CS 3276 Project #2
Building a Scanner

The purpose of this lab is to build a lexical analyzer for the Tiger language. This analyzer is outlined on pages 33-34 of the document titled "*Tiger Ch2 lexing.pdf*" which is available from the assignment page on Brightspace. The Tiger language reference manual also contains lexical analysis information. The analyzer is to be built using **flex**, a tool for generating lexical analyzers.

A set of files is available to help you get started. The files are available on Brightspace in a compressed archive file `project2.tar.gz`. Download this file to your unix system and extract the files from the archive using tar (use the command: `tar -xvf project2.tar.gz --gunzip`). The files in the archive include a makefile, an initial lex file `tig_scan.l`, and some auxiliary files. The files are C++ source files for the GNU g++ compiler. You should familiarize yourself with their contents before attempting changes. Included are the following:

- `errormsg.cpp` & `errormsg.h` -- which aids in giving error messages with the correct line and character position
- `util.cpp` & `util.h` -- a small set of utility functions
- `tokens.h` -- defines a unique numeric token-id for each kind of token
- `driver.cpp` -- for testing the lexer
- `Makefile` -- for compiling and linking all files
- `tig_scan.l` -- the file you will need to modify
- `lextest-good` -- a fully working executable that you can compare your scanner against

In its present state `tig_scan.l` recognizes only a few kinds of tokens: " " (a blank which is ignored), '\n' (a newline which we count so that our error messages produce correct line numbers), "," (token COMMA), the keyword "for" (token FOR), and integer literals (token INT). There is also a function **`adjust()`** to help keep track of our position in the input file.

Basic assignment

Your task is to extend `tig_scan.l` so that it:

- Recognizes all of the tokens in the Tiger language, and returns an appropriate token value. The tokens that need to be recognized are listed below. The only tokens that return additional values should be integer literals (token INT), identifiers (token ID), and string literals (token STRING)
- Recognizes (and ignores) Tiger comments: sequences beginning with a '/*' and ending with a '*/'. For the basic assignment, you do not have to handle nested comments.
- Recognizes (and ignores) whitespace characters.
- Calls **`EM_newline()`** whenever a newline character is found so that line/character positions are reported properly.
- Recognizes and returns string literals. For the basic assignment, you do not have to handle strings with embedded escape sequences or multi-line strings.

The tokens that need to be recognized are as follows:

- Tiger's special-character tokens: `> < = <> >= <= + - * / & | ; : := , . ( ) [ ] { }`

Each should return the appropriate token-code (e.g. '[' should return `LBRACK`). A couple of these are already done for you.

- Tiger's reserved words: `array break do else end for function if in let nil of then to type var while`

  Each should return a token of the appropriate token-code. (Because Tiger is a case-sensitive language, this should be a case-sensitive match.)

- Integer literals (this one is already done for you).
- Identifiers. An identifier is a sequence of letters, digits and underscores in which the first character is a letter. The token-id for an identifier token should be ID; the associated yylval value should be a pointer to a string containing the identifier's name.
- String literals. For the basic assignment, you only need to recognize "simple" string literals, those that do not contain any escape sequences. (They may contain any other printable character, however.) The token-id for a string literal should be STRING; the associated yylval value should be a pointer to a string in which the begin and ending quote-characters have been stripped off.

Extensions

In addition to the basic assignment, you may wish to do some or all of the following.

<u>Extension 1</u>. Implement nested comments, so that a comment continues until the number of '*/' sequences equals the number of '/*' sequences. You will likely need to add additional start states and an instance variable to keep a count. Be sure to keep track of all newline characters encountered so that error messages report correct line numbers.

<u>Extension 2</u>. Recognize Tiger's string escape sequences. Here, your scanner should recognize each of the following as representing a single character:

- `\\  \"  \n  \t`
- `\ddd`, where each of the d's represents a decimal digit that is interpreted as decimal value of an ASCII character
- `\^` followed by a letter (to get the correct control value, bitwise AND the letter with the hex value '1F')

You will need to write code that will replace all multi-character escape sequences with the corresponding single character.

