ASSIGNMENT 1: Lexical Analysis

OVERVIEW

The first major component of your compiler is the lexical analyzer. It is recommended that you build this using the lex/flex tool, which takes a specification file in `.l` format and generates a `.c` file. This file defines a number of internal functions and data structures, as well as one very important public function called `yylex()`. In brief, calling `yylex()` causes the input source code file to be read one character at a time until a token is recognized, then `yylex` returns the token code. Since the ultimate goal is to integrate this into your compiler, it is important to follow the recommended coding practices so your lexer will play nicely with the other components of the compiler. While you may also code the lexer by hand, **this is not a productive use of your valuable time**.

You will also need to write a small testing program to try out your lexer. This program will be thrown away after the first assignment. Finally, you will apply your lexer and test program to a set of test cases, and compare the output to "correct" results.

INPUT ASSUMPTIONS

In the C language, pre-processing takes place before lexical analysis, although many compilers (including gcc) perform both within the same lexer. You may assume that the input to your lexer has already been passed through the preprocessor, which can be invoked with the command gcc -E. Pre-processing handles the following for you:

include files
conditional inclusion of source (e.g. #ifdef)
trigraph conversion and other stupidity (but not C-99 "digraphs")
joining of lines continued with a \ at the end of the line
macros

According to section 5.1.1.2 of the ISO C Standard, this corresponds to translation phases 1 through 4. **You don't need to worry about pre-processing, other than picking up the line numbers:**

LINE NUMBERS AND ERROR REPORTING:

You will see that the output of gcc -E contains lines which begin with a #. These are line number markers, and tell you where in the original source file the next line came from. These markers are output by gcc -E whenever it takes any action which adds or deletes lines. By extracting the information from these marker lines, and keeping track of newlines, you can maintain the current filename and line number for error reporting purposes.

Error reporting and in particular error recovery is not easy, but we're going to get started with this first assignment. If your lexer encounters input which does not conform to the lexical rules, it should output a descriptive error message with the file name and line number of the place of error. It may then halt. As a future refinement, we'll discuss ways to attempt recovery and further processing. Remember: error messages go to stderr, not stdout! Example:

```
$ ./mylexer test1.c

test1.c:10: Error: unrecognized character '@'
```

TOKEN CODES

In the second assignment, we'll introduce the parser generator `bison` which is an improved version of the original `yacc`. yacc/bison expects `yylex()` to return integer token codes. By convention, single-character tokens can be represented simply by their ASCII value. E.g. if you recognize an = token, you may `return('=')`. This obviously won't work for multi-character tokens. You will need to provide a .h file containing a list of #defines for each token code that is not represented by a simple character. yacc/bison likes these numbers to start at 257 to avoid conflict with ASCII codes, e.g.

```
#define EQEQ 257
#define NOTEQ 258
```

Note that yacc/bison prefers to see 0 as the EOF token, not -1, but it tolerates -1.

yacc/bison can generate this .h file for you, or you can create it manually and have it included with your parser. For this assignment, we are not using a parser, and so you will have to manually create the file (or, for testing purposes, use the tokens-manual.h which is provided).

WHAT ARE THE TOKENS?/ WHAT CAN I SKIP?

Read the C standard or Harbison & Steele. Basically, tokens are keywords, identifiers, numeric constants, string literals, or operators/punctuators.

You don't need to implement Universal Character Names, nor wide characters / multibyte characters. String literals needn't have an optional encoding prefix.

You must implement all of the numeric constants, including properly recognizing them and converting them to an internal representation (int, float, etc.). See below about token values. You should recognize floating-point constants with a "p" form but aren't required to understand their value (although that isn't at all difficult). You should recognize an L in front of a character or string, but aren't required to understand or deal with wide characters. *If you find that you are pressed for time, it is better to turn in something with broken floating-point (real numbers) handling than to fall behind. We will not be dealing with floating point arithmetic in the final compiler.*

Recognize **all of the keywords** listed in the standard. If you want to skip "digraphs", which are a C99 extension for progammers unfortunate enough to have broken keyboards, you may, although they aren't difficult either.

Note that **whitespace is not a token**! Whitespace occurs between tokens, and is silently consumed. You will need a lex pattern to recognize and discard whitespace, and also to count newlines.

TOKEN SEMANTIC VALUES

yacc/bison expects the lexer to return token semantic values in a global variable called `yylval`. It doesn't look at this variable to make parse decisions, but as we'll see, being able to maintain semantic values for terminals and non-terminals in the grammar is the underpinning for the entire syntax-directed translation process.

yylval is by default an int but that isn't very useful for anything but a trivial calculator. Let's say that lexical values could be either strings or integers. yacc/bison is expecting a union definition like this:

```
typedef union {
          char *string_literal;
          int integer;
} YYSTYPE;

extern YYSTYPE yylval;
```

The above is representative of the form of the declaration although you will probably need something a bit more complicated. It is up to you to provide this declaration in your header file, which will ultimately be included by your parser. For now, you can return identifiers simply as a char *. In the future, you'll see how it is helpful to have a symbol table as part of your lexer and identifiers will be returned as pointers to a struct of your design. For numeric constants, you need to somehow indicate what *type* of constant you have seen. This is discussed below.

