Programming Project 1: Lexical Analysis

The Goal
In the first programming project, you will get your compiler off to a great start by implementing the lexical analyzer. For the first task of the front-end, you will use `flex` to create a scanner for the GLSL programming language. Your scanner will transform the source file from a stream of bits and bytes into a series of meaningful tokens containing information that will be used by the later stages of the compiler.

This is a fairly straightforward assignment and most students don't find it too time consuming. However, we are giving you more than a week to work on it. Don't let that lull you into procrastinating. Most of the work comes in figuring out how **flex** works. If you've never really played with **flex**, then start sooner than later so you don't find yourself using a late day on the easiest assignment of the quarter. Once you get up to speed, things should go relatively smoothly.

GLSL Lexical Structure
The handout on the GLSL specification contains a full description of the lexical structure of GLSL spec under Chapter 3. Your job in this assignment will be to write a scanner that implements this lexical specification.

When writing your scanner, you will need to handle conversions from integer and real-valued literals into integer and real-valued numeric data. That is, if you encounter the sequence of characters **3.1415E+3**, while scanning the input you should convert this into a **float** with this value. Similarly, when seeing **137** you should convert it to an **int**. To save you some time and energy, you may want to use the **strtol** or **strtod** functions to do these conversions. You can learn more about **strtol** and **strtod** by typing **man strtol** or **man strtod**. Alternatively, if you're familiar with the C++ **stringstream** type, feel free to use that as well. You may assume that any numeric literal, whether integer or real, can be converted to the appropriate type without errors, so don't worry about literals that overflow or underflow.

GLSL adopts the two types of comments available in C++. A single-line comment starts with **//** and extends to the end of the line. Multi-line comments start with **/*** and end with the first subsequent ***/**. Any symbol is allowed in a comment except the sequence ***/** which ends the current comment. Multi-line comments do not nest. Your scanner should consume any comments from the input stream and ignore them. If a file ends with an unterminated comment, the scanner should report an error.

Recall that the following operators and punctuation characters are used by GLSL:

```
+ - * / < <= > >= = == != && || ; . [ ] ( ) { }
```

Note that two-character operators must not have any space in between the two characters. Each single-character operator has a token type, while the multicharacter

operators have named token types associated with them. For example, the token type "**||**" has token type **T_Or**.

Using flex
Inspect the CS131 web site/piazza. You'll find links to various **flex** resources.
We may not cover all the details of the tools in lecture, so you'll forage in the online docs to learn what you need to know. A quick way to get help is using GNU **info**. (Try "**info flex**")

flex is not exactly user friendly. If you put a space or newline in the wrong place, it will often print "syntax error" with no line number or hint of what the true problem is. It may take some delving into the manual, a little experimentation, and some patience to learn its shortcomings. Here are a few suggestions to help keep you sane:

- Be careful about spaces within patterns (it's easy to accidentally allow a space to be interpreted as part of the pattern or signal the end of pattern prematurely if you aren't careful).
- Never put newlines between a pattern and an action.
- When in doubt, parenthesize within the pattern to ensure you are getting the precedence you intend.
- Enclose each action in curly braces (although not required for a single-line action, you're better safe than sorry).
- Use the definitions section to define pattern substitutions (names like **Digit**, **Exponent**, etc.). It makes for much more readable rules that are easier to modify, extend, and debug.
- You must put curly braces around the definition name when you are using it in another definition or a pattern; without them it will only match the literal name.

Starter files
The starting files can be found in piazza. The starting **p1** project contains the following files (the boldface entries are the ones you will need to modify):

`Makefile`	builds scanner
`main.cc`	**main** for scanner
`scanner.h`	type definitions and prototype declarations for scanner
scanner.l	starting scanner skeleton
`errors.h/.cc`	error messages you are to use
`utility.h/.cc`	interface/implementation of various utility functions
`samples/`	directory of test input files

Copy the entire directory to your home directory. Your first order of business is to read through all the files to learn the lay of the land as well as to absorb the helpful hints contained in the files.

You should **not** modify **errors.h/.cc** or **main.cc** since our grading scripts depend on your output matching our defined constants and behavior. You may (but don't need to) modify **utility.h/.cc**.

You will definitely need to modify **scanner.l** to recognize tokens. All required GLSL
token types for this project are already in *enum TokenTypes* declaration in **scanner.h**.
Token types and corresponding names follow the "standard" naming convention here:

- Token types. For example,

```
typedef enum { ...,
                T_VEC2,
                T_VEC3,
                T_VEC4,
                T_MAT2,
                T_MAT3,
                T_MAT4,
                T_IVEC2,
                T_IN,
                T_OUT,
                T_UNIFORM,
                ...,
                T_NumTokenTypes } TokenTypes;
```

- The corresponding token names. For example,

```
static const char *gTokenNames[] = {
            ...,
            "T_Vec2",
            "T_Vec3",
            "T_Vec4",
            "T_Mat2",
            "T_Mat3",
            "T_Mat4",
            "T_Ivec2",
            "T_In",
            "T_Out",
            "T_Uniform",
            ...};
```

- Make sure the token types (enum) are one-to-one mapping to the token
  names (*gTokenNames*). That is, they are listed in declarations in the ***exactly
  same*** order.

