.
TDR-62-632, Rome Air Devel. Cr., Rome, N.Y.

Gill, A. 1962
Introduction to the theory of finite-state machines.
McGraw-Hill, N.Y.

Gilmore, P. C. 1963
An abstract computer with LISP-like machine language without a label operator.
in P. Braffort and D. Hirschberg (eds.) Computer programming and Formal Systems, Amsterdam: North-Holland, pp. 71-86.

Ginsburg, S. 1962
Examples of abstract machines.
IRE Trans. Electron. Computers EC11, pp. 132-135.

Ginsburg, S. 1962
An introduction to mathematical machine theory.
Reading: Addison-Wesley (Mass.)

Ginsburg, S. 1964
A survey of ALGOL-like and context-free language theory.
TM-738/006/00, Sys. Devel. Corp., Santa Monica, Cal.

Ginsburg, S. 1966
The mathematical theory of context-free languages.
New York: McGraw-Hill.

Ginsburg, S. and Greibach, S. A. 1966
Mappings which preserve context-sensitive languages.
Inf. and Control 9, 6, pp. 563-582.

Ginsburg, S. and Greibach, S. A. 1966
Deterministic context-free languages.
Inf. and Control 9, 6, pp. 620-648.
also in: Sys. Devel. Corp. Rep. TM- 738/014/00.

Ginsburg, S. and Greibach, S. A. 1967
Abstract families of languages.
IEEE Conf. Rec. of 8th Ann. Symp. Switch. + Automata Theory, Austin, Texas. Also in: Sys.Dev.Corp. Rep. TM-738/031/00.
To appear in Memoir of AMS.

Ginsburg, S., Greibach, S. A. and Harrison, M. A. 1967
Stack automata and compiling.
JACM 14, 1, pp. 172-201.

Ginsburg, S., Greibach, S. A. and Harrison, M. A. 1967
One-way stack automata.
JACM 14, 2, pp. 389-418.
Ginsburg, S., Greibach, S. A. and Hopcroft, J. E. 1967
Pre-AFL.
SDC Document TM 738/037/00.
also to appear in Memoir of AMS.
Ginsburg, S. and Harrison, M. A. 1966
Bracketed context free languages.
Sys. Dev. Corp. Rep. TM-738/023/00
Ginsburg, S. and Harrison, M. A.
One-way nondeterministic real-time list-storage languages.
JACM 15, 1, p. 428.
Ginsburg, S. and Hibbard, T. N. 1963
Two theorems about regular bases of sets of words.
Sys. Dev. Corp. Rep. TM-738/004/00.
Ginsburg, S. and Hibbard, T. N. 1964
Solvability of machine mappings of regular sets to regular sets.
JACM 11, pp. 302-312.
Ginsburg, S., Hibbard, T. N. and Ullian, J. S. 1965
Sequences in context free languages.
Ill. Jour. Math. 9, pp. 321-337.
Ginsburg, S. and Hopcroft, J. E. 1968
Two-way balloon automata and AFL's.
SDC Document TM 738/042/00.
Ginsburg, S. and Rice, H. G. 1962
Two families of languages related to ALGOL.
JACM 9, 3, pp. 350-371.
Ginsburg, S. and Rose, G. F. 1963
Some recursively unsolvable problems in ALGOL-like languages.
JACM 10, 1, pp. 29-47.
Ginsburg, S. and Rose, G. F. 1963
Operations which preserve definability in languages.
JACM 10, 2, pp. 175-195.
Ginsburg, S. and Rose, G. F. 1966
Preservation of languages by transducers.
Inf. and Control 9, 1, pp. 153-176.

Ginsburg, S. and Rose, G. F. 1966
A characterization of machine mappings.
Can. Jour. Math. 18, pp. 381-388.
Ginsburg, S. and Rose, G. F. 1968
A note on preservation of languages by transducers.
Inf. and Control 12, 5,6, p. 549.
Ginsburg, S. and Spanier, E. H. 1963
Quotients of context-free languages.
JACM 10, 4, pp. 487-492.
Ginsburg, S. and Spanier, E. H. 1965
Mappings of languages by two-tape devices.
JACM 12, pp. 423-434.
Ginsburg, S. and Spanier, E. H. 1966
Semigroups, Presburger formulas, and languages.
Pacif. J. Math. 16, pp. 285-296.
Ginsburg, S. and Spanier, E. H. 19__
Finite-turn pushdown automata.
ISAM J. Control, to be published.
Ginsburg, S. and Spanier, E. H. 1966
Bounded regular sets.
Sys. Devel. Corp. Rep. TM-738/024/00.
Ginsburg, S. and Spanier, E. H. 1964
Bounded ALGOL-like languages.
Trans. AMS 113, pp. 333-368
also in SDC Rep. TM-738/002/00.
Ginsburg, S. and Spanier, E. H. 1967
Control sets on grammars.
SDC Document 738/036/00.
Ginsburg, S. and Ullian, J. 1966
Ambiguity in context-free languages.
JACM 13, 1, pp. 62-88.
Ginsburg, S. and Ullian, J. 1966
Preservation of unambiguity and inherent ambiguity in context-free languages.
JACM 13, 1, p. 364.

Gladkii, A. V. 1963
Grammars with linear memory.
Algebra Logica Sem. 2, pp. 43-55 (Russ.)

Godel, K. 1934
On undecidable propsotions of formal mathematical systems.
(Mimeo. Lec. Notes) Inst. Adv. Study, Princeton, N.J.

Gold, E. M. 1967
Language identification in the limit.
Inf. and Control 10, 5, p. 447.

Gorn, S. 1959-60
A perspective view of mechanical digital languages.
ICIP Proc., Paris. London: Butterworth's, pp. 117-118.

Gorn, S. et al. 1959
Common programming language task, I.
U. of Pa., Phila., Pa.

Gorn, S. and Parker, E. J. 1960
Common programming language task, I.
U. of Pa. Phila. Pa.

Gorn, S. 1961
Some basic terminology connected tith mechanical languages
and their processors.
CACM 4, 8, pp. 336-339.

Gorn, S. 1961
Specification languages for mechanical languages and
their processors - a baker's dozen.
CACM 4, 12, pp. 532-542.

Gorn, S. 1961
The treatment of ambiguity and paradox in mechanical languages.
AFOSR Rep. No. TN-603-61, Off. of Comp. Res. + Educ., U.of Pa.
also in Proc. Symp. Pure Math. 5, recursive function theory,
Providence R.I. AMS.

Gorn, S. 1962
Processors for infinite codes of the Shannon-Fano type.
Proc. Symp. Math. Theory of Automata. Poly. Inst. Bklyn., N.Y.

Gorn, S. 1962
An axiomatic approach to prefix languages.
Proc. ICC Symp. on Symb. Lang. in Data Processing
N.Y. and London: Gordon and Breach, pp. 1-21.

Gorn, S. 1962
Mechanical pragmatics, a time-motion study of a miniature mechanical linguistic system.
CACM 5,12, pp. 576-589.

Gorn, S. 1963
Detection of generative ambiguities in context-free mechanical languages.
JACM 10, 2, pp. 196-208.

Gorn, S., Ingerman, P. Z. and Crozier, J. B. 1960
On the construction of micro-flowcharts.
CACM 2, 10, pp. 27-31.

Gray, J. N. and Harrison, M. A. 1966
The theory of sequential relations.
Inf. and Control 9, 5, p. 435.

Gray, J. N., Harrison, M. A. and Ibarra, O. 1967
Two-way pushdown automata.
Inf. and Control 11, 1,2, pp. 30-70.

Greibach, S. A. 1963
Inverses of phrase structure generators.
Math. Ling. + Automatic Transl., Harvard U. Comp. Lab. Rep. NSF-11.

Greibach, S. A. 1963
The undecidability of the ambiguity problem for minimal linear grammars.
Inf. and Control 6, 2, pp. 117-125.

Greibach, S. A. 1964
Formal parsing systems.
CACM 7, pp. 499-504.

Greibach, S. A. 1965
A new normal form theorem for context-free phrase structure grammars.
JACM 12, 1, pp. 42-52.

Greibach, S. A. 1965
A note on pushdown store automata and regular systems.
SDC Rep. TM-738/016/00.

Greibach, S. A. 1965
A note on code sets and context free languages.
Math. Ling. + Autom. Transl., Harvard U. Comp. Lab.
Rep. NSF 15 II.1-II.5.

Greibach, S. A. 1966
The unsolvability of the recognition of linear context-free languages.
JACM 13, 4, pp. 582-587.

Greibach, S. A. 1967
A note on undecidable properties of formal languages.
SDC Doc. TM 738/038/00.

Greibach, S. A. 1968
Checking automata and one-way stack languages.
SDC Doc. TM 738/045/00.

Greibach, S. A. and Hopcroft, J. E. 1967
Independence of AFL operations.
SDC Doc. TM 738/034/00.

Greibach, S. and Hopcroft, J. 1968
Scattered context grammars.
SDC Rep. 738/043/00.

Griffiths, T. V. 1966
An upper bound on the time required for parsing by predictive analysis with abortive path elimination.
4th Ann. Meet. Assoc. Mach. Transl. + Comp. Ling. UCLA.

Griffiths, T. V. 1968
Some remarks on derivations in general rewriting systems.
Inf. and Control 12, 1, pp. 27-54.

Griffiths, T. V. 1968
The unsolvability of the equivalence problem for Λ-free nondeterministic generalized machines.
JACM 15, 1, p. 409.

Griffiths, T. V. and Petrick, S. R. 1965
On the relative efficiencies of context-free grammar recognizers
AF Camb. Res. Rep.
rev. version in CACM 8, pp. 289-300.

Gross, M. 1963
Linguistique mathematique et langages de programmation.
Chiffres 2, 4, pp. 231-254.
Gross, M. 1964
Inherent ambiguity of minimal linear grammars.
Inf. and Control 7, 3, pp. 366-368.
Gruska, J. 1965
On structural unambiguity of formal languages.
Czech. Math. J. 15, pp. 283-293.
Gruska, J. 1965
Induction in formal languages. Some properties of reducing transformations and of isolable sets.
Czech. Math. J. 15, pp. 406-414.
Gruska, J. 196_
Isolable and weakly isolable sets.
Czech, Math. J., to be published
Grzegorczyk, A. 1953
Some classes of recursive functions.
Rosprawy matematyczne 4, Instytut Math. Polskiej Akademie Nauk, Warsaw.