Note: It is a good idea to pass all information about the lexical semantic value of a token in the single global `yylval` union. While it is tempting to have other global variables for this purpose, and especially tempting in this first assignment, this approach will cause major problems later on!

Memory Management

We'll have a brief unit discussing memory allocation and management strategies within the compiler. Note that the variable `yytext` will be a pointer to a string which is the *lexeme* when a pattern in lex matches. It will be tempting to simply do something like `yylval=yytext` but if you do that, please remember that yytext is pointing to an area of memory that will be overwritten at the next invocation of yylex! Therefore, at some point you need to make a copy (e.g. with strdup) of this information. Failing to do this is a common assignment 1 problem: it will appear to work correctly in testing, since you are going one token at a time, but then will create crazy problems later on, such as variables changing names!

TESTING

To test your lexer, write a simple main program which repeatedly calls yylex() and prints out **one line for each token**, delimited by **tabs**, and containing the following columns:

filename in which the token was found
line number at which the token was found
token name
token semantic value
other token information (if necessary)

As we'll discuss in class, by standardizing on the output format, we will be able to compare and contrast the performance of each student's lexer, and/or to compare the output of your lexer against a presumed good output.

filename: The name of the file at which the token was found. This information can be extracted from the # directives if the input is coming from the preprocessor. If you have not yet seen any such directives, use <stdin>

line number: an integer formatted as %d

token name: for single-character tokens, use the actual character, e.g. [. For multi-character or variable-length tokens, you will have a defined name for each. To aid testing, please use the names found in the header file supplied on the course web site. They should be self-explanatory.

token semantic value: don't output this column (including the \t) if there is no value associated with the token, such as punctuation marks. For all other token types, output according to the following:

```
Token Type                            Output
------------------------------------------------------------------------------
         IDENT                        The identifier name
         NUMBER                       value, %lld for ints, %Lg for reals
         CHARLIT                      The actual character, if it is
                                      printable, otherwise the escape code:
                                               \0,\a,\b,\f,\n,\r,\t,\v
                                         else:  \\%03o
                                      also, always escape ’ " and \
         STRING                       Each character of the string, except
                                      for the implicitly added nul terminator,
                                      all escaped if necessary as with CHARLIT.
                                      Ponder: "\0\0\0" is not the same as ""
```

For CHARLIT and STRING token types, the token value is the single byte or the array of bytes, respectively. It is **not** the original lexeme. There are many different ways to represent the same character or string literal. They should be reduced to an identical internal representation. Write the code to convert the character literal to a char, or the string to an array of chars terminated by a \0. This includes all of the escape code processing. Then write the code to display a single character, or a string of characters, including the escape conventions for output described above. Do **not** simply print out `yytext` to show the semantic value of a STRING or CHARLIT.

Handling all cases of character literals and string literals can be somewhat difficult. Suggestion: read up on "start conditions" in lex/flex. Note that an elegant solution will use the same logic and code to handle character escapes, regardless of whether they are seen inside a character literal or a string literal.

For some token types which are numbers, output a 5th column with additional information about the type of the number as inferred from its lexical form. E.g. a decimal constant followed by the letters `ULL` tells us that it is an unsigned long long. Although we are a long way from discussing the C type system, it is important that we capture these lexical type cues now.
For real NUMBERs, output the size specifier:FLOAT, DOUBLE, LONGDOUBLE.
For integer NUMBERs, output UNSIGNED, if the constant is tagged as unsigned, then output the size specification: INT, LONG, LONGLONG.

As an example, here is part of the output from the test cases:

```
ltests/chars.c     1        STRING    \0\0\0
ltests/chars.c     5        STRING    \0018\377\001\256Z
ltests/kw.c        1        AUTO
ltests/kw.c        1        ;
ltests/num.c       5        NUMBER    REAL      0.1       DOUBLE
ltests/num.c       8        NUMBER    INTEGER   63        UNSIGNED,LONGLONG
```

TEST CASES

You’ll find a few .c files in the subdirectory `ltests` on the course web site. You can run all of them through your testing program, e.g.

```
         gcc -E ltests/*.c |./lexertester
```

The files ltests.out and ltests.err contain example output and errors from doing so. Note that the test cases contain a few tricky conditions and warnings/errors, as discussed in class.

The goal of assignment #1 is not simply to get your output to match against the reference output! Step-by-step functional testing is important throughout the development of any complex project. Your lexer tester is simulating what the parser will later be demanding from the lexer. We want to make sure the lexer functions correctly under all circumstances. There are many shortcuts that might tempt you (some of which are mentioned herein) along the way. Steer away from these. Assignment #1 may seem daunting at first, but it in fact represents about 5% of the total intellectual and technical complexity of the entire project. Saving a little time now and bypassing something that you don't quite understand or can't quite get to work right might cost you much more time in later stages of the project.

TOKEN NAMES

The token names that were used in the test case output are defined in the file `tokens-manual.h` which is on the course website. They are an enum although you could also use #define. Note that single-character tokens such as + are represented by their ASCII value and are not in this list.