CPSC 326: Homework Assignment 1 *Due: Thursday, Jan. 26*

Goals:

- Set up your programming and submission environment for MyPL assignments;
- Practice building and running programs using `cmake` and `make`;
- Refresh your `C++` skills by writing a basic command-line interface (CLI) program.

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`, 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, 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. Finish the `mypl.cpp` program file within the `src` directory in the downloaded starter code.
3. Ensure your program works correctly and does the necessary tasks explained below.
4. Test your implementation on the examples below and using new examples you create.
5. Create a short write up as a **pdf file** named `hw1-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. You must include additional files you used for testing along with the additional test cases you used for checking that standard input works (see below). Be sure your `hw1-writeup.pdf` file is in the main directory of your assignment (and **not** within the `src` directory or any other subdirectory).
6. 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.

Detailed Requirements: The `mypl.cpp` file will contain the `main` function for our eventual MyPL interpreter implementation. It will handle the various command-line options available for running the interpreter. For this assignment your job is to provide a basic skeleton for the `mypl` interpreter program, which we will fill in as we implement different parts of the interpreter pipeline. The main way that we'll call the interpreter is by passing it a MyPL source file. However, the `mypl` command can also be run (in a basic way) using standard input. The interpret will also support an option flag, which can be one of a

set of different execution "modes" (described below), along with a "normal" execution mode, which is the default case used when no flag is provided.

1. *The* `mypl` *program for HW-1.* The `mypl` program supports six different options (modes) along with a default mode. In addition, your program should allow for either file input or standard input. That is, if no file is provided (regardless of the mode), standard input should be used. The six options are: `--help`, `--lex`, `--parse`, `--print`, `--check`, and `--ir`. For this homework assignment, the options will do the following.

   - `--help` should print a "usage" message.
   - `--lex` should print the first character of the input.
   - `--parse` should print the first two characters of the input.
   - `--print` should print the first word of the input.
   - `--check` should print the first line of the input.
   - `--ir` should print the first two lines of the input.
   - The default mode (no option given) should print the entire file.

   The following shows what the output should be for each option (including no option) using the `example.txt` file provided in the starter code within the `examples` subdirectory. Note that the `mypl` command is assumed to be executed below from the command line.

```
./mypl --help
Usage: ./mypl [option] [script-file]
Options:
  --help     prints this message
  --lex      displays token information
  --parse    checks for syntax errors
  --print    pretty prints program
  --check    statically checks program
  --ir       print intermediate (code) representation

./mypl --lex examples/example.txt
[Lex Mode]
T

./mypl --parse examples/example.txt
[Parse Mode]
Th

./mypl --print examples/example.txt
[Print Mode]
There
```

```
./mypl --check examples/example.txt
[Check Mode]
There sat that beautiful big machine whose sole job was to copy things

./mypl --ir examples/example.txt
[IR Mode]
There sat that beautiful big machine whose sole job was to copy things
and do addition.  Why not make the computer do it? That's why I sat

./mypl examples/example.txt
[Normal Mode]
There sat that beautiful big machine whose sole job was to copy things
and do addition.  Why not make the computer do it? That's why I sat
down and wrote the first compiler. It was very stupid. What I did was
watch myself put together a program and make the computer do what I
did. -Grace Hopper
```

Note that if no input file is provided, your program should read from standard input. For `--lex`, `--parse`, and `--print` modes, after typing the `enter` key, the output will be displayed. For the `--ir` mode, the output will be printed after the user types two lines of text. For the "default" mode, `Control-d` can be used to finish entering text after which the results will be printed (where `Control-d` adds an EOF character to the input stream). Below, the input is marked with "`<- input`", which is **not** included as part of the input itself, but instead is used below to help denote what the input text is versus the output text.

```
./mypl --lex
[Lex Mode]
foo bar baz                                          <- input
f

.n/mypl --parse
[Parse Mode]
foo bar baz                                          <- input
fo

./mypl --print
[Print Mode]
foo bar baz                                          <- input
foo
```

```
./mypl --check
[Check Mode]
foo bar baz                                          <- input
foo bar baz

./mypl --ir
[IR Mode]
foo bar baz                                          <- input
qux quux quuz corge                                  <- input
foo bar baz
qux quux quuz corge

./mypl
[Normal Mode]
foo foo foo                                          <- input
bar bar bar                                          <- input
baz baz baz                                          <- input
foo foo foo
bar bar bar
baz baz baz
```

Finally, if an invalid option or file is given, the program should report an error.

```
./mypl --fix
ERROR: Unable to open file '--fix'

./mypl examples/nosuchfile.txt
ERROR: Unable to open file 'examples/nosuchfile.txt'
```

If too many parameters are provided, the usage information should be printed.

```
./mypl a b c
Usage: ./mypl [option] [script-file]
Options:
  --help     prints this message
  --lex      displays token information
  --parse    checks for syntax errors
  --print    pretty prints program
  --check    statically checks program
  --ir       print intermediate (code) representation
```

Note that the above explains what the modes will do once we implement the various parts of MyPL.