Haines, L. H. 1964
Note on the complement of a (minimal) linear language.
Inf. and Control 7, 3, pp. 307-314.
Haines, L. H. 1965
Generation and recognition of formal languages.
Ph.D. Thesis, M.I.T., Cambridge, Mass.
Halmos, P. R. 1960
Naive set theory.
D. Van Nostrand Co., New Jersey.
Harary, F. and Palmer, E. 1967
Enumeration of finite automata.
Inf. and Control 10, 5, p. 499.
Harrison, M. A. 1965
Introduction to switching and automata theory.
McGraw-Hill, New York.

Hartmanis, J. 1967
On memory requirements for context-free language recognition.
JACM 14, 1, p. 663.

Hartmanis, J. 1967
Context-free languages and Turing machine computations.
Proc. Symp. Appl. Math. 19, AMS, Providence, R. I.

Hartmanis, J. 1967
On the complexity of undecidable problems in automata theory.
IEEE Conf. Record of 8th Ann. Symp. on Switch.+ Automata Theory,
Austin, Texas, pp. 112-116.

Hartmanis, J. 1968
Computational complexity of one-tape Turing machine computations
JACM 15, 1, p. 325.

Hartmanis, J., Lewis II, P.M. and Stearns, R. E. 1965
Hierarchies of memory limited computations.
IEEE Conf. Record on Switch. Circuit Theory + Log. Des.,
Ann Arbor, Mich. pp. 179-190.

Hartmanis, J., Lewis, P. M. and Stearns, R. F. 1965
Classifications of computations by time and memory requirements.
Proc. IFIP Congress 65, Spartan Books, Inc., pp. 31-35.

Hartmanis, J. and Shank, H. 1968
On the recognition of primes by automata.
JACM 15, 1, p. 382.

Hartmanis, J. and Stearns, R. E. 1964
Computationsl complexity of recursive sequences.
Proc. 5th Ann. Symp. on Switch. Circuit Theory + Log. Des.
Princeton, N.J. pp. 82-90.

Hartmanis, J. and Stearns, R. E. 1965
On the computational complexity of algorithms.
Trans. AMS 117, pp. 285-306.

Harwood, F. W. 1955
Axiomatic syntax, the construction and evaluation of a
syntactic calculus.
Language 31, pp. 409-414.

Hays, D. 1962
Automatic language-data processing.
in Computer Applications in the Behavioral Sciecnes,
H. Borko (ed.) Prentice Hall, Englewood, N. J.
Hennie, F. C. 1965
One-tape, off-line Turing machine computations.
Inf. and Control 8, 6, pp. 553-578.
Hennie, F. C. 1966
On-line Turing machine computations.
IEEE Trans. Elec. Comp. EC-15, pp. 35-44.
Hennie, F. C. and Stearns, R. E. 1966
Two-tape simulation of multitape Turing machines.
JACM 13, 4, pp. 533-546.
Hibbard, T. N. 1968
A generalization of context-free determinism.
Inf. and Control 11, p. 196.
Hibbard, T. N. and Ullian, J. 1966
The independence of inherent ambiguity from complementedness
among context-free languages.
JACM 13, 1, p. 588.
Holt, A. W. and Turanski, W. J. 1959
Common programming language task, II.
U. of Pa., Phila, Pa.
Holt, A. W., Turanski, W. J., and Parker, E. J. 1960
Common programming language task, II.
U. of Pa., Phila., Pa.
Holt, A. W. 1963
A mathematical and applied investigation of tree structures
for computer syntactic analysis.
Thesis, U. of Pa., Phila., Pa.
Hopcroft, J. E. and Ullman, J. D. 1967
Nonerasing stack automata.
JCSS 1, 2, pp. 166-186.
Hopcroft, J. E. and Ullman, J. D. 1967
An approach to a unified theory of automata.
Bell Sys. Tech. Jour. 46, 8, pp. 1863-1829.

Hopcroft, J. E. and Ullman, J. D. 1968
Decidable and undecidable questions about automata.
JACM 15, 2, pp. 317-324.

Hopcroft, J. E. and Ullman, J. D. 1968
Deterministic stack automata and the quotient operator.
JCSS 2, 1, pp. 1-12.

Hopcroft, J. E. and Ullman, J. D. 1968
Sets accepted by one-way stack automata are context sensitive.
Inf. and Control, to appear.

Hopcroft, J. E. and Ullman, J. D. 1968
Two results on one-way stack automata.
IEEE Conf. Rec. of 8th Ann. Symp. on Switch. + Automata Theory,
Austin, Texas.

Hopcroft, J. E. and Ullman, J. D. 1968
Relations between time and tape complexities.
JACM 15, 1, p. 414.

Hopcroft, J. and Ullman, J. 1969
Formal languages and their relation to automata.
Addison and Wesley.

Huffman, D. A. 1954
The synthesis of sequential switching circuits.
Jour. of the Franklin Inst. 257, 3,4 pp. 161-190, 275-303.

Hooper, P. K. 1966
Monogenic post normal systems of arbitrary degree.
JACM 13, 1, p. 359.

Hooper, P. K. 1966
The immortality problem for post normal systems.
JACM 13, 1, p. 594.

Ingerman, P. Z. 1962
A translation technique for languages whose syntax is expressible in extended Backus Normal Form.
Proc. ICC Symp. Lang. in Data Proces.
N.Y. and London: Gordon and Breach, pp. 23-64.

Ingerman, P. Z. 1963
A syntax-oriented compiler for languages whose syntax is expressible in Backus Normal Form, and some propssed extensions thereto.
M.S., U. of Pa., Moore Sch. Elec. Engr., Phila., Pa.

Irons, E. T. 1961
A syntax directed compiler for ALGOL 60.
CACM 4, 1, pp. 51-55.

Irons, E. T. 1963
The structure and use of the syntax-directed compiler.
in R. Goodman (ed.) Ann. Rev. in Autom. Prog. 3, pp. 207-227
N.Y. and London: Pergamon Press.

Irons, E. T. 1964
Structural connections in formal languages.
CACM 7, 2, pp. 67-72.

James, J. S. 1965
A formal macro assembler -- version IV.
Natl. Inst. of Mental Health, unpublished paper.

Karp, R. M. 1959
Some applications of logical syntax to digital computer programmi
Thesis, Harvard U., Cambridge, Mass.

Kasami, T. 1966
An efficient recognition and syntax analysis algorithm for context free languages.
U. of Ill.

Kasami, T. 1967
A note on computing time for recognition of languages generated by linear grammars.
Inf. and Control 10, 2, pp. 209-214.

Kasami, T. and Torii, K. 1967
Some results on syntax analysis of context free languages.
Record of Tech. Group on Automata Theory of Inst. of Electronic Communication Engrs., Japan.

Kasami, T., Torii, K. and Ozaki, H. 196_
Translation of finite state languages by a sequential machine.
J. Inst. Elec. Commun. Engs. Japan, to be published.

Kimball, John P. 1968 (67)
Predicates definable over transformational derivations by intersection with regular languages.
Inf. and Control 11, p. 177.

Kleene, S. C. 1936
General recursive functions of natural numbers.
Mathematische Annalen 112, pp. 727-742.

Kleene, S. C. 1952
Introduction to Metamathematics.
Princeton: D. Van Nostrand (N.J.)

Kleene, S. C. 1956
Representation of events in never sets and finite automata.
Automata Studies, Princeton U. Press, Princeton N.J., pp. 3-42.

Knuth, D. E. 1965
On the translation of languages from left to right.
Inf. and Control 8, 6, pp. 607-639.

Knuth, D. E. 1967
A characterization of parenthesis languages.
Inf. and Control 11, 3, pp. 269-289.

Knuth, D. E. and Bigelow, R. 1967
Programming languages for automata.
JACM 14, 4, pp. 615-635.

Kobayashi, K. and Sekiguchi, S. 1966
On the class of predicates decidable by two-way multitape finite automata.
JACM 13, 1, p. 236.

Konig, D. 1950
Theorie der Endlichen und Unendlichen Graphen.
Chelsea Publ. Co., N.Y.

Korenjak, A. J. 1967
A practical approach to the construction of deterministic language processors.
RCA Labs., Princeton, N. J.

Korenjak, A. J. and Hopcroft, J. E. 1966
Simple deterministic languages.
IEEE Conf. Rec. of 7th Ann. Symp. on Switch. + Autom. Theo.
Berkeley, Cal., pp. 36-46.

Kreider, D. L. and Ritchie, R. W. 1966
A basis theorem for a class of two-way automata.
Zeitschr. f. math. Logik und Grundlagen d. Math., Bd.12,pp.243-55

Krutar, R. 1968
unpublished
Computer Sci. Dept., Carnegie-Mellon U.

Kuno, S. and Oettinger, A. G. 1963
Multiple-path syntactic analyzer.
Information Processing 62 (IFIP Cong.)
Popplewell (ed.), Amsterdam: North-Holland, pp. 306-311.

Kuno, S.
The predictive analyzer and a path elimination technique.
CACM 8, pp. 453-463.

Kuroda, S. Y. 1964
Classes of languages and linear-bounded automata.
Inf. and Control 7, 2, pp. 207-223.

Landin, P. J. 1964
The mechanical evaluation of expressions.
Comp. J. 6, 4, pp. 308-320.

Landweber, P. S. 1963
Three theorems on phrase structure grammars of type 1.
Inf. and Control 6, 2, pp. 131-136.

Landweber, P. S. 1964
Decision problems of phrase structure grammars.
PGEC 13, 4, pp. 354-362.

Levien, R. E. 1962
Studies in the theory of computational algorithms,1.
Formalization, computability, representation, and analysis proble
RM-3007-PR, Rand Corp., Santa Monica.

Lewis, P. M. and Stearns, R. E. 1966, 68
Syntax directed transduction.
IEEE Conf. Rec. 7th Ann. Symp. Switch. + Autom. The.
Berkeley, Cal., pp. 21-35.
Also JACM 15, 1, p. 464.

Lewis, P. M., Stearns, R. E. and Hartmanis, J. 1965
Memory bounds for recognition of context-free and
context-sensitive languages.
IEEE Conf. Rec. on Switch. Circ. Theo. + Log. Des.,
Ann Arbor, Mich., pp. 191-202.

Lynch, W. C. 1963
Ambiguities in Backus Normal Form languages.
Ph.D., U. of Wisc., Madison.

McCarthy, J., Abrahams, P., Edwards, D., Hart, T. and Levin, M. 1962
LISP 1.5 Programmers Manual.
MIT Press, Cambridge, Mass.

McCarthy, J. 1963
Basis for a mathematical theory of computation.
in P. Braffort and D. Hirschberg (eds.) Computer Programming
and Formal Systems, Amsterdam: North-Holland, pp. 33-71.

McCulloch, W. S. and Pitts, W. 1943
A logical calculus of the ideas immanent in nervous activity.
Bull. Math. Biophysics 5, pp. 115-133.

