CPSC 326: Homework Assignment 4 *Due: Tuesday, Feb. 28*

Goals:

- Develop a better understanding of syntax analysis as a front-end compilation stage.
- Extend the initial simple parser for MyPL to build an AST.
- Practice working with the visitor pattern for traversing and coding against the AST.
- Practice writing unit tests and develop a full suite of parser regression 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. Copy your lexer implementation (`lexer.h` and `lexer.cpp`) from HW-2, your simple parser implementation (`simple_parser.h` and `simple_parser.cpp`) from HW-3, and your `mypl.cpp` file from HW-3 into the `src` directory for HW-4. Your lexer is required for your ast parser, and the simple parser is required for your mypl program.
3. Modify your `mypl.cpp` program from HW-3 (see below).
4. Complete `ASTParser` by copying the various recursive descent functions you used in `SimpleParser` into `ast_parser.h` and `ast_parser.cpp`, and then augmenting the functions to build up AST objects as discussed in class. Note that the `ast.h` file in the `src` directory provides the definition of the various AST classes. You are not allowed to modify the AST classes for HW-4.
5. Complete the `PrintVisitor` class to pretty print MyPL source code. See below for additional requirements for printing.
6. Ensure your code passes the unit tests provided in `ast_parser_tests.cpp` and `parser_syntax_tests.cpp` within the `tests` subdirectory.
7. Ensure your parser correctly handles the example files `print-1.mypl`, `print-2.mypl`, and `print-3.mypl` within the `examples` subdirectory. Note that the output your program should produce is given in `print-1.out`, `print-2.out`, and `print-3.out`, respectively. You can use the command-line `diff` tool to quickly check if your output is the same.

8. Create ten additional unit tests as specified in `ast_parser_tests.cpp`. Include a description of the tests you created in your homework writeup.

9. Create a short write up as a **pdf file** named `hw4-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 your new unit tests. Be sure your `hw4-writeup.pdf` file is in the main directory of your assignment (and **not** within the `src` directory or any other subdirectory).

10. 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-2 so that the `--print` flag calls your ast parser and the print visitor. Your code for calling these should look something like the following. Note that the snippet below assumes that the lexer has already been created.

```
try {
  ASTParser parser(lexer);
  Program p = parser.parse();
  PrintVisitor v(cout);
  p.accept(v);
} catch (MyPLException& ex) {
  cerr << ex.what() << endl;
}
```

2. When writing your "pretty printer" (implementing `PrintVisitor`), you must use the following `MyPL` code styling rules (also see the test cases provided separately in the examples subdirectory).

   (a) Indent all statements within a block. The indentation should add two spaces at each indentation level.

   (b) Each statement should be on a separate line without blank lines before or after the statement. The exceptions to this rule are struct and function definitions.

   (c) Format variable declarations with one space separating each component, i.e., as *type id* = *expr* for non-array variables and `array` *type id* = *expr* for array variables.

   (d) Format variable assignments with one space before and after the assignment symbol, i.e., as *id* = *expr*.

   (e) Format struct definitions such that the reserved word `struct`, the type name, and the opening brace appear on one line, with one space between each, each field is indented (two spaces from the start of `struct`) and on a separate line (with no blanks between), if a comma is needed it occurs directly after the variable name, and the ending brace is on the next immediate line after the last variable declaration and aligned with `struct`. There should be one blank line before a type declaration. Here is an example:

```
struct Employee {
  int yr_hired,
  string name,
  Employee manager
}
```

(f) Format function declarations such that the return type, the function name, the parameter list, and the opening brace are all on the same line, the body of the function is indendented appropriately, and the closing brace is on a separate line, immediately following the last body statement, aligned with the function return type. There should be one blank line before each function declaration. Here is an example:

```
int add(int x, int y) {
  sum = x + y
  return sum
}
```

(g) Format while statements such that `while`, the boolean expression (in parenthesis), and the opening brace occur on the same line, there is one space before and one space after the start and end parentheses, the body of the while loop is appropriately indendented (with each statement on a separate line), and the closing brace is aligned with `while` and occurs on the line immediately after the last statement of the body. Here is an example:

```
while (flag) {
  i = i + 1
  j = j - 1
}
```

(h) Format for statements such that `for`, the variable declaration, the condition, the assignment, and the opening brace occur on the same line. There must be one space before and one space after the and end parentheses. There is also one space before the condition and one space before the assignment. The body of the for loop must be appropriately indendented (with each statement on a separate line), and the closing brace is aligned with `for` and occurs on the line immediately after the last statement of the body. Here is an example:

```
for (int i = 0; i < n; i = i + 1) {
  j = j * 2
}
```

(i) Format if-elseif-else statements similar to while statements such that the body of each section is indented, `elseif` and `else` statements appear on seperate lines (with no blank lines before or after), opening braces appear on the same line as its corresponding `if`, `elseif`, or `else`, and closing braces appear on separate lines with no preceding blank lines. Here is a simple example:

```
if (x < 0) {
```

```
  print("negative")
}
elseif (x == 0) {
  print("zero")
}
else {
  print("positive")
}
```

(j) Format simple expressions (basic rvalues) without any extra spaces. Path expressions should not contain spaces between corresponding dots (e.g., `x.y.z`), and similarly for array expressions (e.g., `x[0]`).

(k) Format complex expressions with spaces between their corresponding parts. For example, if the original was written as `3+4+5` the pretty-printed version should be written as `3 + 4 + 5`.

(l) Print parentheses that were in the original input. For example, if the original was written as `(3+4)+5`, print `(3 + 4) + 5`.

(m) Boolean expressions should follow the same rules as for complex expressions except for the case of `not`, which should have the entire expression (after the `not`) parenthesized. For example, `not (x>1) and (y>1)` should print as `(not ((x > 1) and (y > 1)))`.

(n) Format struct object creation such that there is one space between `new` and the type name (e.g., `new Employee`). Similarly, for array creation, place the brackets and array size together with no spaces, e.g., `new Employee[10]`. The expression determining the size should be printed following the above expression rules.

(o) Format function calls such that the function name is immediately followed by an opening parenthesis, followed by a comma-separated list of expressions, followed by a closing parenthesis. There should be one space after each comma, e.g., `f(a, b, c)`.

(p) Additional examples can be found in the `print-1`, `print-2`, and `print-3` files under the examples directory.

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.