

Compilers (in one lecture!)

CPE380, Fall 2025

Hank Dietz

<http://aggregate.org/hankd/>

Reference Materials

1. Introduction to Compilers and Translation Engineering course notes, by H. Dietz

<https://aggregate.org/CPE380/notes.pdf>

2. Mucky source code & CGI interface

<https://aggregate.org/CPE380/mucky.c>

<http://garage.ece.engr.uky.edu:10043/cgi-bin/mucky.cgi>

3. Online MIPS gcc in the “Compiler Explorer”

<https://godbolt.org/>

Compiling a C Program

1. **Preprocessor** (yeah, we skipped this before)
 2. **Compiler** generates **assembly code**
 3. **Assembler** creates **binary modules**
 4. **Linker** combines needed modules into one
 5. **Loader** is the part of the OS that loads a module into memory for execution
- **Usually, HLL programmers don't see this;**
1-4 invoked by **cc**, 5 when you run the program
 - We will focus on step 2, with a bit of 1 & 3

Inside a Compiler

- **Lexical analysis**: recognizing words
 - Isthiseasytoread?
 - Is this easy to read?
- **Parsing**: recognizing nestable patterns
 - Like sentence diagramming in gradeschool
- **Code Generation**: output of translated code
 - Typically output of assembly language code
 - May be embedded in parser or complex

Lexical Analysis

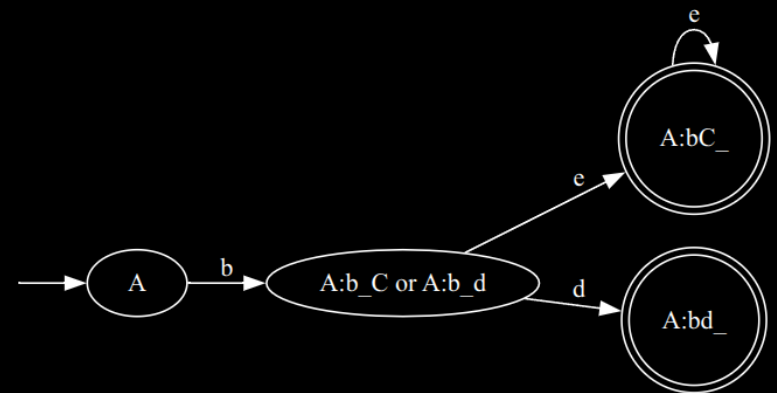
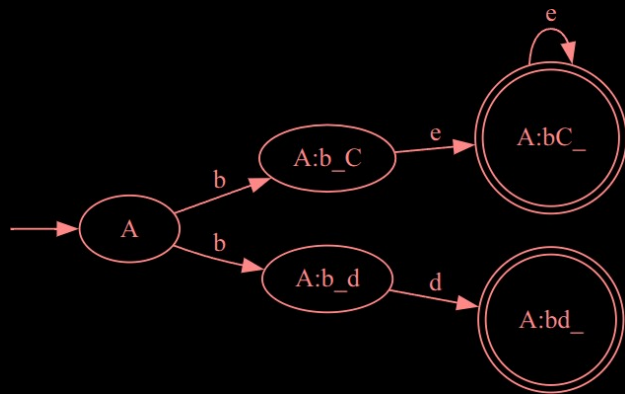
- **Regular** patterns (aka, **Type 3 Grammars**)
 - Every rule like $N : t^*N$ or $N : t^*$
 - Recognizable using **FSM** without a stack;
NFA or **DFA** (**Deterministic Finite Automaton**)

$A : bC$

$A : bd$

$C : eC$

$C : e$

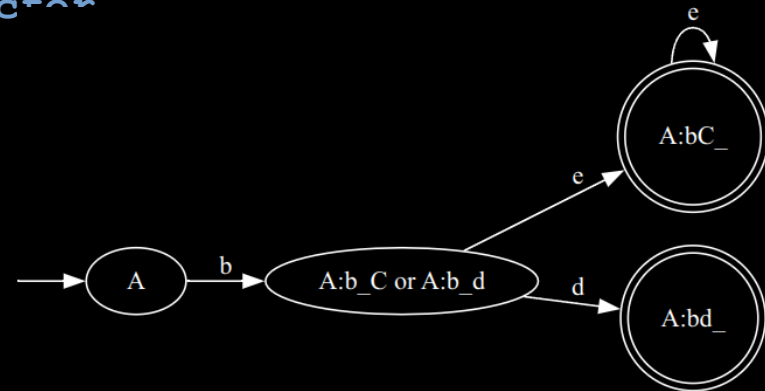


Lexical Analyzer

- Recognize next terminal
 - Always looking one character ahead
 - Lexer returns type of terminal found

```
int nextc = getchar();    // next character
int nextt;                // next token

int lexer() {
    switch(nextc) {
        case 'b': nextc = getchar();
            switch (nextc) {
                case 'd': nextc = getchar();
                    return(nextt = Abd);
                case 'e': nextc = getchar();
                    while (nextc == 'e') nextc = getchar();
                    return(nextt = AbC);
            }
        }
    error();
}
```