McIlroy, M. D. 1960
Macro instruction extension of compiler languages.
CACM 3, 4, pp. 214-220.

McNaughton, R. 1966
Testing and generating infinite sequences by a finite automaton.
Inf. and Control 9, 5, p. 521.

McNaughton, Robert 1968
The loop complexity of pure-group events.
Inf. and Control 11, p. 167.

McNaughton, R. 1967
Parenthesis grammars.
JACM 14, 3, pp. 490-500.

McNaughton, R. and Yamada, H. 1960, 59
Regular expressions and state graphs for automata.
PGEC 9, 1, pp. 39-47.
Also Rep. 60-66, U. of Pa., Moore Sch. Elec. Engr., Phila.

Markov, A. A. 1961
Theory of algorithms.
Moscow: USSR Acad. Sci. ('54)
Translated into English by the Israeli Program for
Scientific Translations, Jerusalem.

Martin, W. A. 1968
A left to right then right to left parsing algorithm.
Project MAC, MIT.

Matthews, G. H. 1963
Discontinuity and asymmetry in phrase structure grammars.
Inf. and Control 6, pp. 137-146.

Matthews, G. H. 1964
A note on asymmetry in phrase structure grammars.
Inf. and Control 7, pp. 360-365.

Matthews, G. H. 1967
Two-way languages.
Inf. and Control 10, 2, p. 111.

Mezei, J. and Wright J. B. 1967
Algebraic automata and context-free sets.
Inf. and Control 11, p. 3.

Minsky, M. L. 1961
Recursive unsolvability of Post's problem of "tag" and
other topics in the theory of Turing machines.
Annals of Math. 74, 3, pp. 437-455.

Minsky, M. L. 1967
Computation: finite and infinite machines.
Englewood Cliffs: Prentice-Hall (N.J.)

Minsky, M. and Papert, S. 1966
Unrecognizable sets of numbers.
JACM 13, 1, p. 281.

Mooers, C. N. 1966
TRAC, a procedure-describing language for the reactive typewriter
CACM 9, 3, pp. 215-219.

Moore, E. F. 1956
Gedanken experiments on sequential machines.
Automata Studies, Princeton U. Press, Princeton,N.J., pp. 129-53.

Myhill, J. 1957
Finite automata and the representation of events.
Wright Air Devel. Command Tech. Rep. 57-624, pp. 112-137.

Myhill, J. 1960
Linear bounded automata.
WADD Tech. Note, pp. 60-165. Wright Patterson AFB, Ohio.

Naur, P. (ed.) 1963
Revised report on the algorithmic language ALGOL 60.
CACM 6, 1, pp. 1-17.

Nerode, A. 1958
Linear automaton transformations.
Proc. AMS 9, pp. 541-544.

Newell, A. and Shaw, J. C. 1957
Programming the logic theory machine.
Proc. Western Joint Comp. Conf. pp. 230-240.

Ochranova-Dolezelova, Renata 1968
Real-time decidability, computability, countability and generability.
Zeitschr. f. math. Logik und Grundlagen d. Math. 14, pp. 283-88.

Oettinger, A. G. 161961
Automatic syntactic analysis and the pushdown store.
Proc. Symp. Appl. Math. 12, AMS, Providence R.I.

Palermo, G. and Pacelli, M. 1962
Sequential translation of a problem-oriented programming language
Proc. ICC Symp. Symb. Lang, in Data Proces. NY+Lond:Gordon+Breach
pp. 263-269.
Parikh, R. J. 1961, 66
Language generating devices.
Quart. Prog. Rep. 60, MIT Res. Lab. Elect., pp. 199-212.
reprinted as "On context-free languages" JACM 13, 4, pp. 570-81.
Paul, M. 1962
A general processor for certain formal languages.
Proc. ICC Symp. Symb. Lang. Data Proces.
N.Y. and London: Gordon and Breach, pp. 65-74.
Paul, M. 1962-3
ALGOL 60 processors and a processor generating
Proc. IFIP Congress, Munich '62. Amsterdam:North-Holland,pp.439-
Paul, M. and Unger, S. H. 1967
Structural equivalence of context-free grammars.
IEE Conf. Rec. of 8th Ann. Symp. Switch.+ Auto. Theo.
Austin, Tex. pp. 7-13.
Petrone, L. 1965
On precedence grammars.
Olivetti Electron. Res. Lab. Rep.
Picciafuoco, U. and Pacelli, M. 1962
Nondynamic aspects of recursive programming.
Proc. ICC Symp. Symb. Lang. Data Proces.
N.Y. and London: Gordon and Breach, pp. 317-24.
Pickett, H. E. 1967
Note concerning the algebraic theory of automata.
JACM 14, 1, p. 382.
Post, E. L. 1936
Finite combinatory processes - formulation I.
J. Symb. Log. 1, pp. 103-105.
Post. E. L. 1943
Formal reductions of the general combinatorial decision problem.
Am. J. Math. 65, pp. 197-215.

Post, E. L. 1944
Recursively enumerable sets of positive integers and their decision problems.
Bull. AMS 50, pp. 284-316.

Post, E. L. 1946
A variant of a recursively unsolvable problem.
Bull. AMS, 52, pp. 264-268.

Post, E. L. 1947
Recursive unsolvability of a problem of Thue.
Jour. Symb. Logic 12, pp. 1-11.

Proctor, R. M. 1964
A logic design translator experiment demonostrating relationships of language to systems and logic design.
IEEE Trans. on Elec. Comp. EC-13, 4, pp. 422-430.

Rabin, M. O. 1963
Real-time computation.
Israel J. Math. 1, 4, pp. 203-211.

Rabin, M. O. and Scott, D. 1959, 64
Finite automata and their decision problems.
IBM J. Res. 3, 2, pp. 115-125.
Also in Sequential Machines: Selected Papers, E.F. Moore (ed.)
Reading: Addison-Wesley (Mass.) pp. 63-91.

Reynolds, J. C. undated [before 1966]
A compiler and generalized translator.
Appl. Math. Div., Argonne Natl. Labs.,Argonne, Ill.

Ritchie, R. W. 1963
Classes of predictably computable functions.
Trans. AMS 106, pp. 139-173.

Rogers, H. 1967
The Theory of Recursive Functions and Effective Computability.
New York: McGraw-Hill.

Rose, G. F. 1964
An extension of ALGOL-like languages.
CACM 7, 2, pp. 52-61.

Rosenberg, A. L. 1967
A machine realization of the linear context-free languages.
Inf. and Control 10, 2, p. 175.

Rosenberg, A. L. 1967
Real-time definable languages.
JACM 14, 10, pp. 645-662.

Rosenberg, A. L. 1968
On the independence of real-time definability and certain structural properties of context-free languages.
JACM 15, 1, p. 672.

Rosenkrantz, D. J. 1967
Matrix equations and normal forms for context-free grammars.
JACM 14, 3, pp. 501-507.

Rosenkrantz, D. J. 1967
Programmed grammars -- a new device for generating formal langua
IEEE Conf. Rec. of 8th Ann. Symp. on Switch.+ Auto. Theory,pp.14-

Ross, D. T. 1963
On the algorithmic theory of languages.
Elec. Sys. Labs., MIT, Cambridge, Mass.

Ross, D. T. 1964
On context and ambiguity in parsing.
CACM 7, 2, pp. 131-133.

Scheinberg, S. 1960
Note on the Boolean properties of context-free languages.
Inf. and Control 3, 4, pp. 372-375.

Schorre, D. V. 1965
A necessary and sufficient condition for a context-free grammar to be unambiguous.
SDC Rep. SP-2153

Schutzenberger, M. P. 1961
Some remarks on Chomsky's context-free languages.
MIT Res. Lab. Elect. Quart. Prog. Rep. 63.

Schützenberger, M. P. 1962
Finite counting automata.
Inf. and Control 5, 2, pp.91-107.

Samelson, K. and Bauer, F. L.
Sequential formula translation.
CACM 3, 2, pp. 76-82.

Schutzenberger, M. P. 1963
On context-free languages and pushdown automata.
Inf. and Control 6, 3, pp. 246-264.

Schutzenberger, M. P. 1964
Classification of Chomsky's languages.
Proc. IFIP Working Conf. on Formal Lang. Descrip. Langs.,
Baden, Austria.

Schützenberger, M. P. 1968
A remark on acceptable sets of numbers.
JACM 15, 1, p. 300.

Schützenberger, M. P. and Chomsky, N. 1964
The algebraic theory of context-free languages.
in P. Braffort (ed.) Computer programming and Formal Systems,
Amsterdam: North-Holland.

Shamir, E. 1962
A remark on discovery algorithms for grammars.
Inf. and Control 5, pp. 246-251.

Shamir, E. 1965
Mathematical models of languages.
Proc. IFIP Congress 65, Wash. D.C.: Spartan Books, pp. 71-75.

Shamir, E. 1965
On sequential languages.
Z. Phonetik, Sprachwiss. Kommunikationsforsch 18, pp. 61-69.

Shamir, Eliahu 1968
A representation theorem for algebraic and context-free
power series in noncommuting variables.
Inf. and Control 11, p. 239.

Shannon, C. E. 1956
A universal Turing machine with two internal states.
Automata Studies, Princeton U. Press, Princeton, N.J., pp. 129-53.

Shepherdson, J. C. 1959
The reduction of two-way automata to one-way automata.
IBM J. Res. 3, pp. 198-200.

Shepherdson, J. C. and Sturgis, H. E. 1963
Computability of recursive functions.
JACM 10, 2, pp. 217-255.
Smullyan, R. M. 1961
Theory of formal systems.
Am. Math. Stud., No. 47, Princeton,U. Press.
Solomonoff, R. J. 1959
A new method for discovering the grammars of phrase-structure languages.
The Zator Co., Cambridge, Mass.
Standish, T. 1968
A preliminary sketch of a polymorphic programming language.
Comp. Sci. Dept., Carnegie-Mellon U.
Stanley, R. J. 1965
Finite state representations of context-free languages.
Quart. Prog. Rep. 76, MIT Res. Lab. Elect., pp. 276-279.
Stearns, R. E. 1967
A regularity test for pushdown machines.
Inf. and Control 11, 5, pp. 323-340.
Stearns, R. E. and Hartmanis, J. 1963
Regularity preserving modifications of regular expressions.
Inf. and Control 6, pp. 55-69.
Stearns, R. E., Hartmanis, J. and Lewis, P. M. 1965
Hierarchies of memory limited computations.
Proc. 6th Ann. Symp. Switch. Circ. Theory and Log. Des., IEEE, New York, pp. 179-190.
Steel, T. B. (ed.) 1966
Formal language description languages for computer programming.
North Holland Proc. of IFIP Work. Conf. on Formal Lang. Descrip. Langs.
Strachey, C. 1965
A general purpose macrogenerator.
Comp. Jour. 8, pp. 225-241.
Strand, Pavel 1968
On-line Turing machine recognition.
Inf. and Control 12, 5,6, p. 442.