2. *Building* `mypl` *using* `cmake` *and* `make`. To compile (build) your source code and create the `mypl` executable, you need to first run the following `cmake` command within the main directory (where the `CMakeLists.txt` file is located):

```
cmake .
```

The following shows the output provided by `cmake` after the command succeeds. Note that if there are issues in your setup, or if files are missing, the command will report errors.

```
-- The C compiler identification is GNU 11.3.0
-- The CXX compiler identification is GNU 11.3.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/bowers/cpsc326/src/hw1
```

Once `cmake` successfully completes, run `make` to compile the `mypl` executable:

```
make
```

The first time you run `make` you should see the following:

```
[ 50%] Building CXX object CMakeFiles/mypl.dir/src/mypl.cpp.o
[100%] Linking CXX executable mypl
[100%] Built target mypl
```

Note that if you have any syntax errors, they will be shown as a result of running `make`. After you modify your source code, rerun `make` to rebuild your code. Note that you should only have to run `cmake` once (whereas you will typically run `make` many times). After `make` succeeds, you will see the `mypl` executable file within the "main" directory.

3. *Reading text from the input.* For this assignment, you **must** use the `C++ istream` library's `get()` and `peek()` functions to obtain stream characters. Note that both `get()` and `peek()` return a single character from the input stream. This means getting characters, words, and lines must be done one character at-a-time. The reason for this is that you will use a similar approach to implement the Lexer in HW-2.

4. *Additional hints for implementing HW-1.*

- This assignment can be fully written within the `main` function, but you can define additional helper functions as you see fit (although it isn't necessary). In my implementation, e.g., I defined a `void usage(const string& command)` function to print the usage information for the `--help` flag and when incorrect arguments are given.
- Command line arguments are passed in C++ via `main`'s `int argc` and `char* argv[]` (or `char** argv`) parameters. Note that `argc` gives the number (or count) of the command-line arguments provided and `argv` gives an array of the argument values. The first argument (i.e., `argv[0]`) is the name of the command used to run the program (thus `argc` is always 1 or greater).
- To make things easier, I converted each argument in `argv` to a `string` value held in an array `args`. You do not need to do this, but it can make things a bit simpler especially if you are more familiar working with `string` values compared to "`C` strings" in C++.
- To abort (exit) the program on error—e.g., if the wrong number of arguments are provided—you can use a return statement such as "`return 1`".
- In C++, `cin` can be assigned to a variable of type `istream` but only by reference or via a pointer, such as:

```
istream* input = &cin;
```

- To open a file you can use `ifstream` (input file stream) as follows, assuming `input` is of type `istream*`.

```
input = new ifstream(argv[1]); // assuming file name is second argument
```

- To check if a file exists, you can use the `fail()` function:

```
if (input->fail())
```

- The `get()` method on an `istream` object returns an integer that can be cast to a char value.

```
char ch = input->get();
```

- You can also get the next character in the stream by using `peek()`. Note that `peek()` leaves the character in the input stream, unlike `get()`, which removes the character from the stream.

```
char ch = input->peek();
```

- To test if a newline is entered, you can compare to the newline character.

```
if (ch == '\n')
```

- The `isspace()` function can be used to check for whitespace (space, tab, newline, and so on).

```
if (isspace(ch))
```

- Finally you can check for the end-of-file by comparing to the EOF constant.

```
if (ch == EOF)
```

- Some editors in Windows will use a carriage return followed by a line feed for denoting newlines (as opposed to just line feeds as in Linux and MacOS). In many editors, including VS Code on Windows, you can change the default so that it doesn't output carriage returns.

- You must make sure your program does not have any memory leaks. You can check for leaks using `valgrind`.

  ```
  valgrind ./mypl examples/example.txt
  ```

  The `valgrind` tool prints various diagnostic messages, which includes a "heap summary" that should say that "All heap blocks were freed – no leaks are possible". Note that you will need to ensure no leaks are possible for both file input and standard input.

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. 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.