SWAGAC'07 Lexical Analysis

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.

The construction is broken down into three stages: lexical analysis, parsing, and code generation. Your lexical analyzer will simply recognize a token at a time, with a degenerate parser that simply keeps calling the lexer until an EOF token is returned. You'll need a symbol table, but all it will do for now is record how many times each identifier is referenced. So that we can see things are working, this "parser" also will print information about each token to `stderr`.

1 SWAGAC'07 Lexemes

SWAGAC'07 is based on C. Consequently, it follows C's syntax and semantics fairly closely. This means that the rules for handling of white space, characters in an identifier, etc. are the same except as noted here.

An identifier begins with an alphabetic character, and is optionally followed by any number of alphabetic or numeric characters; the underscore character (`_`) is considered to be alphabetic. Case is significant; `a` is not the same as `A`. It is allowable to ignore characters after the first 7 in an identifier.

The SWAGAC keywords all would be identifiers if they were not keywords. They are:

```
mono
poly
if
else
where
while
return
all
left
right
```

```
input
output
NPROC
IPROC
```

Note that all the keywords are lowercase except the last two.

Your lexer also must recognize decimal numbers, which look like a sequence of one or more digits. In C, a leading 0 digit is used to indicate an octal value, so, technically, `0` is the octal value zero. However, SWAGAC doesn't have octal constants, so leading 0s do not change the base to octal.

There are not too many special symbols treated as tokens in SWAGAC, and none of them is more than one character long. The special symbols you must recognize are:

```
( ) , ; { } & | ^ < > + - * = ! ~
```

Note that end-of-file, EOF, also behaves somewhat like a special symbol.

2 Structure Of This Project

Different kinds of lexical analyzers have been discussed; this project will involve writing a simple lexical analyzer and minimal symbol table interface. 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. Your symbol table may use linear search or whatever method you wish, but again, it must be your own code.

Your program is to be written in a style that facilitates reuse of the lexer in a parser. Toward this goal, the `main()` *must* call a function `lex()` to return each token's type until an EOF token is returned. The `main()` should print, on a separate line for each lexical item, the type and lexeme of the token just read. The formatting should be:

- For any identifier, output the lexeme followed by the number of previous times this lexeme has been seen; this number should be recorded in the symbol table entry created by the lexer
- For any of the keywords listed, simply output the keyword's lexeme
- For any decimal number, output the value of that number using `fprintf(stderr, "%d", ...)`
- For any special symbol, simply output the symbol; for the end-of-file, output `EOF`

The lexer should not return a token for any other characters; `lex()` should essentially ignore the other symbols except in the sense of treating them as "edges" between tokens – like whitespace. It is generally easiest to do this by having the lexer advance past other characters before recognizing the current token.

For example, input on `stdin` of:

```
poly abc, def;
func(mono p)
{
    if (abc>23) def=p*5+left(abc);
}
```

Should cause output of the following to `stderr`:

```
poly
abc 0
,
def 0
;
func 0
(
mono
p 0
)
{
if
(
abc 1
>
23
)
def 1
=
p 1
*
5
+
left
(
abc 2
)
;
}
EOF
```

3 Due Date & Submission Requirements

This project is due by 11:59PM on **October 19, 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 `lexer` 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."