Tarski, A. 1956
Logic, semantics, and metamathematics.
Oxford: Clarendon Press.

Turing, A. M. 1936
On computable numbers with an application to the Entscheidungsproblem.
Proc. London Math. Soc. 2-42, pp. 230-265.
Correction Ibid. 43, pp. 544-546.

Turing, A. M. 1937
Computability and lambda-definibility.
J. Symb. Logic 2, pp. 153-163.

Ullian, J. S. 1966
Failure of a conjecture about context free languages.
Inf. and Control 9, 1, pp. 61-65.

Ullian, Joseph S. 1967-8
Partial algorithm problems for context free languages.
Inf. and Control 11, p. 80.

Ullman, J. D. 1965
Pushdown automata with bounded backtrack.
SDC Rep. TM-738/022/00.

Ullman, J. D. 1968
Halting stack automata.
unpublished manuscript.

Wang, H. 1957
A variant to Turing's theory of computing machines.
JACM 4, 1, pp. 63-292.

Van Wijngaarden, A., Maillous, B. J. and Peck, J. E. L. 1968
A draft proposal for the algorithmic language ALGOL-68.
IFIP Working Group 2.1, MR 92.

Warshall, S. 196_
An algebraic language for flow charts.
Rep. No. TO-B-60-33, Tech. Operations Inc., Wakefield, Mass.

Warshall, S. 1961
A syntax-directed generator.
Proc. Eastern Joint Comp. Conf. 20, pp. 295-305. NY:MacMillan Co.

Warshall, S. 1962
A theorem on Boolean metrices.
JACM 9, 1, pp. 11-12.
Wells, M. B. 1964
Aspects of language design for combinatorial computing.
IEEE Trans. on Elec. Comp. EC-13, 4, pp. 431-438.
Wijngaarden, A. van 1962
Generalized ALGOL.
Proc. ICC Symp. Symbl Lang. in Data Proces. NY+Lond: Gordon+Breach
pp. 409-419.
Williams, J. 1968
unpublished
Comp. Sci. Dept., U. of Wisc.
Wirth, N. and Weber, H. 1966
EULER: a generalization of ALGOL, and its formal definition, I.
CACM 9, 1, pp. 13-23.

Yamada, H. 1962
Real-time computation and recursive functions not real-time computable.
PGEC 11, 6, pp. 753-760.
Yanov, Y. I. 1960
The logical schemes of algorithms.
Problems of Cybernetics 1, pp. 82-140 NY: Pergamon Press
Yasuhara, A. 1967
A remark on post normal systems.
JACM 14, 1, p. 167.
Yntema, M. K. 1967
Inclusion relations among families of context-free languages.
Inf. and Control 10, 6, p. 572.
Younger, D. H. 1966
Context free language processing in time n^3.
G.E. Res. and Devel. Ctr., Schenectady, N.Y.
Younger, D. H. 1967
Recognition and parsing of context-free languages in time n^3.
Inf. and Control 10, 2, pp. 189-208.

- - - - - - - - - -

Appendix II

NEW YORK UNIVERSITY
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N. Y. 10012
AREA 212 460-7100

December 1968

The following document (by C. Earnest) represents an effort (originating with John Cocke and Jack Schwartz) to initiate a discussion among a selected sample of experts concerning some of the principal issues in the design of commercial-grade compilers. It is hoped that, even when a consensus does not emerge from the discussion, strategic issues will at least be defined, directions for future research indicated, facts brought out, etc. Your frank, forthright opinion is solicited; in areas of controversy, the documents produced will be regarded as "debates on paper."

A first response is attached, and may help to indicate the style of response which would be helpful.
In addition to any questions raised by specific assertions in the enclosed document or by points of comment thereby suggested, we would like to have your opinion specifically on the following questions.

1. What quantitative information on compiler performance standards (size, speed) is available? Speed is probably best stated in terms of instructions per statement compiled, some indication of the amount of optimization performed and the general adequacy of the compilation being given. Bases for or reference for figures should be cited where possible.

   How are the overall data and space requirements of a given compiler affected by the size of the programs which it aims to compile, and by the complexity of the language to be compiled?

2. What proportion of compile time do various compilers devote to various principal subprocesses? What are the data space and code space requirements for the principal logical subparts of typical compilers?

3. What are the best compile-speed figures attainable by non-optimizing or only very slightly optimizing compilers? What are the best sizes attainable by such compilers?

4. What are the principal issues which arise in attempts to optimize the use of storage (automatic overlay planning, segment planning, etc., as distinct from optimizations which aim at reducing the number of instructions executed)? Do existing COBOL compilers (or possibly JOVIAL or ALGOL compilers) contain any useful ideas on this score?

5. To what extent should the machine on which a compiler is to be implemented affect its design? Is this merely a matter of core size and a few other overall parameters, or must much more detailed considerations concerning order code structure, storage hierarchy, etc., play a role?

6. What are the principal issues to be faced in the sectioning of a compiler into phases? How does the size of available core affect this question? What are the principal solutions employed in sectioning existing compilers? How is compiler efficiency affected by the need for sectioning?

7. Can anything about register use optimization be said on reasonably machine-independent basis?

8. What are the principal compiler variations occasioned by special features of the language to be compiled? In particular, do compilers for very large languages (like PL/1) have a fundamentally different structure than compilers for smaller languages, or are they merely expanded versions of the same sort of processor which would be used to compile a smaller language?

Note that the paragraphs of the following document have been individually numbered. This is intended to facilitate reference to particular points; we ask you in your comments to make use of these paragraph numbers where appropriate.

Courant Institute of Mathematical Sciences 12/68
New York University 10012

Comments on Industrial Compiler Practice

C. E. Earnest

I.(1) These comments outline some of the ways in which industrial compilers for higher order algebraic languages such as FORTRAN, Algol, and JOVIAL have solved, or attempted to solve, the various problems which complex design must face. The differences between academic, or experimental compilers as contrasted with industrial, or (usually) practical compilers are explored. Also discussed are the compiler-compiler approach, the features that make a good industrial compiler, and some actual speeds and sizes for existing compilers.

(2) Throughout, various existing compilers are cited as examples. Although some of the best compilers are mentioned, it is often true that a particular technique is illustrated best by an inferior compiler, so that citation in this paper should not be taken as a sign of excellence. We do note outstanding compilers.

II. Structure

(3) The compiling process, as understood currently consists of the following phases:

1. (FORTRAN only) Prescan. Most FORTRAN compilers include a prescan for each statement to determine statement type. For ASA Standard FORTRAN IV, this prescan can be done by a finite state machine: this technique is implemented in at least one existing compiler (CSC). For a FORTRAN which allows subscripted subscripts, the finite state machine must be supplemented by at least a nesting level counter. FORTRAN can be parsed without a prescan, but the added complexity of syntax would make the compiler bigger and slower.
2. Lexical Scan. The lexical scan picks the tokens out of the source string: reserved words, operation signs or words, identifiers, constants, and comments. The first three

categories are often handled identically by the token scan, except that reserved words and operation words or signs are pre-posted in the symbol table, along with any built-in function names the language may have; these are found and handled properly during a standard identifier search.

(4) One important function of the lexical scan is to reduce the size of the input string -- clearly this is accomplished by replacing identifiers, constants, etc., with pointers to symbol table entries. It is also done by causing strings of blanks to be represented by either zero or one blank (according to the language), except within character constants. It has been found that comparing each input string word with a word consisting of the code for a string of blanks causes an increase in compile speed of up to 1/4, especially in simpler compilers; this feature is found in most industrial compilers.

(5) We know of no American industrial compiler in which the lexical scan is a separate pass. It is always combined with the first pass syntax scan. Typically, the syntax scanner calls the lexical scanner to obtain the next segment of the string. In most cases, the lexical scanner is a hand-coded routine, usually in assembly code, based on a table lookup of characters in one or more tables (which table(s) depends on the current state), which implicitly "knows" the language being compiled. Thus it can throw away comments, recognize and convert constants, etc. In any compiler, this tailoring of the lexical scan greatly increases efficiency; in syntax directed compilers based on Chomsky context free grammars, the tailoring is mandatory in order that the compile speed be reasonable.

(6) There are exceptions to the above: An SDS 910 Algol compiler by Programmatics, and all of the GENESIS family of compilers by CSC, include a lexical scan which is at least partly based on a formal (analytic) grammar. Older compilers in the GENESIS family include part of the lexical scan in the normal first pass syntax scan -- viz., the recognition of constants, identifiers, and some reserved words. These compilers

do include a lexical scan, the state of which is set by the syntax scan (following along behind it).

3. Syntax scan

(7) The syntax scan accepts the source string, usually partially predigested by the lexical scan, and produces as output a code string in some form -- often reverse Polish. Whether the syntax scan is one pass or two depends on the language, the space available, and the quality of code desired. Algol, for example, is naturally a two pass language, because uses of identifiers can appear before their definitions (and the definitions, when they do appear, can even influence parsing); a single pass syntax scan is possible only at the cost of restricting the language, producing some rather bad code, or cleaning up the code later. When a scan is done in two passes, the first essentially processes name definitions, and the second produces code.

(8) FORTRAN and JOVIAL are naturally one pass languages.

(9) The syntax scan also produces almost all of the diagnostic messages produced by the compiler (they may be printed later, as in the CSC GENESIS compilers, the CSC Univac 1107 Fortran compiler, etc., but the information is developed during the syntax scan), and may make some error corrections.

(10) The process of parsing is by far the best understood part of compilers, and for that reason has received most of the attention in print. The syntax pass is built around a particular parsing technique, but the non-syntactic functions of the syntax scan usually require at least twice as much time and space as the syntactic functions, especially in compilers which produce fairly good code. We return to this subject later.

(11) The methods used for syntax scans in production compilers fall into three categories:

a) Standard, or Bauer-Samuelson scan. This method has been used since before 1960; evidently the earliest production compiler to use it was that for Burroughs 220 Algol, a version of Algol 58; the compiler was largely designed by Joel Erdwinn.

(12) Since then, the standard scan has been used in, among others, the Univac 1107 and 1108 FORTRAN compilers, the IBM 360 FORTRAN H and TSS 360 FORTRAN compilers. (It has also been used in assemblers allowing non-trivial expressions; for example, the SLEUTH II assembler for the 1107).

b) Scans based on Chomsky type 2 grammars. This type of scan has received most of the attention in the literature; usually some restriction of a context free grammar is allowed (bounded context, operator precedence, etc.).

