CPSC 326: Homework Assignment 2 *Due: Thursday, Feb. 2*

Goals:

- Implement the MyPL lexical analyzer;
- Ensure development environment is setup to run Google test framework;
- Practice working with unit tests.

You are free to use whatever IDE and machine you prefer for this class. However, to complete the assignment, you will need `git`, `g++` (version 11 or higher), `cmake`, `make`, the google test framework, and `valgrind` installed. It can also be useful to have a debugger such as `gdb` installed as well. Each of these are already installed on the remote development server (`ada.gonzaga.edu`) provided by the CS Department. However, you may also install these programs on your own machine, via a virtual machine, running WSL2, or on your own remote server. Note that you will also need a GitHub account for obtaining starter code and for submitting your assignment.

Instructions:

1. Use the GitHub Classroom link (posted in Piazza) to copy the starter code into your own repository. Clone the repository in the directory where you will be working on the assignment (e.g., onto `ada` or your own machine).
2. Modify your `mypl.cpp` program for HW-2 (see below).
3. Write the `next_token()` function in `lexer.cpp`.
4. Ensure your code passes the unit tests provided in `lexer_tests.cpp` within the `tests` subdirectory.
5. Ensure your lexer implementation correctly handles the example file (`tokens.mypl`).
6. You must create addition example test files for this assignment along with ensure your program works correctly from standard input. Include the tests you used in your homework writeup.
7. Create a short write up as a **pdf file** named `hw2-writeup.pdf`. For this assignment, your write up should provide a short description of any challenges and/or issues you faced in finishing the assignment and how you addressed them along with how you tested your program. Include any additional files you used for testing. Be sure your `hw2-writeup.pdf` file is in the main directory of your assignment (and **not** within the `src` directory or any other subdirectory).
8. Submit your program. Be sure to add, commit, and push all assignment files to your GitHub repo. You can verify that your work has been submitted via the GitHub page for your repo.

Additional Requirements: Note that in addition to items listed below, details will also be discussed in class, in lecture notes, and in the discussion section.

1. Modify your `mypl.cpp` file from HW-1 so that the `--lex` flag calls your lexer (supporting both standard input and file input) and prints out the corresponding tokens (one token per line). Your code for calling the lexer and printing the result should look something like the following.

```
try {
  Token t = lexer.next_token();
  cout << to_string(t) << endl;
  while (t.type() != TokenType::EOS) {
    t = lexer.next_token();
    cout << to_string(t) << endl;
  }
} catch (MyPLException& ex) {
  cerr << ex.what() << endl;
}
```

2. It is fine to implement the `next_token()` function without breaking it into separate helper functions (i.e., you can have it be one large function). However, if you would like to "modularize" it, you are welcome to.

3. You must implement your `next_token()` function by reading one character at a time via the `read()` and `peek()` helper functions provided in `lexer.h` and `lexer.cpp`. In addition, to report errors, you must use the `error()` helper function provided by the Lexer class.

4. You will want to use `EOF` as well as the C++ `isspace()` helper function used in HW-1. You will also want to use the C++ `isdigit()` and `isalpha()` helper functions to check for digits and letters, respectively.

5. The full set of token types for MyPL are provided in the `token.h` file. Note that your `next_token()` function is creating and returning `Token` objects with these types.

6. Note that `next_token()` uses an iterator, i.e., stream-based, model. This means that one call to `next_token()` returns only the next token in the input. The `Lexer` object maintains state, including where it is in the current input file (to be able to return the next token in the input, and so on).

7. The `Lexer` class has `line` and `column` member variables. These variables keep track of the current line and column for building tokens. Your `next_token()` function will need to update these member variables and also use them to build up new token objects.

8. Each token should have a non-empty lexeme. For tokens with "unimportant" lexemes, you can just use their corresponding symbol. For example, the lexeme for `+` should be `"+"` and the lexeme for `int` should be `"int"`.