<u>Extension 3</u>. Recognize Tiger's "continuation string literals" that span more than one line. These are the string literals in which backslash-whitespace-backslash sequences can be embedded that are treated as a "comment" in that they are not part of the final string. Their primary purpose is to allow long strings to extend across multiple lines, as in:

```
 var abc:string := "I can never get enough programming.  It's what I want to \
                   \do every moment of the day.  When I'm not programming, \
                   \I feel dull and listless."
```

The backslash-whitespace-backslash sequences need to be removed from the string and all newline characters counted so that our error reporting is done correctly.

Memory Management

You cannot depend upon the stability of the internal buffers that lex/flex uses, since it will likely overwrite their contents when it reads the next chunk of the input file. Thus a copy of any string pointed to by `yytext` needs to be made prior to returning that string to the parser (`driver.c` in this case). The responsibility of freeing such strings, if required, will be given to the parser.

Error Handling

Your scanner should attempt to catch the following error conditions:

- Unrecognizable character (this one is already done for you)
- Unterminated comment
- Unterminated string
- Invalid escape sequence in a string (if you attempt extension 2 or 3)
- Unmatched closing comment (a "*/" not preceded by a matching "/*")

You need not attempt error correction, but must at a minimum report a useful error message.

Building and Testing

In order to successfully build your scanner, you'll need to set things up properly. First you will need to make sure that your Linux system has the necessary flex & g++ software. That can be accomplished by entering the following commands in a terminal window:

```
sudo apt-get update
sudo apt-get install flex
sudo apt-get install g++
```

Next download the supplied archive and extract all the files:

```
gunzip project2.tar.gz
tar –xvf project2.tar
```

Once you have extracted the files from the archive, you should be able to go into the directory that contains the files (project2/) and type "`make`". It should run flex and then run the g++ compiler to build everything. Type "`make`" again whenever you have edited a file and it will compile any changed files and generate a new executable. If at any time you want to recompile everything, type "`make clean`" and then "`make`".

At this point, you should be able to run the scanner on a test file. To run it on a file `test.tig`, you would give the command

```
lextest test.tig
```

If your file contains nothing but blanks, newlines, ",", "for", and integer-literal tokens, it should report relevant information about each, one per line. If it contains anything else, it should give an error message for each character that it doesn't know how to handle.

As you improve `tig_scan.l`, the scanner should recognize more and more kinds of tokens. I recommend that you add a few tokens and test, then add support for more tokens and test, etc. Professor Appel has supplied a number of test cases, which are also available on Brightspace (under Content -- Course Resources) in a compressed archive file. Place the "`testcase`" directory at the same level as your "`project2`" directory;

then you can use the supplied `test-all` script to run your `lextest` executable on all the test cases (requires csh is installed). Note: you may wish to create your own test cases too, as I have many additional test cases that I will use when grading your scanner. There is also a `compare-all` script that will compare the output of your `lextest` executable against the output of the `lextest-good` executable. The testing scripts expect that you have `csh` installed on your Linux system.

Example

The completed assignment should be able to recognize all of Tiger's tokens, printing the relevant information one token per line. If the source file contains:

```
/* an array type and an array variable */
let
 type  arrtype = array of int
 var arr1:arrtype := arrtype [10] of 0
in
 arr1
end
```

The output should look like this:

```
     LET   43
    TYPE   48
      ID   54 arrtype
      EQ   62
   ARRAY   64
      OF   70
      ID   73 int
     VAR   78
      ID   82 arr1
   COLON   86
      ID   87 arrtype
  ASSIGN   95
      ID   98 arrtype
  LBRACK  106
     INT  107 10
  RBRACK  109
      OF  111
     INT  114 0
      IN  116
      ID  120 arr1
     END  125
```

Each token is listed followed by its position in the input file (character offset from the start of the file). Values associated with certain tokens are also printed when applicable (for identifiers, integers, and strings).

Project Submission

Submit your projects on Brightspace via the assignment page. You should submit the file `tig_scan.l` since that is the only file you should have modified. We will make an executable from scratch using your `tig_scan.l` file to make sure everything compiles cleanly. Also submit a `README` file that describes your project. The `README` file should state which parts and extensions of the project you completed, as well as describe any particularly interesting code or structures. Please also include an honor statement in the `README` file.

Grading

All students will start with a score of 85, a middle-B grade (on a scale of 100). If you correctly implement the basic scanner, 85 will be your grade. Your score can move up if you correctly implement any of the extensions (5 points per extension). Your score can move down if your implementation is incorrect or has errors. Your score can also move down for poor programming style or the absence of a useful `README` file.