You may use our sample **Makefile** as a start to build the project. The **Makefile** has a
target to build the **glc** executable. **glc** must read input from **stdin**; therefore, you can
use standard UNIX file redirection to read from a file. For example, to invoke your
compiler (well, your scanner) on a particular input file, you would use:

```
% ./glc < samples/foo.glsl
```

If a file ends with an unclosed multi-line comment, report an error via a call to one of the
methods in the **ReportError** class. In order to match our output exactly (which is
important for our testing code), please use the standard error messages provided in
errors.h .

Scanner implementation

The **scanner.l** lex input file in the starter project is where you'll do your work. The **yylval** global variable is used to record the value for each lexeme scanned and the **yylloc** global records the lexeme position (line number and column). The action for each pattern will update the global variables and return the appropriate token code. Your goal is to modify **scanner.l** to

- skip over white space; but need to keep tracking the column numbers after skipping. For example, single space increases the column by 1, and tab increases the column by 8;
- recognize all keywords and return the correct token from scanner.h;
- recognize punctuation and single-char operators and return proper tokens;
- recognize two-character operators and return the correct tokens;
- recognize int, float, and bool constants, return the correct token and set appropriate field of yylval;
- recognize identifiers, return the correct token and set appropriate fields of yylval;
- record the line number and first and last column in yylloc for all tokens;
- and report lexical errors for lengthy identifiers and invalid characters

We recommend adding token types one at a time to **scanner.l**, testing after each addition. Be careful with characters that have special meaning to **flex** such as * and **–** (see docs for how/when to suppress special-ness). The patterns for integers and floats will require careful testing to make sure all cases are covered (see the **man** pages for **strtol**/**strtod** for details on converting strings to numbers). For this assignment, you may assume that all integer constants can be represented by a 32-bit integer. Similarly, you can assume that it is safe to use **strtod** to convert float constants.

Recording the position of each lexeme requires you to track the current line and column numbers (you will need global variables) and update them as the scanner reads the file, mostly likely incrementing the line count on each newline and the column on each token. A tab character accounts for 8 columns. There is code in the starter file that installs a function to be automatically included with each action (that's much nicer that repeating the call everywhere!), and we strongly encourage you to use it for this purpose.

Lastly, you need to be sure that your scanner reports the various lexical errors. The action for an error case should call our **ReportError** class with one of the standard error messages provided in **errors.h** . For each character that cannot be matched to any token pattern, report it and continue parsing with the next character. If an identifier is longer than the GLSL maximum (1023 characters), report the error, truncate the identifier to the first 31 characters (discarding the rest), and continue.

Testing your work

In the starting project, there is a **samples** directory containing various input files and matching **.out** files which represent the expected output. You should **diff** your output against ours as a first step in testing. Now examine the test files and think about what cases aren't covered. Construct some of your own input files to test your scanner even more.

- What lexemes look like numbers but aren't?
- What sequences might confuse the processing of comments?

This is precisely the sort of thought process a compiler writer must go through. Any sort of input is fair game and you'll want to be sure yours can handle anything that comes its way, correctly tokenizing it if possible or reporting some reasonable error if not.

Note that lexical analysis is responsible only for correctly breaking up the input stream and categorizing each token by type. The scanner will accept syntactically bogus sequences such as:

```
int if vec2 + 4.5 [ bool}
```

In your next two assignments, you'll check input programs for these sorts of errors.

For those of you using **vim** as your primary editor, be aware that **vim** automatically appends a newline to the end of any file that you create, which might potentially mess up some of your test cases. If you are editing in the terminal, I would suggest using **emacs** as your editor to avoid this.

General requirements
- Your executable should be named **glc** and should require no command-line arguments. It should process input from **stdin**, write tokens to **stdout**, and report errors to **stderr**.
- Compilers are notorious for leaking memory. Given that they run once and exit, this is not considered much of a problem. We will not examine your programs for leaks and not expect them to free dynamically allocated memory.
- When debugging, you may want to run your program in **valgrind** to confirm that your program does not have any memory corruption errors. If **valgrind** reports any memory leaks, that's fine, but if you're accidentally reading a garbage pointer **valgrind** will be invaluable in helping pin down your error.
- Avoid manually doing any scanning. Although **flex** exports several functions that can let you single-step over the characters of the input file, you should not use this functionality. Instead, try using **flex**'s other features. It's possible to complete this assignment without ever manually inspecting the characters of the input.

Grading
This project is worth 5% of your overall course grade. Most of the points will be allocated for correctness with some consideration for design and readability. We will run your program through the given test files from the samples directory, as well as other tests of our own, using **diff -w** to compare your output to that of our solution.

Deliverables
You are to electronically submit your entire project using and electronic submission process that I'll outline shortly. Be sure to include your **README** file, which is your chance to explain your design decisions and why you believe your program to be correct and robust, as well as describe what to expect from your submission and its error handling.

Good Luck!