Lexical Analysis Tricks

- Pronouns/identifiers change type by context
 - What is `abc`? Label, variable, function...?
 - Initially lexer says `abc` is IDENT, after parser sees `int abc;` it becomes VAR by looking-up `abc` in the **symbol table**
- If keywords look like identifiers, use symbol table to disambiguate: an **atomic** lexer
- Some languages need weird lexers...
 - Fortran `DO 10 I=1,8` vs. `DO 10 I=1.8`

Parsing

- **Context-Free** patterns (aka, **Type 2 Grammars**)
 - Every rule like $N: (t|N)^*$
 - Need a stack to remember position within a rule; e.g., pairing 0 or more **b,c** around a **d**:

A: bAc

A: d

```
int A() { // recognize an A, return 1 if successful
    switch (nextt) {
        case 'b': lex(); A(); if (nextt == 'c') lex(); /* A:bAc_ */ return(1);
        case 'd': lex(); /* A:d_ */ return(1);
    }
    return(0);
}
```

Parsing Complications

Some grammars are too hard to parse by...
so computer languages avoid such structures

- **Context-Sensitive** patterns (**Type 1 Grammars**)
 - Every rule like $\alpha N \beta : \alpha (\tau | N) + \beta$ read as $(\tau | N) +$ becomes N in the context of α, β
 - In theory, pronouns/identifiers need this, but lexer + symbol table lookup suffices
- **Unrestricted** patterns (**Type 0 Grammars**)
 - Parse by nondeterministic Turing machine

Parsing Decisions

When do you decide what rule applies?

- How many tokens (terminals) of **lookahead**?
- **LL(1)** aka **recursive descent**
 - Read input $L \rightarrow R$, pick rule 1 past L edge:
i.e., $A:bAc$ and $A:d$ by looking at b or d
 - **LL(k)** is convenient for code generation...
- **LR(0)** aka **shift/reduce**
 - Read input $L \rightarrow R$, pick rule at R edge:
i.e., $A:bAc$ and $A:d$ by seeing bAc or d

Embedded Code Generation

- Simplest method embeds code generation as actions within the parser
 - `printf()` can generate assembly code
 - Place actions at the green spots:

A:bAc

A:d

```
int A() { // recognize an A, return 1 if successful
    switch (nextt) {
        case 'b': lex(); A(); if (nextt == 'c') lex(); /* A:bAc_ */ return(1);
        case 'd': lex(); /* A:d_ */ return(1);
    }
    return(0);
}
```

Grouping Expressions

- Let's generate stack code for expressions:
 - **+** has lowest priority, groups $L \rightarrow R$
 - ***** groups $R \rightarrow L$
 - **()** overrides grouping
 - variables can be **a**, **b**, or **c**

```
/* BNF for LR parsing */
add: add '+' mul printf("add\n");
add: mul
mul: par '*' mul printf("mul\n");
mul: par
par: '(' add ')'
par: 'a' printf("push a\n");
par: 'b' printf("push b\n");
par: 'c' printf("push c\n");
```

```
/* EBNF for LL parsing */
add: mul ('+' mul printf("add\n"));*
mul: par {'*' mul printf("mul\n");}
par: '(' add ')'
par: 'a' printf("push a\n");
par: 'b' printf("push b\n");
par: 'c' printf("push c\n");
```