(13) A number of compiler-compilers have been based on grammars of this type as well; most of them are not production tools, however. One which evidently has been used to produce a compiler, based on an explicit context-free grammar, is the TRANGEN system of Compass, Inc. Actually, this system is based more on Floyd productions than on context free grammars.

(14) Some production compilers have been built using a hand-coded version of this type of syntax scan; that is, the grammar is not explicitly part of the code for the compiler, but the structure of the scan is based on it. The Digitek FORTRAN compilers use essentially this technique. There is also a very poor CDC 1604 Algol compiler, written at Oak Ridge, based on a context free grammar technique.

(15) IBM for a while supported a project to produce compilers from context-free grammars. There was at least one FORTRAN compiler for the 7094 produced in this way, but as it was significantly slower and larger than other IBM compilers, it was never released.

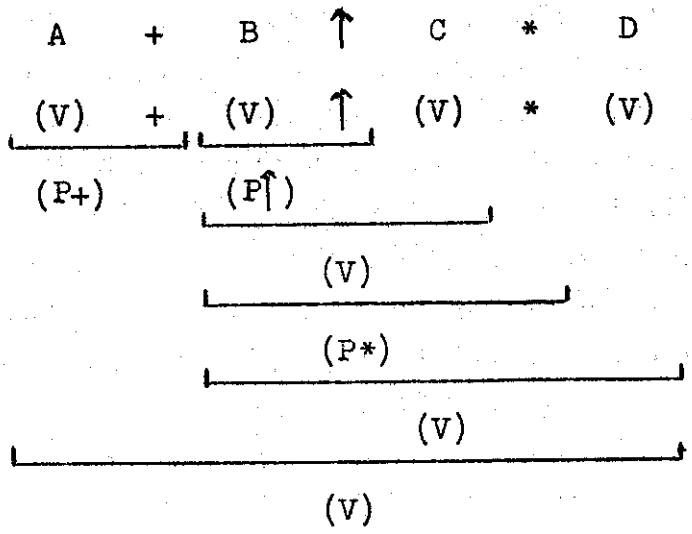
(16) There is also in existence a JOVIAL compiler for the 7090, written by SDC, which uses a matrix technique quite similar to an operator precedence grammar.

c) Scans based on analytic grammars. The only industrial compilers using these are the SDS 910 Algol compiler written by Programmatics, and the entire family of GENESIS compilers, produced by CSC, for Algol, Jovial, and a proprietary CSC language.

(17) The Programmatics compiler is based on syntax tables produced from a single syntax language by the use of assembler

macros. The CSC compilers are produced by a full-fledged compiler-compiler.

(18) This scan method allows speeds comparable to that of the standard scan. Some of the same techniques are used in writing the grammar as are used in a standard scan; for example, the typical analytic grammar expression scan is essentially a standard scan, except that the operator pushdown list is contained in the string being parsed. For instance, using (V) as the metalinguistic name of any expression or variable, and (P+), (P*), and (P↑) as the names of a pushed-down +, a pushed-down *, and a pushed-down ↑, respectively, a scan might proceed as follows:



Note that, if pointers to the variable entries in the symbol table are connected with the (V), the (P↑), the (P+) and the (P*) names in some way, this method of scanning allows trivial construction of expression trees directly from the scan.

4. Optimization phase

(19) Many industrial compilers, especially those for small machines, omit this as a separate phase. Either very little optimization is done, or it is done locally and the phase is combined with the syntax scan and/or the code generation phase. With careful attention to machine details, and with a simple

register memory during code generation, it is possible to produce quite reasonable code without a separate optimization phase. All the early Digitek compilers and some of the current ones, including the off-the-shelf-FORTRAN, do not perform global optimization; this is true of the first crop of GENESIS compilers. A number of less successful compilers also omit the optimization phase; e.g., the CDC 1604 Algol compiler, and most of the SDC JOVIAL compilers.

(20) Techniques used for global optimization, and the exact set of optimizations performed, vary widely. Roughly, one can say that common sub-expressions are eliminated in certain places, loop constant expressions are removed from loops, computations in loops are reduced in strength, etc.

(21) The first optimizing compiler was of course FORTRAN I, written by a team at IBM; it contained the seeds of a great many good ideas, and was quite remarkable for its time. The best code currently produced by a compiler probably is produced by the IBM 360 FORTRAN H compiler. The TSS FORTRAN compiler, using a different technique, runs a close second, and is much faster (originally 4-5 times faster). An optimizing compiler need not run slowly. Illustrations of this point are furnished by the Univac 1107 and 1108 FORTRAN compilers; these compile at approximately 14,000 instructions executed per source card, including input/output. An even better illustration is furnished by the TSS FORTRAN compiler (360), which runs at approximately 8,000 instructions executed per source card (CPU time only).

5. Code generation phase

(22) The code generation transforms the internal form of the code, using the symbol table, to some representation of actual machine instructions. Except for the symbol table, only local information is used. The internal forms used may of course be only a sequence of calls on routines in another phase.

(23) Code generators typically consist of a rather large number of individual generation routines, corresponding to the operators in the code generator input string. Probably the hardest task

of a code generator is to assign the machine registers, and generate necessary temporary storage cells. This task may be made easier by information collected earlier by an optimization phase.

(24) For some small machines, of course, register assignment is a simple problem, because there are only 1 or 2 registers available; the Digitek SDS 910 FORTRAN compiler, for example, has only an immediate register memory for the machine accumulator. Most industrial compilers by now incorporate a register memory, destroyed at code entry points, for the machine accumulator and index registers. A commonly used technique for choosing a register for a new quantity is to pick an empty register if possible, or to destroy (with a possible store) the contents of the register containing a quantity which is used farthest away in a forward direction; the Univac 1107 FORTRAN compiler, among many others, uses this method.

(25) Much of the most important object code optimization is accomplished during the code generation phase. This includes a whole bag of very locally applicable coding tricks: deletion of an add of zero, or a multiply by 1; replacing a division by a constant with a multiplication by the reciprocal; converting constants to the correct type; replacing a multiply or divide by a power of two by a shift; and innumerable others made desirable by the quirks of a particular machine. Most of the better code generators also expand built-in subroutines, including those for some type conversions, in-line.

(26) Code generation is typically quite fast, because of its local, non-combinatorial nature. This is not true in the CDC 6600 FORTRAN compiler, however, because the code generator also schedules the instructions in an attempt to allow the machine maximum overlap when the code is run; this process is highly combinatorial. Other fourth generation machines will require compilers with the same characteristic. Also, if a very careful job of register assignment is done, the code generator will be slowed down quite a bit.

6. Output Editing

(27) This phase produces the actual binary load modules containing the object code, and may produce an assembly style listing of the generated code. The editing phase also carries out the assembly process -- substituting addresses for symbol table pointers -- on the code.

(28) Some compilers actually use the standard assembler for the machine to accomplish the output editing; that is, the compiler produces actual card images and inputs them to the assembler. For experimental compilers, this technique is often quite useful; for production compilers, it is not to be recommended, because of the speed sacrifice. The IBM 7094 FORTRAN compiler used this technique, as did the later BOS FORTRAN compiler for the 360.

(29) In a few compilers -- notably those of the GENESIS family produced by CSC -- the output editor also allocates the storage for object program data, in a fashion strictly determined by earlier processes, and passed to the editor as a system of pointers and sizes. This approach is useful for compilers which produce code for more than one object machine.

(30) The editor may also merge diagnostics produced by several phases together with the source string, and possibly an assembly listing of the generated code. Most compilers, including all IBM industrial compilers, produce either no or an extremly unreadable listing of the generated code. Exceptions to this are the Univac 1107 and 1108 Fortran compilers, and the family of GENESIS compilers.

(31) Editing is, rather surprisingly, somewhat slow, especially when a full output listing is produced. This is evidently due to the fact that each generated instruction requires quite a bit of attention, and that most machines on which compilers run are ill-suited to the character and bit string manipulations that an editor must perform.

(32) The editor of course also produces any debugging symbol tables usable by the system, and sorted symbol dictionary listings and maps.

III. Storage allocation and file handling in the compiler

1. Symbol tables

(33) Almost all commercial compilers use symbol tables based on a hashing search technique. The hash table is typically separate from the data storage area, and links are used for the hash chains -- this allows simple allocation of the space in the data storage area. In an optimizing compiler, the same hash technique is often used to find formally common sub-expressions as is used to find identical names or constants.

(34) There are exceptions to the above: Digitek compilers use a tree storage method for source names. A single list (stored contiguously) contains all the characters which have occurred as the first character in a name, and for each a name or location of a sublist containing the second character which followed that particular first character in some name, etc. Some Jovial compilers provide for a "compool" --that is, a common pool of symbol and other information -- which is larger than the compiling machine can contain in core. The program is first scanned, and all names not defined within the program (definitions in the program take precedence) are collected, sorted lexicographically, then compared against the previously sorted list of compool names on a tape -- thus the size of the symbol table is determined by the names actually used in the program, and only one pass over the tape is required.

(35) Except for the above-mentioned Jovial technique, symbol tables are almost always kept entirely in core throughout the compilation, and data is not moved within the table, so that pointers to the data can be kept easily. An exception to the latter part of the statement is again provided by the Digitek compilers -- see below.

(36) For languages having more than one scope of definition for names, a number of different techniques are used. Jovial and some FORTRAN's (having subroutines included within the main program) have only two scopes: main program or subprogram. This is usually handled by posting inner scope names so that the search process finds them first, and by deleting all names

defined in the inner scope at the end of that scope. Often the restriction is made that a previous scope -- even an outer one -- is not resumed once an inner scope has been entered; in this case, deletion of the inner scope names is not necessary.

(37) Another technique,* useful where it is inconvenient or expensive to delete names or to change chain pointers for names in an inner scope, is to keep with each symbol table entry a scope begin number = the number of the begin scope, numbering consecutively from the start of the program, within which the particular entry is defined. A single bit for each different scope range is also kept; these are initially all zero, and are set to 1 individually as each individual scope closes. The "within scope" check for a name is then that the current scope begin number is greater than or equal to the stored scope begin number, and that the end marker for the stored scope begin number has not yet been set. Forward referenced labels present a further problem: this can be solved by assuming that each label is defined at each scope (creating entries as necessary when a label is referenced), and that some inner labels have the same address as corresponding outer labels (a label entry is set to point to the corresponding entry for the next outer level when a scope is exited and no definition for the label has occurred).

(38) Equivalence or overlay information is another important detail, the solution to which is not particularly enlightening, and which varies from compiler to compiler. Some system of links is typically used.

