Computer Engineering 175
Phase I: Lexical Analysis

Polonius: “What do you read, my lord?”
Hamlet: “Words, words, words.”
Shakespeare, *Hamlet*, Act II

1 Overview

In this assignment, you will use `flex` to generate a scanner for the Simple C language. This assignment is worth 10% of your project grade. Your program is due at 11:59 pm, Sunday, January 14th.

2 Lexical Structure

The following points summarize the lexical rules for Simple C:

- spaces, tabs, newlines, and other whitespace are ignored
- comments are surrounded by `/*` and `*/` and may not include a `*/`
- an identifier consists of a letter or underscore followed by optional letters, underscores, and digits; a letter is one of `a`, `b`, ..., `z`, `A`, `B`, ..., `Z`; a digit is one of `0`, `1`, ..., `9`
- keywords are `auto`, `break`, `case`, `char`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extern`, `float`, `for`, `goto`, `if`, `int`, `long`, `register`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `union`, `unsigned`, `void`, `volatile`, and `while`; keywords may not be used as identifiers
- an integer consists of one or more decimal digits
- a real number consists of one or more digits, a decimal point, one or more digits, and an optional exponent, which itself consists of the letter `e` in either lowercase or uppercase followed by an optional sign and then one or more digits
- a string consists of zero or more characters enclosed in double quotes (`"` ... `"`) and may not contain a newline or a double quote unless the quote is immediately preceded by a backslash
- a character consists of one or more characters enclosed in single quotes (`'` ... `'`) and may not contain a newline or a single quote unless the quote is immediately preceded by a backslash
- operators are =, `||`, &&, `|`, ==, `!=`, <, >, <=, >=, +, −, ∗, /, %, &, `!`, ++, `--`, `->`, `(`, `)`, `[`, `]`, {, }, `;`, `:`, `.`, `,`, and `...`
- any other character is illegal

3 Assignment

You will use `flex` to generate a lexical analyzer for Simple C. The analyzer is specified as a series of rules written as pattern-action pairs. Each pattern is specified as a regular expression. An action consists of arbitrary C++ code. If a pattern is matched, the action is executed. For example, the following rule would match and ignore common whitespace characters:

```
[ \f\n\r\t\v]+                { /* do nothing */ }
```

The pattern with the longest possible match is used. If multiple patterns match strings of the same length, then the first such pattern is used. Therefore, the rule for identifiers should come after the rules for keywords:

```
...
while                          { /* keyword matched */ }
[a-zA-Z_][a-zA-Z0-9_]*         { /* identifier matched */ }
```

The rule file is processed by `flex` to produce a C/C++ file that when compiled reads from standard input by default, so no change is necessary. Upon recognizing a lexical construct, your scanner will indicate what it has recognized to the ***standard output*** (i.e., `std::cout`) as follows:

- ignored and illegal constructs produce no output
- keywords are indicated as `keyword` *keyword*
- identifiers are indicated as `identifier` *identifier*
- integers are indicated as `integer` *literal*
- reals are indicated as `real` *literal*
- strings are indicated as `string` *literal*
- characters are indicated as `character` *literal*
- operators and other valid symbols are indicated as `operator` *text*

For example, the input string `int x;` consists of three tokens: a keyword, an identifier, and an operator. Therefore the output would consist of three lines: `keyword int`, `identifier x`, and `operator ;`. Your program will only be given ***lexically correct*** programs as input. However, it is strongly advised that you should test your program against lexically incorrect programs as a way of finding errors in your implementation.

4 Hints

Place each keyword on its own line as `flex` can be fussy about breaking a long regular expression across lines. For some tokens, you will need to escape characters such as quotes that otherwise have special meaning:

```
\"(...)*\"                     { /* string matched */ }
```

A string or character cannot contain a newline character. They can however contain `\n` (two characters consisting of a backslash followed by the lowercase letter `n`), which is an escape sequence used to represent a newline character, just as `\t` is an escape sequence used to represent a tab character.

You will find it very difficult to write a regular expression to recognize a comment. Instead, it is easier to implement a scanner by hand. To do this, you will need to have an action that is executed upon seeing the beginning of a comment. You will then consume input characters checking for the end of the comment. To do this, you can use `yyinput()` to read a character and (optionally) `yyunput(int c)` to put back a character onto the input stream. (If you do use `yyunput`, make sure you are not using the `nounput` option in any `%option` declaration, otherwise you will get an error.)