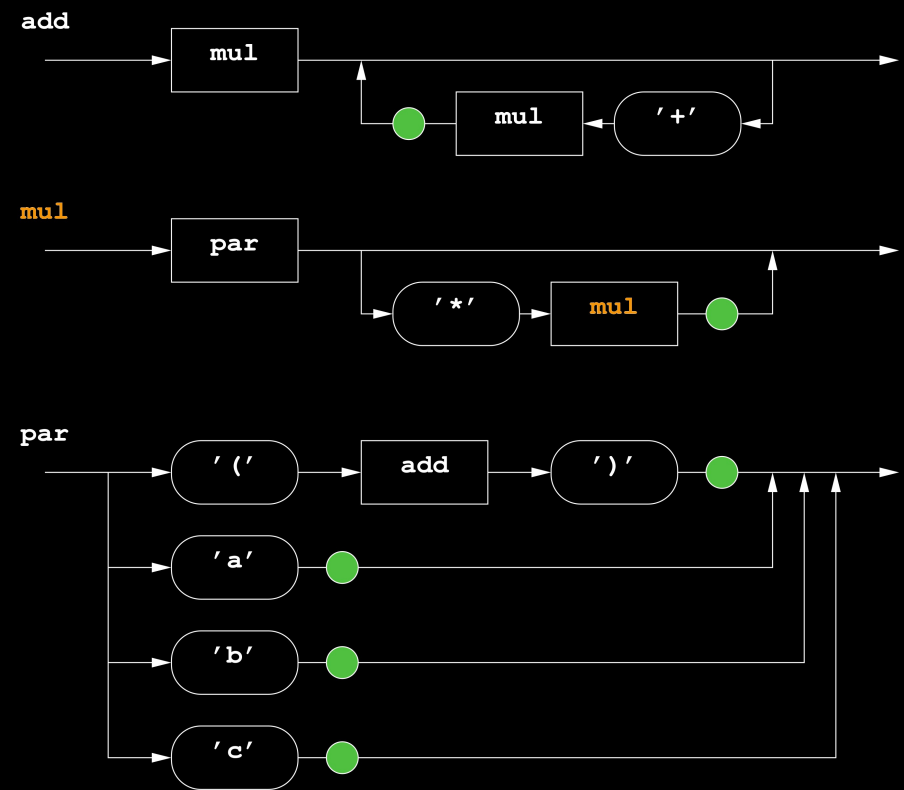
Grouping Expressions

- EBNF can be rewritten as a **syntax diagram**

```
/* EBNF for LL parsing */  
add: mul ('+' mul printf("add\n")); *
```

```
mul: par {'*' mul printf("mul\n");}
```

```
par: '(' add ')'  
par: 'a' printf("push a\n");  
par: 'b' printf("push b\n");  
par: 'c' printf("push c\n");
```