(39) Most of the information stored with a symbol entry is of fixed length, so that a standard amount of storage can be reserved at the time the entry is made. Exceptions to this are the characters of the name itself (except in FORTRAN compilers) and such things as dimension information. Some compilers keep separate name tables, with pointers to attribute information; this has the disadvantage of splitting the available

---

* used in the GENESIS compilers.

space into two parts. Another technique is to store the name just before the attribute information in core, keep as one attribute the number of characters (words) occupied by the name, and reference the entry by a pointer to the start of the attribute storage -- this permits rapid (non-variable length) access to attributes, and also allows slightly slower access to the name itself from the attribute part.

(40) For languages which allow a small maximum number of dimensions for an array, some compilers allow space for the maximum number in every array entry. FORTRAN has the problem that dimension information may occur after the first reference to a name, so that allowing space for exactly the correct number of dimensions when the symbol table entry is first made is not always possible. Algol has the same problem, although more locally. Therefore dimension information is often stored remotely, and pointed to from the name. One nice technique, used in the GENESIS family of compilers, which is convenient for both constant and variable dimensions, is to store the information as a Polish string (remote from the variable entry) which can be appended to the actual subscripts at reference. For example, for a two dimensional array A, declared in Algol as array A [LB1:UB1, LB2:UB2]; the stored Polish string would be:*

LB2, -, UB1, LB1, -, 1, +, *, + , LB1, -, ↓

At an actual reference, e.g. A[I,J], the names or expressions A, I and J would simply be followed by the stored string in the code output, to produce:

A, I, J, LB2, -, UB1, LB1, -, 1, +, *, +, LB1, -, ↓

---

*More generally, the stored Polish string for an n-dimensional array is:

$$\overbrace{LBn,\ -,}\ \overbrace{UB1,\ LB1,\ -,\ 1,\ +,\ *,\ LB(n-1),\ -,}\ \overbrace{UB2,\ LB2,\ -,\ 1,\ +,\ *,\ LB(n-2),\ -,}\ \overbrace{\cdots\ \cdots}\ \overbrace{UB(n-1),\ LB(n-1),\ -,\ 1,\ +,\ *,\ LB1,\ -}$$

In non-Polish, this is:

$$A\ [(\ I + (J - LB2) * (UB1 - LB1 + 1)) - LB1]$$

Of course, the actual subscripts and the upper and lower bounds can be expressions or variables or constants in this representation.* This permits consistent handling of static and dynamic arrays at reference, and provides extremely rapid generation of code at reference. This technique does not work quite as well for FORTRAN because of the necessary reversal of the actual subscripts.

(41) For compilers running under time sharing systems where space is charged for in increments of pages, it is important to use as few pages as possible -- that is, not to have a number of pages part full if fewer almost full pages would do. The IBM TSS/360 FORTRAN compiler (written by CSC) has been planned with this in mind.

2. Internal table storage

(42) Except for the symbol table, and input and output files, tables in a compiler are usually fixed length. One noteworthy exception to this is the storage scheme of the Digitek compilers: each table is referenced every time through a pointer, so that tables can move freely. Whenever the contiguous space for one table is used up, storage is reallocated and data moved, if possible, to create more contiguous space for the full table. This has the advantage of making maximum use of the available core -- which for Digitek compilers is typically very small. It has the disadvantage that when the available space becomes nearly full, much time is spent reallocating. The extra time required by the extra level of indirectness is also an added cost, but a minor one; because tables are referenced essentially interpretively (see below), there is little extra code space required.

---

* constants may be combined if desired, of course.

(43) Every commercial multi-pass compiler uses the same sections of core for different programs or tables in the different passes. Most operating systems provide for code overlays in this fashion; one of the most flexible schemes is provided in the Univac 1107 and 1108 systems.

3. Input/Output files

(44) Typically, the source string, a scan or construct string (for two pass syntax scans), various intermediate code strings, the final binary output string, and the listing file, if any, are I/O files, at least on compilations too large to fit into core. Of course, not all compilers have all these files -- the Digitek compilers have only an input source file, and an output binary file and print file (they are one pass).
(45) Optimizing compilers often keep the intermediate code string in core between the syntax scan and the optimization pass; one convenient form for expressions, for example, is trees, which do not lend themselves to easy input and output.
(46) A technique used in the GENESIS family of compilers is to provide rather large buffers for each of the intermediate files, so that they can be kept entirely in core for small compilations. If the file does become too large to stay in core, the first buffer is kept in core to cover the tape rewind time at the start of the next phase -- all other buffers are written on tape. This technique is also used in at least one SDC JOVIAL compiler.

IV. Flexibility; debugging of the compiler

1. Flexibility

(47) Except for the smallest commercial compilers, wherein space constraints determine almost the entire plan, a reasonable amount of space and speed is sacrificed in order that the compiler be debuggable and that it be amenable to reasonable change.

(48) One of the most important contributors to this flexibility is a clean interface between each of the various phases. That is, the number of tables, files, etc., should be kept to a minimum, and the implicit information in each should be kept to reasonable limits. For example, in the Univac 1107 FORTRAN compiler, and in the entire GENESIS family of compilers, the only information passed from the syntax phase to the code generation or optimization phase is the symbol table, the code string (in intermediate format), and a few parameters (like 5-10). This approach is carried to an extreme in the IBM 7094 FORTRAN compiler, where only the assembly format card file is passed between the syntax phase and the code generation phase. The reason that clean, explicit interfaces cost time and space is obvious: the amount of data passed is relatively large, and it is sometimes necessary to go to some work to discover information in a later phase that was much easier to find in an earlier phase, but which could not be or was not included in the explicit interface neatly.

(49) Machine or language independence of the compiler adds greatly to the interface problems -- see the discussion of GENESIS compilers below.

2. Debugging

(50) Every good commercial compiler of medium to large scale includes within itself some tools used only for the debugging of the compiler itself.

(51) For example, the Univac 1107 FORTRAN compiler recognizes some unpublicized "reserved words" as legal statement types; these cause a trace of a following statement, as it is processed by the compiler, to be printed.

(52) The GENESIS compilers include routines to produce a core dump listing in relocatable format, using the names of routines and tables which are part of the compiler. They also include a formatted symbol table dump, which among other things, prints the names as character strings; and a formatted Intermediate Language dump, to list the intermediate format of the code string in a fashion most readable to a compiler writer. All these routines occupy part of the space allocated for the symbol table, so that when the compiler is operating in non-debug mode, no space is required -- the routines are merely written over by data storage. An octal correction facility for the parts of the compiler itself is also included. The choice of which, if any, debugging tools are to be activated is indicated in the normal option list on the card which calls the compiler into operation.

(53) The IBM TSS/360 FORTRAN compiler also includes some of the same types of tools.

(54) See also the next section.

V. Compiler-compilers

(55) So far as we know, the only compiler-compiler worthy of the name which is being used to produce commercial compilers is CSC's GENESIS system. Other compiler-compilers either do a small part of the job (e.g., build syntax tables only), or they produce compilers which are toys. Since GENESIS is reasonably successful, but does have some problems, it is worth some discussion.

(56) GENESIS consists of a large system which accepts the source for a language specification, makes a number of theoretical and practical checks on it, produces tables and code from it, and provides for debugging of it within the GENESIS system. In addition, GENESIS consists of a compiler structure plan in some detail, an Intermediate Language design, and a number of code generators, optimizers, and editors for a number of machines. The grammar used to express the syntax of a language is an analytic one; the pragmatic functions (e.g., symbol table posting and searching, code production) are in the first generation of GENESIS written in a special interpretive language -- in the second generation in a procedure oriented language (SYMPL).

(57) The debugging tools included in the GENESIS system include symbol table and IL dump routines -- callable at any time, and a syntax level trace of the recognition process (turnable on and off), as well as the logical consistency checks made automatically on a language specification.

(58) The output of GENESIS is a set of binary tables in the format of the compiling machine system, and the code in the same binary format (as well as several kinds of listings).

(59) GENESIS has by now been used to produce Algol compilers running on the IBM 7094, the Univac 490 and 494, and the GE635. The compilers running on the 7094 and the 490 each produce output code which runs on these two machines, and on the CDC 3600 and the SDS 910. The 7094 and the GE 635 compilers also produce code which runs on the GE 635. A Jovial compiler running on and producing code for the IBM 360, which includes

an optimizer, is in the latter stages of checkout. PL and FORTRAN compilers are planned. To give an indication of the machine independence of the language specification, be it noted that the first two passes (the syntax scan, including code production, etc.) of the 5 compilers running on the IBM 7094 are identical, in source and in absolute binary, for all 5 compilers. A single language specification (source and binary) serves for all five.

(60) The GENESIS compilers -- particularly those of the first generation, which include a large amount of interpretive code -- are not the most rapid available: the 7094 compilers run at about 137,000 instructions executed per source card including all I/O, etc. The new compilers are faster; the Jovial compiler runs at 100,000 i.e./s.c. on the Model 65; a SYMPL compiler runs at 84,000 i.e./s.c. on the 1108.* The space required is about average for a medium scale compiler, but a hand-coded compiler with similar features could probably be written to occupy 1/2 to 2/3 the space occupied by the GENESIS compilers. Improvements could be made to increase the speed almost to that available from good hand-coded compilers; the space required could not be reduced much more than at present without a large speed sacrifice, or loss of modularity.

(61) Somewhat surprisingly, the Intermediate Language has not created many problems, except those normally associated with explicit interfaces of any kind. Although most compiler theoreticians have convinced themselves that a general Intermediate Language is impossible, the fact remains that a practical one, requiring only minimal change to accommodate a new machine or source language, is quite possible (at least for the Algol, FORTRAN, Jovial, PL/1 class of languages).

(62) The biggest problem with GENESIS is proving to be standardization. Any system which supports a number of compilers cannot be allowed to change without careful

---

*this compiler generates code for the 60-bit 6600, which means editing takes 60 to 70 percent of the time.

consideration of the consequences for existing and future compilers. There are already two major versions of GENESIS--the second incorporating a number of major improvements -- as well as a few minor wrinkles introduced to adapt the system for a particular compiler without careful enough consideration. (63) GENESIS does not permit the same kind of easy definition of a new language that, say, a university compiler-compiler system might. This is mostly because it is aimed at producing commercial compilers, and this is not an easy job in general (see the discussion below). GENESIS does make implementing a compiler a quicker and more reliable job than hand coding allows; it provides for greater compatibility between compilers; and significantly eases the maintenance of a compiler.

VI. The Digitek method

(64) Digitek provides off-the-shelf FORTRAN compilers, particularly for small machines, which are probably the easiest compilers to move from machine to machine in existence. The compilers are not fast, do not produce verbose diagnostics or code output listings, and do very little optimization. They are extremely small, compile all of FORTRAN plus some sensible extensions, and as mentioned, are easy to move to a new machine.