9. The `tokens.out` file within the `example` subdirectory gives an example of the result of running `./mypl --lex` on the `examples/tokens.mypl` file. Your program must output the exact same information as what is in `tokens.out` to be considered correct.

10. A non-comprehensive set of unit tests are provided in the file `lexer_tests.cpp` within the `tests` subdirectory. Using the `cmake` file provided with the starter code, running `make` will both build the

`mypl` executable and an executable named `lexer_tests`. Running `lexer_tests` will execute the unit tests on your implementation. Your implementation will also need to pass all of the unit tests from `lexer_tests` to be considered correct.

Hints and Tips:

1. As mentioned in HW-1, you need to be careful with Windows style newlines. At a minimum, you must support UNIX/Linux style newlines (which are denoted by the linefeed character `'\n'`). You may also want to also support Windows-style newlines (denoted by a carriage return and linefeed, i.e., `'\r\n'`), however, it is not required for the assignment. As a warning, however, if you use an editor that adds carriage returns, your program may not work correctly on test files you create.
2. The basic layout for `next_token()` that I used in my implementation is, in order: (1) read all whitespace and comments (checking for EOF); (2) check for EOF; (3) check for single character tokens (e.g., arithmetic operators, punctuation, etc.); (4) check for the trickier symbols that can involve or require two characters (e.g., < vs <=, `!=`, and so on); (5) check for character values (tricky cases here involve quoted characters like `` `\n' ``; (6) check for string values; (7) check for integer and double values; (8) check for reserved words; and then (9) identifiers. Again, it is much easier to do this incrementally as opposed to all at once and then try to debug. Note that you should structure your `next_token()` function to avoid receiving a compile warning regarding not returning a value at the end of the function. (That is, you should make sure you don't have any compiler warnings.)
3. You are encouraged (but not required) to create your own unit tests. Note that the unit tests provided are not guaranteed to be comprehensive. In general, it is a good engineering practice to create unit tests for all code you write, and so giving it a try with this homework is an opportunity for you to practice writing unit tests.

Homework Submission and Grading. Your homework will be graded using the files you have pushed to your GitHub repository. Thus, you must ensure that all of the files needed to compile and run your code have been successfully pushed to your GitHub repo for the assignment. Note that this also includes your homework writeup. This homework assignment is worth a total of 30 points. The points will be allocated according to the following.

1. **Correct and Complete (24 points).** Your homework will be evaluated using a variety of different tests (for most assignments, via unit tests as well as test runs using specific files). Each failed test will result in a loss of 2 points. If 10 or more tests fail, but some tests pass, 4 points (out of the 24) will be awarded as partial credit. Note that all 24 points may be deducted if your code does not compile, large portions of work are missing or incomplete (e.g., stubbed out), and/or the specified techniques, design, or instructions were not followed.
2. **Evidence and Quality of Testing (2 points).** For each assignment, you must provide additional tests that you used to ensure your program works correctly. Note that for most assignments, a specific set of tests will be requested. A score of 0 is given if no additional tests are provided, 1 if the tests are only partially completed (e.g., missing tests) or the tests provided are of low quality, and 2 if the minimum number of tests are provided and are of sufficient quality.

3. **Clean Code (2 points).** In this class, "clean code" refers to consistent and proper code formatting (indentation, white space, new lines), use of appropriate comments throughout the code (including file headers), no debugging output, no commented out code, meaningful variable names and helper functions (if allowed), and overall well-organized, efficient, and straightforward code that uses standard coding techniques. In addition, when compiled, your code should not have any warnings. A score of 0 is given if there are major issues, 1 if there are minor issues, and 2 if the "cleanliness" of the code submitted is satisfactory for the assignment.

4. **Writeup (2 points).** Each assignment will require you to provide a small writeup addressing challenges you faced and how you addressed them as well as an explanation of the tests you developed. Homework writeups do not need to be long, and instead, should be clear and concise. A score of 0 is given if no writeup is provided, 1 if parts are missing, and 2 if the writeup is satisfactory.