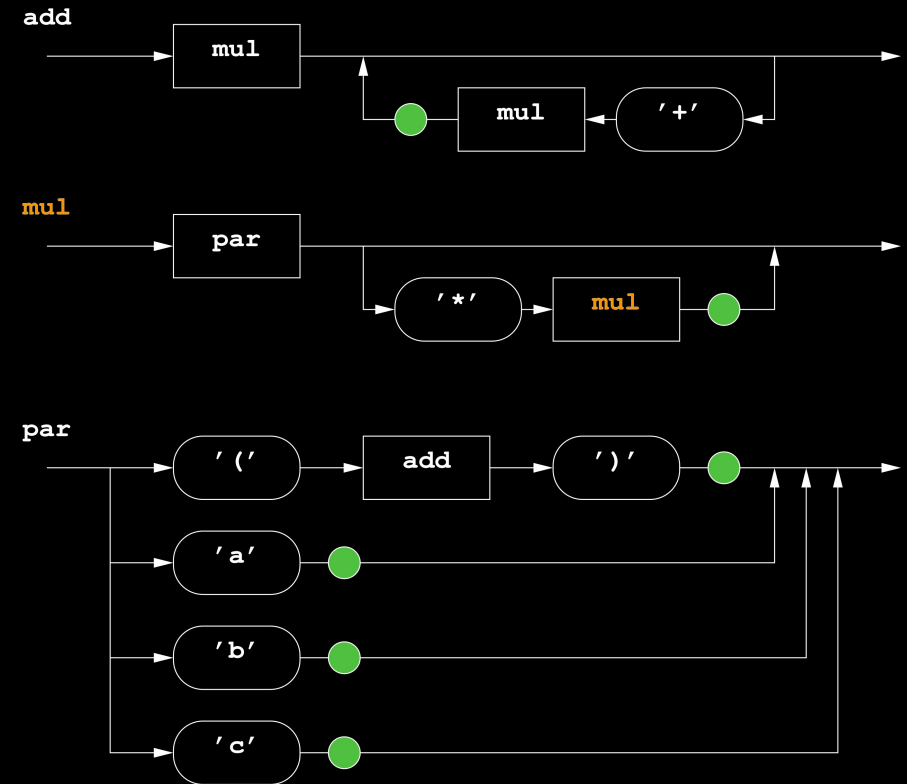


LL(1) Parser & Code Generator

```
void add() {
    mul();
    while(nextt == '+') {
        lex(); mul(); printf("add\n");
    }
}
```

```
void mul() {
    par();
    if (nextt == '*') {
        lex(); mul(); printf("mul\n");
    }
}
```

```
void par() {
    switch (nextt) {
        case '(': lex(); add(); lex(); break;
        case 'a': lex(); printf("push a\n"); break;
        case 'b': lex(); printf("push b\n"); break;
        case 'c': lex(); printf("push c\n");
    }
}
```



```

#include <stdio.h>
int nextc, nextt;
void lex(), add(), mul(), par();

void lex() {
    for (;;) switch (nextc) {
        case ' ': case '\t': case '\n':
            nextc = getchar(); break; // ignored
        case '+': case '*': case '(': case ')':
        case 'a': case 'b': case 'c':
            nextt = nextc; nextc = getchar(); return;
        default:
            nextt = '.'; return; // end of input
    }
}

void add() {
    mul();
    while(nextt == '+') {
        lex(); mul(); printf("add\n");
    }
}

void mul() {
    par();
    if (nextt == '*') {
        lex(); mul(); printf("mul\n");
    }
}

void par() {
    switch (nextt) {
        case '(': lex(); add(); lex(); break;
        case 'a': lex(); printf("push a\n"); break;
        case 'b': lex(); printf("push b\n"); break;
        case 'c': lex(); printf("push c\n");
    }
}

int main() {
    nextc = getchar(); // prime lexer
    lex();             // prime parser
    add();
    return(nextt != '.'); // ended OK?
}

```

Let's Run It!

```
$ cc comp.c -o comp
```

```
$ ./comp
```

```
a+b+c.
```

```
push a
```

```
push b
```

```
add
```

```
push c
```

```
add
```

```
$ ./comp
```

```
a+(b+c).
```

```
push a
```

```
push b
```

```
push c
```

```
add
```

```
add
```

```
$ ./comp
```

```
a*b*c.
```

```
push a
```

```
push b
```

```
push c
```

```
mul
```

```
mul
```

```
$ ./comp
```

```
(a*b)*c.
```

```
push a
```

```
push b
```

```
mul
```

```
push c
```

```
mul
```

```
$ ./comp
```

```
a+b*c.
```

```
push a
```

```
push b
```

```
push c
```

```
mul
```

```
add
```

```
$ ./comp
```

```
a*b+c.
```

```
push a
```

```
push b
```

```
mul
```

```
push c
```

```
add
```

```
$ ./comp
```

```
a*(b*a+c)+c.
```

```
push a
```

```
push b
```

```
push a
```

```
mul
```

```
push c
```

```
add
```

```
mul
```

```
push c
```

```
add
```

```
$ █
```

Better Code Generation

- Stack code is easy to generate...
- How do we generate better code?
 - Generate an intermediate representation (IR); e.g., **ASTs (abstract syntax trees)**
 - Analyze & decorate the IR
 - Apply **correctness-preserving transformations**
 - Walk the IR to generate code
- **It's complicated.** Nearly every data structure and algorithm has been used in compilers.

What About The Preprocessor?

- It is a very simple **source-to-source translator**
 - Most code passes through unchanged
 - Preprocessor directives get replaced by the translated text, usually using simple string matching and substitution
- Source-to-source translation is commonly used; e.g., C++ originally was implemented as a fancy preprocessor for C

What About The Assembler?

- Assemblers are mostly very simple compilers
- But they must handle **forward references**
 - A forward reference happens when you use something in a program before defining it
 - Remember `void lex(), add(), mul(), par();`?
 - Most compilers output code with forward references for the assembler to resolve:

```
        beq  $t0, $0, L
        ...
L:
```

Resolving Forward References

- Before seeing `L:`, you don't know the address, but can't output `0x1100????` for `beq $t0,$0,L`
- **Backpatching**: remember where you left space in output and patch each value when defined
- **Multi-pass resolution** re-parses the source
 - **Pass 1** notes locations in the symbol table
 - **Pass 2** generates code
 - May need multiple Pass 1, but can handle variable-size things (e.g., branch vs. jump)

Let's Look A "Real" Compiler

- Mucky (MIPS uCompiler from KY)
 - Accepts an undocumented C subset
 - Generated code isn't MIPS ABI compliant

<http://garage.ece.engr.uky.edu:10043/cgi-bin/mucky.cgi>

- Here's the full source code for Mucky
 - No IR; this uses embedded code generation
 - Uses CGI WWW form interface for I/O

<https://aggregate.org/CPE380/mucky.c>

- AIK assembler is at <https://aggregate.org/AIK/>

Conclusion

- You now have a decent understanding of how compilation can be implemented!
 - Some understanding of grammars & parsing
 - Simple embedded code generation
 - Saw a complete simple compiler (Mucky)
- You know they exist, but we didn't show you:
 - Generation, analysis, & decoration of IRs
 - Optimization & automatic parallelization
 - Forward reference processing in assemblers