(65) An attempt was made, jointly by Digitek and PRC, to use the same technique to implement PL/1, with notable lack of success. Evidently the technique is best for small languages.

(66) The compilers are based on a set of POPs - Programmed Operators. Each POP is essentially an instruction in a higher level machine -- the POPs are interpreted, fairly efficiently. Each POP typically consists of an operation to be performed and the name of a list involved, and possibly a skip flag. The list names are consecutive integers, used to index into an address table containing pointers to the lists (see Internal Table Storage, above), so that the field required in the instruction is quite small. The 360 POP set requires only 2 bytes per POP, including the list name. Almost all lists are pushdown lists, so that subscripting is usually not necessary, and so that recursive techniques may be easily programmed. Included in the interpreter for the POPs is a storage allocation routine which uses every last word of available space for the lists, if necessary (see Internal Table Storage, above).

(67) The reason that the compilers move so easily to a new machine is that most code is written in POPs, so that only a new POP interpreter (or part of one -- the interpreter itself may use POPs) is necessary, plus new code generation POPs for the new machine. Of course, the compilers are relatively small, and the techniques are quite well understood by now, which helps.

VII. The language in which a compiler is written

(68) Until fairly recently, all commercial compilers were written in assembly code, except for the SDC JOVIAL compilers. The best compilers, from a user point of view (Univac 1107 and 1108 FORTRAN, IBM 360 FORTRAN H and TSS FORTRAN), still are. FORTRAN H for the 360 was originally written in an extended FORTRAN, but was recoded in assembly code for system release 14; the assembly code version runs much faster than the older version, although it is full of bugs.

(69) The SDC Jovial compilers written in Jovial were very large and very slow-- partly because the art of compiler writing was not as well understood then as now, and partly because of the inefficiency of the Jovial code. These compilers included very little optimization, so the compiler code itself was poor.

(70) It is becoming possible to write a compiler in a higher order language with little loss of speed. The GENESIS compilers are written in either Algol or SYMPL. Even though the optimization for these compilers is of the local variety, speeds are quite reasonable. Newer compilers, compiled through an optimizing compiler themselves, are faster, but are not yet released. There are also one or two existing Jovial compilers (one by SDC, one by CSC) which are written in Jovial, compile most of the language, are not globally optimized, which run quite fast. The CSC Jovial on the 1107 (rather experimental, and somewhat rudimentary) runs faster than 10,000 instructions executed/source card.

(71) It must be stated that the biggest improvements in speed of compiler code written in a higher order language come from local optimizations, including a rather careful use of machine facilities and a reasonable register assignment scheme, and from the careful coding by the programmer to avoid inefficiencies which the compiler cannot remove. Global optimization will allow further improvements, and a sophisticated register assignment scheme even more.

(72) Unfortunately, none of the standard existing higher order languages are well-suited for writing compilers. The desirable features of a language in which to write compilers are:

1. Access to machine part words as variables in the language. This has implications for the object code data storage allocation as well as for the code production.
2. Some form of based storage. Using array subscripts as pointers is possible, but awkward, and it may lead to some unnecessary machine word size dependence. This is particularly true with the newer machines, which have several different sizes of data which are easily accessed.
3. Multi-level scope of definition of names. This permits combining separate routines written by different people with little trouble.
4. Efficient calling sequences, either as standard in the language, or as an alternate to the standard. The ability to leave arguments in registers is very valuable, even at the cost of extra declarative information required from the programmer.
5. Some access to machine instructions, either by subroutine call, or as built-in language functions. This point combines with point 4.
6. Data structures and allocation rules which permit efficient (not variable at many levels, as in PL/1) access to data, control over placement of variables in memory, including control of overlays, and the combining of heterogeneous types of variables in a single structure or entry (as in a symbol table).
7. Some provision for recursive routines. This is not absolutely necessary, but is very convenient, as many compilation processes are naturally recursive. Again, extra declarative information -- e.g., which routines are recursive, which variables are to be stacked -- may be required in the language without reducing the effectiveness of the tool much. Recursion is becoming quite cheap, with many machine registers, re-entrant code, etc. This point also combines with point 4 very strongly.

8. Some rudimentary macros and conditional compilation tools. Something as simple as the JOVIAL DEFINE is useful; statement functions which are compiled in line at reference, as in the Univac 1107 FORTRAN compiler, are better and usually sufficient. Conditional compilation allows the easy adaptation of the same source code to several different machines, for example.
9. Some provision for initialization of variables.
10. Some provision for at least rudimentary handling of character and bit string (of word length or less) variables and constants. In addition, octal (not -- repeat, not -- hexadecimal) constants are extremely handy to have.

(73) A number of other features would be convenient, but not strictly necessary. An example is the status variable in JOVIAL and SYMPL, and the status switch in SYMPL. It is necessary in general to have both real and integer variables and expressions, but the real variables are used very seldom--only in input/output conversions and in constant combination or generation in the code generation phase -- so that if a library of routines is provided, real arithmetic is not strictly necessary in the compiler language.

(74) PL/1 probably has more of the above features than any other language, but in particular it does not have 4, 5 or 6. In addition, it has so many features which are essentially useless in writing a compiler that much unnecessary inefficiency is unavoidable. Jovial, FORTRAN, and Algol also each have some of the required features, but not all. SYMPL is probably the best existing choice, but still does not have feature 7, and has a weak form of feature 8.

VIII. Differences between Commercial Compilers and Toy Compilers

(75) We will list the features of a "good" compiler in each field, which are different for the two types:

Commercial Compiler

1. Well-integrated with the machine operating system. This includes a good correction facility for source code (the IBM 7094 and 360 notwithstanding), intelligent use of I/O, careful use of storage space, and control statements consistent with other system compilers (this last is often surprisingly hard to accomplish).
2. Well-debugged and well-documented.
3. All features of a language implemented.
4. Informative diagnostic messages, which catch most of the most common programmer errors, and which do not require many passes over the source to catch all existing errors (if possible, of course).
5. Output code is reasonably efficient.
6. Full listings, including such things as sorted symbol dictionaries, cross-reference listings, allocation tables, etc.
7. Easily maintainable. Debugging tools are built into the compiler, and the coding practices are such that changes do not upset the whole compiler (if possible).

(76) The only real feature that a toy compiler, or a toy compiler-compiler has which a commercial compiler does not is a clarity and consistency of design from a theoretical point of view. This in turn permits rather large changes to be made in the language, or new languages to be added, fairly easily.

IX. Implementation Problems for a Commercial Compiler

(77) It is well known that production of a commercial compiler takes much longer and is much more expensive, as a general rule, than the production of a toy compiler. It is worth noting the reasons for this, because they are not the obvious ones.

(78) Probably the biggest problem in implementing a commercial compiler is in fitting it to the existing machine operating system. No existing operating system is well enough documented or debugged to permit adding a nontrivial compiler without a fair amount of trial and error to discover just how to make the system do something close to what is desired; this is true in particular of relocatable loader input formats. Other trouble areas are system I/O, and overlay handling. Of course, if the system is small, or if no attempt is made to use it well, this problem largely disappears. The former condition applies to most of the systems within which the Digitek compilers live, which explains part of the ease of moving them. The system is not used well if the compiler produces an assembly input file, as do many toy compilers; this does avoid the relocatable format problem handily, though. This overall problem of system compatibility cannot be over-emphasized, and must be experienced to be believed. It behooves any implementation project to assign at least one man from the start to learning to live with the system. Note that for languages with I/O, the system interaction problem also exists for the object time library.

(79) The other large problem in implementing a commercial compiler is usually that of exact language definition. Often a new machine will necessitate changes in an existing language -- cf. 2 byte integers in 360 FORTRAN IV. Another problem, often overlooked, is what to do about incorrect code; if an addition to the languge is made to make a commonly made error legal, then compatibility with other past and future compilers for the same language may be affected. Further, particular conventions with regard to definition points for

common variables in FORTRAN, for example, may hurt optimization in the future considerably. Calling sequences and storage allocation rules must be designed very carefully, as these usually affect the definition of the language itself. The entire problem of exact language definition is likewise always underestimated, and must be experienced to be appreciated. Of course, for toy compilers, the decisions convenient for the moment are made.

(80) There are some other more obvious reasons why a compiler is a major effort. The object time library must usually be written and debugged (a toy compiler can usually use an existing one). All the aforementioned points relating to clean interfaces, flexibility, maintainability, etc., cost time in implementation.

Appendix (I) A Summary of Speed and Size Data for a Few Principal Compilers.

(81) Machine speed figures used in converting source lines per minute into instructions executed per source card:

Machine	Instructions per Sec.
7090	400,000
7094	800,000
360 mod 65	800,000
1107	220,000
1108	1,200,000
6600	2,500,000
1604	100,000
3600	300,000
625	?

(82) Note that for all compiler speeds given, the actual measurement was in cards per minute. The figures for different compilers are not really directly comparable, because some include I/O time and others do not. Note also that FORTRAN is a simpler language, and typically has less on a card than Algol, JOVIAL, or SYMPL, which partly explains the fast FORTRANs.

Compiler Speeds, Sizes

(83) TSS FORTRAN (machine code)

CPU time: 6000 source lines/minute on mod 67

! ⟶ approximately 8,000 instructions executed/source card

Percent of time in each phase (approx.):

scan (includes very careful diag., user correction interaction)	30 %
optimizer (includes global register assign.)	28 %
code generation (includes local reg. ")	28 %
edit	5 %

Sizes

scan	20,000	lines of code (assembly code)
interlude	2,000	"
optimization	10,000	"
code	30,000	"
edit	5,000	"
Total	67,000	lines of code

1107 ALGOL (GENESIS - scan only) -- space required

(84) Language Specification

Tables	Pass 1	Pass 2
Lexical scan, misc.	700	300
Syntax trees	1570	2170
Pragmatic functions	1360	1475
	3630	3945

(85) Code to Interpret the Above

Control, space handling, misc.	750
Lexical scan	300
Syntax	700
Pragmatic functions	3700
	5450

(All in machine words occupied (36 bit words).)

(86) FORTRAN H

(Old version, written in FORTRAN)

CPU time: 1300 source lines per minute on mod 65

Approximately 37,000 instructions executed/source card

(87) GENESIS Compiler

7094 Algol

350 cpm (mod II) = 137,000 i.e./s.c.

360 JOVIAL

500 cpm on mod 65 (including I/O, no optimization)

Approximately 96,000 i.e./s.c.

6600 JOVIAL

1500 cpm on 6600

Approximately 100,000 i.e./s.c.

