SWAGAC'07 Parser

October 18, 2007

SWAGAC'07, SIMD Within A Gate Array 2007, is a wimpy little parallel dialect of the C language designed to handle attached processing using a processor implemented within a gate array (FPGA) chip. SWAGAC'07 features just a few control constructs, fixed-size integer vector variables and expressions, and user-defined functions. Your parser will parse SWAGAC'07 code and print debugging, error, and warning messages to the standard error stream. It will not generate output code (or anything else) on the standard output stream – that's where generated code goes.

1 Syntax

SWAGAC'07 follows C's syntax and semantics fairly closely. The following grammar, combined with the lexical rules in the previous assignment, defines the language structure. Note that the grammar is given in an Extended BNF (EBNF) notation which allows mixing of regular expression operators with BNF constructs; for example, `(decl)*` means zero or more `decl`. The nice thing about this notation is that it trivially allows the constructs of syntax diagrams to be written as text.

```
prog: (decl)* (func)* EOF
    ;
func: WORD '(' typ WORD (',' typ WORD)* ')' stat
    ;
decl: typ WORD (',' WORD)* ';'
    ;
typ: "mono"
   | "poly"
   ;
stat: '{' (decl)* (stat)* '}'
    | "if" expr stat {"else" stat}
    | "where" expr stat {"else" stat}
    | "while" expr stat
    | "return" expr ';'
    | "all" stat
    | expr ';'
```

```
    | ';'
    ;
expr: expr1 (('&' | '|' | '^') expr1)*
    ;
expr1: expr2 (('<' | '>') expr2)*
     ;
expr2: expr3 (('+' | '-') expr3)*
     ;
expr3: expr4 ('*' expr4)*
     ;
expr4: NUMBER
     | WORD {('(' expr (',' expr)* ')') | ('=' expr)}
     | '-' expr4
     | '!' expr4
     | '~' expr4
     | '(' expr ')'
     | "left" expr4
     | "right" expr4
     | "input"
     | "output" expr
     | "NPROC"
     | "IPROC"
     ;
```

The grammar is LL(1) as given. However, there is a minor ambiguity, common to most high level languages, in the grouping of `else` clauses. Given two or more nested `if` or `where` statements, an `else` clause is always treated as belonging to the innermost un-`else`'d construct. Don't worry too much about this because it is naturally how it will be recognized – you would have to do some pretty convoluted things to make it work any other way.

You may have noticed that a few liberties are taken with C's syntax in SWAGAC'07. For example, C's `if` and `while` statements require parens around the condition expression; the grammar above does not require parens, but will happily accept them as part of the expression. Similarly, function bodies do not need braces. These simplifications are harmless, essentially simplifying the grammar while imposing a very reasonable handling of what would otherwise have been easily corrected warning-level violations of the syntax.

2 Structure Of This Project

As for the lexer, you have a free choice of how to build your parser. The solution *must* be in the form of a C program which you will submit directly via a WWW form (without a paper copy). It *must* be your own work, and you are not permitted to use tools such as `lex`, `yacc`, or `PCCTS` to construct the C code. It is expected that you will reuse the lexical analyzer that you built in the previous

project; `WORD` in the above grammar refers to identifiers as discussed in the lexer phase. You also can reuse your symbol table code, although it will need to be extended to handle the relevant attributes and lexically-nested scoping.

In the next phase, your parser will be enhanced to generate code to `stdout` – for this project, nothing should go anywhere but `stderr`. What should be output?

- Every time a `WORD` that has not been declared is seen in a context that looks like a reference to a variable (as opposed to a function call), output an *error* message stating that the variable was not declared. This error should not terminate the parse. At your option, you may also handle other errors, and it is permissible for other errors to terminate the parse.
- The comma (`','`) only appears in places in this grammar where the meaning would be unambiguous even with that symbol missing. If it is omitted, output a *warning* message stating that the parser has assumed this symbol was missing. At your option, you may also handle other warnings; no warning should ever prematurely terminate the parse.
- Only if your code was compiled with the `SYMBUG` flag defined (i.e., `-DSYMBUG` on the `cc` command line), every time a `WORD` is seen, print a message giving the name (lexeme) of the `WORD` and the type that your parser/symbol table have associated with it. The type is one of three things: `mono`, `poly`, or `function`.

There is actually a standard format for error and warning messages, designed to facilitate programming environment tools automatically processing the messages... however, you need not strictly comply with that standard for this project. The standard message format is something like:

filename`:`*linenumber*`:` *type*`:` *message*

Your project is reading from `stdin`, so it doesn't know the file name; a dash (`-`) would be used to indicate this. The line number reported by your project is allowed to be somewhat inaccurate; you may report the line number the lexer was at when the parser noticed the problem, rather than the prefered line number at which the faulty construct began. An easy way to approximately track line numbers is to have a global variable which starts at 1 and is incremented every time a newline character (`'\n'`) is read from `stdin`. The type is either `error` or `warning`. The message is a simple explanation of what the parser thinks was wrong.

3 Due Date & Submission Requirements

This project is due before class on **October 25, 2007**. It should be submitted via the the appropriate WWW form on the course site as a "tarball" created using `tar -c`. It must contain at least three files:

1. Your C source code, in one or more files.

2. A `Makefile` which will build an executable file called `parser` by simply typing `make`.

3. Minimal "implementor's notes" describing how your code works... or fails to work. The document should be formatted as very straightforward HTML and placed in a file named `notes.html`. The HTML should be self-contained, not referencing anything outside of the contents of the tarball. Code which is broken but documented as such will get only half as many points taken off for such errors. This is true even to the extent of an empty assignment being given half credit if the documentation says something like "This doesn't work because I didn't get around to writing the project code."