CSE340 Spring 2017 Project 1: Lexical Analysis

Due: **Friday, January 27, 2017 by 11:59 pm MST**

The goal of this project is to give you hands-on experience with lexical analysis. You will extend the provided lexical analyzer to support more token types. The next section lists all new token types that you need to implement.

1. Description

Modify the lexer to support the following 3 token types:

```
REALNUM   = (pdigit digit*) DOT digit digit* + 0 DOT digit* pdigit digit*
BASE08NUM = ((pdigit8 digit8*) + 0) (x) (08)
BASE16NUM = ((pdigit16 digit16*) + 0) (x) (16)
```

Where

```
pdigit   = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9
digit    = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9
pdigit8  = 1 + 2 + 3 + 4 + 5 + 6 + 7
digit8   = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7
pdigit16 = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + A + B + C + D + E + F
digit16  = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + A + B + C + D + E + F
```

Note that `NUM` and `DOT` are already defined in the lexer, but here are the regular expressions for the sake of completeness (`DOT` is a single dot character, the quotes are used to avoid ambiguity):

```
NUM = (pdigit digit*) + 0
DOT = '.'
```

The list of valid tokens including the existing tokens in the code would be as follows. This list should be used to determine the token, if the input matches more than one regular expression.

```
 1. IF          8. DIV          15. RBRAC        22. GTEQ
 2. WHILE       9. MULT         16. LPAREN       23. DOT
 3. DO         10. EQUAL        17. RPAREN       24. NUM
 4. THEN       11. COLON        18. NOTEQUAL     25. ID
 5. PRINT      12. COMMA        19. GREATER      26. REALNUM
 6. PLUS       13. SEMICOLON    20. LESS         27. BASE08NUM
 7. MINUS      14. LBRAC        21. LTEQ         28. BASE16NUM
```

2. Instructions

Follow these steps:

- Download the `lexer.cc`, `lexer.h`, `inputbuf.cc` and `inputbuf.h` files accompanying this project description. Note that these files might be a little different from the code you've seen in class or elsewhere.
- Add your code to the files to support the token types listed in the previous section.
- Compile your code using GCC compiler on `CentOS 7`. You will need to use the `g++` command to compile your code in a terminal window. See section 4 for more details on how to compile using GCC.

  Note that you are required to compile and test your code in CentOS 7 using the GCC compiler. You are free to use any IDE or text editor on any platform, however, using tools available in CentOS (or tools that you could install on CentOS) could save time in the development/compile/test cycle.
- Test your code to see if it passes the provided test cases. You will need to extract the test cases from the zip file and run the test script `test1.sh`. See section 5 for more details.
- Submit your code on the course submission website before the deadline. You can submit as many times as you need. Make sure your code is compiled correctly on the website, if you get a compiler error, fix the problem and submit again.

Keep in mind that

- You should use C/C++, no other programming languages are allowed.
- All programming assignments in this course are individual assignments. Students must complete the assignments on their own.
- You should submit your code on the course submission website, no other submission forms will be accepted.
- You should familiarize yourself with the CentOS environment and the GCC compiler. Programming assignments in this course might be very different from what you are used to in other classes.

3. Evaluation

The submissions are evaluated based on the automated test cases on the submission website. Your grade will be proportional to the number of test cases passing. If your code does not compile on the submission website, you will not receive any points.

NOTE: The next two sections apply to all programming assignments.

You should use the instructions in the following sections to compile and test your programs for all programming assignments in this course.

4. Compiling your code with GCC

You should compile your programs with the GCC compilers which are available in CentOS 7. GCC is a collection of compilers for many programming languages. There are separate commands for compiling C and C++ programs:

- Use the `gcc` command to compile C programs
- Use the `g++` command to compile C++ programs

Here is an example of how to compile a simple C++ program:

```
$ g++ test_program.cpp
```

If the compilation is successful, it will generate an executable file named `a.out` in the same folder as the program. You can change the output file name by specifying the `-o` option:

```
$ g++ test_program.cpp -o hello.out
```

To enable C++11 with `g++`, use the `-std=c++11` option:

```
$ g++ -std=c++11 test_program.cpp -o hello.out
```

The following table summarizes some useful GCC compiler options:

Switch	Can be used with	Description
`-o path`	`gcc`, `g++`	Change the filename of the generated artifact
`-g`	`gcc`, `g++`	Generate debugging information
`-ggdb`	`gcc`, `g++`	Generate debugging information for use by GDB
`-Wall`	`gcc`, `g++`	Enable most warning messages
`-w`	`gcc`, `g++`	Inhibit all warning messages
`-std=c++11`	`g++`	Compile C++ code using 2011 C++ standard
`-std=c99`	`gcc`	Compile C code using ISO C99 standard
`-std=c11`	`gcc`	Compile C code using ISO C11 standard

You can find a comprehensive list of GCC options in the following page:

https://gcc.gnu.org/onlinedocs/gcc-4.8.5/gcc/

Compiling projects with multiple files

If your program is written in multiple source files that should be linked together, you can compile and link all files together with one command:

```
$ g++ file1.cpp file2.cpp file3.cpp
```

Or you can compile them separately and then link:

```
$ g++ -c file1.cpp
$ g++ -c file2.cpp
$ g++ -c file3.cpp
$ g++ file1.o file2.o file3.o
```

The files with the `.o` extension are object files but are not executable. They are linked together with the last statement and the final executable will be `a.out`.

You can replace `g++` with `gcc` in all examples listed above to compile C programs.

5. Testing your code on CentOS

Your programs should not explicitly open any file. You can only use the **standard input** e.g. `std::cin` in C++, `getchar()`, `scanf()` in C and **standard output** e.g. `std::cout` in C++, `putchar()`, `printf()` in C for input/output.

However, this restriction does not limit our ability to feed input to the program from files nor does it mean that we cannot save the output of the program in a file. We use a technique called standard IO redirection to achieve this.

Suppose we have an executable program `a.out`, we can run it by issuing the following command in a terminal (the dollar sign is not part of the command):

```
$ ./a.out
```

If the program expects any input, it waits for it to be typed on the keyboard and any output generated by the program will be displayed on the terminal screen.

To feed input to the program from a file, we can redirect the standard input to a file:

```
$ ./a.out < input_data.txt
```

Now, the program will not wait for keyboard input, but rather read its input from the specified file. We can redirect the output of the program as well:

```
$ ./a.out > output_file.txt
```

In this way, no output will be shown in the terminal window, but rather it will be saved to the specified file. Note that programs have access to another standard stream which is called standard error e.g. `std::cerr` in C++, `fprintf(stderr, ...)` in C. Any such output is still displayed on the terminal screen. It is possible to redirect standard error to a file as well, but we will not discuss that here.

Finally, it's possible to mix both into one command:

```
$ ./a.out < input_data.txt > output_file.txt
```

Which will redirect standard input and standard output to `input_data.txt` and `output_file.txt` respectively.

Now that we know how to use standard IO redirection, we are ready to test the program with test cases.

Test Cases

A test case is an input and output specification. For a given input there is an *expected* output. A test case for our purposes is usually represented by two files:

- `test_name.txt`
- `test_name.txt.expected`

The input is given in `test_name.txt` and the expected output is given in `test_name.txt.expected`.

To test a program against a single test case, first we execute the program with the test input data:

```
$ ./a.out < test_name.txt > program_output.txt
```

The output generated by the program will be stored in `program_output.txt`. To see if the program generated the expected output, we need to compare `program_output.txt` and `test_name.txt.expected`. We do that using a general purpose tool called `diff`:

```
$ diff -Bw program_output.txt test_name.txt.expected
```

The options `-Bw` tell `diff` to ignore whitespace differences between the two files. If the files are the same (ignoring the whitespace differences), we should see no output from `diff`, otherwise, `diff` will produce a report showing the differences between the two files.

We would simply consider the test **passed** if `diff` could not find any differences, otherwise we consider the test **failed**.

Our grading system uses this method to test your submissions against multiple test cases. There is also a test script accompanying this project `test1.sh` which will make your life easier by testing your code against multiple test cases with one command.

Here is how to use `test1.sh` to test your program:

- Store the provided test cases zip file in the same folder as your project source files
- Open a terminal window and navigate to your project folder
- Unzip the test archive using the `unzip` command:

  ```
  $ unzip test_cases.zip
  ```

  NOTE: the actual file name is probably different, you should replace `test_cases.zip` with the correct file name.

- Store the `test1.sh` script in your project directory as well
- Make the script executable:

  ```
  $ chmod +x test1.sh
  ```

- Compile your program. The test script assumes your executable is called `a.out`
- Run the script to test your code:

  ```
  $ ./test1.sh
  ```

The output of the script should be self explanatory. To test your code after you make changes, you will just perform the last two steps (compile and run `test1.sh`).