GE 625 Algol

315 cpm = ?

1108 SYMPL

860 cpm = 84,000 i.e./s.c.

(This compiler produces 60 bit output for the 6600, which means that editing takes 60-70 % of the time.)

Speed breakdown per pass

scan	29 %
object allocation	0.5
code generation	1.5
editing	69.0

If we replace the editing times to be approximately equal to scan times, as is normal in other 1 pass scan GENESIS compilers, we get approximately (and fictitiously):

1600 cpm = 42,000 i.e./s.c.

(88) Non-GENESIS compilers

SDC J-3 JOVIAL (in JOVIAL)

1604	110 cpm	=	55,000	instructions executed per source card
3600	311 cpm	=	55,000	i.e./s.c.

SDC J-2 JOVIAL (in JOVIAL)

7090	130 cpm	=	184,000	i.e./s.c.

SDC J-S JOVIAL (one pass) (in JOVIAL)

7090	200 cpm	=	120,000	i.e./s.c.

CSC 1107 J-S JOVIAL (internal) (in JOVIAL)

1107	1300 cpm	=	10,000	i.e./s.c.

1107-1108 FORTRAN (machine code)

900-5000 cpm		=	14,000	i.e./s.c.

(includes optimization, I/O)

Appendix (II.) Register Allocation in TSS FORTRAN

(89) Up to 8 "global" registers are assigned from the innermost loop out. Which variable should be in a register is decided by a weighted "popularity" count for expressions, adcons and variables. All other registers are assigned locally as needed. Evidently little or no flow tracing is done, although the information about what was moved, reduced in strength, etc., is available.

Comments on Jack Schwartz' questions by C. Earnest

RE

1: Probably the most meaningful compiler speed figure is instructions executed/source card (or statement) compiled. A bad compiler may generate voluminous amounts of code, so instructions executed/instruction compiled is not so good. A very rough conversion factor between these ratios is 3-5 instructions generated/source card.

4: I don't know much about this. Of course, JOVIAL and ALGOL compilers can and do overlay local storage blocks in different scopes.

5: Given its gross description, particular details concerning a compiling machine affects compiler design only slightly. The most significant parameter is core size, which determines how many passes must be used, how many luxury features can be included, etc.; size constraints can require the use of an entirely different technique if severe enough (e.g., Digitek FORTRAN), or permit otherwise impractical techniques if large enough (e.g., optimizing compilers which represent expressions by trees kept in core throughout the compilation process).

The other main factor to be concidered is the operating system under which a compiler is to run. In TSS, the storage layout chosen for the TSS FORTRAN was suggested by the attempt to make maximum use of a page once any part of the page was used (as mentioned elsewhere). The operating system under which a compiler is to run also affects the details of its file handling significantly.

As an illustration of this entire point, note that the GENESIS compilers are all identical in gross design, number of passes, number of files, etc. GENESIS compilers have been written so that the last syntax pass and the code generation or optimization pass can run as one pass if enough core is available, but none of them actually run in this way (in fact, changing from one style of segmentation to another amounts to replacing file input and output handling routines by a bilateral linkage routine).

Of course, the details of the machine being compiled affect the code generator, and the optimizer, but, for current machines, not very profoundly. Some nasty detailed problems arise in, for example, the 360 (cover registers, calling sequences) but these are not of general interest.

6: This varies greatly. Digitek compilers are one pass, with no overlays. Most compilers follow the following outline of phases for the passes:

1) Prescan, lexical scan, pass 1 syntax scan
2) (Pass 2 syntax scan)
3) (Optimization phase)
4) Code generation
5) Editing

The only things that carry over from pass to pass are the compiler control and I/O programs, and the symbol table. Buffers may also be kept (see I/O files).

8: There is really nothing general I can say about this. Certainly smaller languages can use simpler techniques (FORTRAN compilers can assume 6 characters for identifiers, for example), and more complex languages are a lot more work and may require fancier scans.

Probably most of the difficulty with designing and writing a compiler for a big language -- aside from the obvious amount of work required to implement all the features -- comes in the area of language design, and understanding just how the features interact with each other.

Comments on Paper by C. P. Earnest -- Anon.

Most successful commercial compilers seem to be of one of three types: a) the directly hand-coded compilers making maximal use of hand code optimizations and compiler tricks, b) the CSC GENESIS-produced compilers, c) the Digitek Corp.'s "POP-method" produced compilers. The following comments principally concern GENESIS compilers.

The GENESIS compiler generating system seems to be capable of producing compilers whose performance belongs to the presently acceptable commercial range. On the other hand the system does not presently produce compilers that really rate with the fastest hand-generated compilers. Roughly, a hand-coded compiler will out-perform a GENESIS compiler by a factor of two in compile speed. Moreover, the GENESIS compilers seem to be somewhat bulky, partly because of the logical compromises necessary to secure the machine independence at which these compilers aim. The bulk ratio (GENESIS compiler/hand-produced compiler) seems to be fairly represented by a factor of two or three. In particular, GENESIS produced compilers seem to be doubtful candidates in any situation in which a compiler must run in less than 32,000 -- 32 bit words is required. In this design range, only the hand and the Digitek-type compiler seem to be candidates.

Part of the time-space problem noted above seems to be a result from the fact that the SYMPL language used within the GENESIS system for the SYMPL language used within the GENESIS system for the expression of semantic generators is compiled without any substantial optimization. This seems to result in a bulk of machine code which is about four times the size of hand code written to express the same function. It might be expected that application of an optimizer to the SYMPL intermediate code would substantially increase the quality of GENESIS produced compilers. However, unless a very sophisticated optimizer were designed, some residual loss could be expected due to the requirement of machine independence inherent in the GENESIS system. It would be quite interesting to estimate the relative part of present inefficiency attributable to the two causes in machine independence versus non-optimization.

Some of the specific details of the GENESIS generator language seem somewhat primitive. In particular, the code generation procedure could use considerably more detailed structuring, in the style of the "bit strip" method used in the IBM Fortran H compiler. The lack of such a system feature makes the GENESIS generator subroutines extremely large. The GENESIS system would also benefit considerably from a more careful and systematic treatment of the register assignment problem, and, in particular, lacks a reasonable algorithm for the assignment of registers in program regions.

Some of the special features of the GENESIS system do seem quite interesting. GENESIS seems to represent one of the most successful efforts to limit the effects of machine dependence on compiler writing, and in particular, is a definite advance over prior efforts like the SLANG effort at IBM and the NELIAC effort of the San Diego Naval Electronics Lab. The use in GENESIS of multiple scan cursors operating in tandem seems to be an interesting special feature.

It seems that the intermediate language structure adopted by a compiler can have considerable effects on the overall success of the compiler design. This is an area that has been insufficiently thought out, especially in regard to proposals for a single optimizing compiler back end to be used in connection with several different compiler front ends. It is to be hoped that further investigation will produce a generally acceptable solution to this problem.

Appendix III

COMPARATIVE FIGURES FOR VARIOUS COMPILERS

(quoted from Fortran, a Comparative Study by F. Bryant, Atlas Computer Laboratory, England)

Features of the compiled code

The following questions are listed A, B, C,... and an asterisk in the answer matrix means yes.

A. Is there any optimization of subscripts over a statement?

B. Is there optimization of common sub-expressions over a statement?

C. Is there optimization over many statements?

D. Is code removed from DO loops where possible?

E. Are immediate IF successors recognized?

F. Are DO loop indices optimized over the loop in any way?

G. How are array accesses treated?

(a) by code;
(b) vectors by code, two dimensions and above by subroutine;
(c) by code using dope vectors;
(d) vectors and two dimensions by code, above by subroutine.

H. Are constant subscripts recognized?

I. Is division by a constant recognized and implemented by multiplimultiplication?

J. Is exponentiation by small integers done by repeated multiplication?

K. Is multiplication and division by powers of two treated either by shifting or exponent arithmetic?

L. How good is object code listing?

e = excellent, g = good, m = moderate, p = poor .

M. Treatment of dummy arguments -- passed across by prologue:

(a) plants addresses in code;
(b) passes address of argument list;
(c) passes address in index registers.

	Atlas (London)	Atlas (Hartran)	PDP-10	1108	1900 XFAN	1900 XFAT	6600	360 H00	360 H02	4/50
A		*		*			*		*	*
B		*		*			*		*	*
C			*	*					*	*
D				(*V)					*	
E		*	*	*			*	*	*	*
F	*			*			*		*	*
G	c	a	a	a	b	d	a	a	a	a
H	*	*	*	*		*	*	*	*	*
I		*	*	*			*			*
J	*	*	*	*			*	*	*	
K			*	*?						
L	p	e	m	e	p	p	m	g	g	p
M	a	c	b	a	b	b	c	b	b	c

*? only if power is integral i.e. 2 not 2.0.

Note: Estimate that Fortran H2 optimization increases compile time (over H0) by factor of 3.

A test program for the various compilers

ODE: this is a Runge-Kutta type integration program. It consists of a master routine, the main subroutine DOA1A and a small subroutine DYBDX.

DYBDX is interesting in that the function of X in the DO loop can be removed to outside the loop. The 1108 V and 360 H02 do this, with the H02 also finding that it can keep an accumulator loaded with this function for the entirety of the loop.

The 1900 code from XFAM and XFAT is almost identical. There is some space saving in XFAT as it does some of the work in passing arguments by subroutine, whereas XFAM does it directly. XFAT cannot use its improved indexing scheme as the arrays are arguments and the scheme relies on knowing absolute addresses at load time.

The usual factor of around two in improvement by H02 over H00 is noticed. Here this is obtained mainly by optimization of the loop indices; they are kept in index registers in the correct form for accessing arrays.

The differences between London and Hartran are due to Hartran's better method of passing arguments.

The 1108 V usually passes arguments in the same way as 1108 IV. Here it does not do so, for undiscovered reasons, and its performance suffers correspondingly.

	Instructions Obeyed long	short	Ratio for large N taking 1108V=1.0	Code words	Space bits $\times 10^2$
London	247 + 156N		1.4	294	141
Hartran	271 + 179N		1.6	225	108
PDP-10	238 + 233N		2.1	262	94
1108 IV	209 + 163N		1.5	221	79
1108 V	323 + 112N		1.0	273	98
XFAM	312 + 451N +12S		4.0	458	109
XFAT	214 + 446N +34S		4.0	412	98
6600	111 + 141N +6S	98 + 140N	2.5	161	96
H00	356 + 365N	86 + 27N	3.5	507	162
H02	413 + 134N	72 + 24N	1.4	288	92
System 4	358 + 226N	96 + 11N	2.1	252	80

(S = unknown subroutines)
(N = program parameter)