CPSC 326: Homework 3 *Due By Mon., Mar. 4*

Goals:

- Extend the basic recursive descent parser for MyPL from HW-2 to build the AST objects;
- Practice working with and writing your own unit tests.

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.
2. Copy the various homework files from `HW-2` that you will need to run your program, except `mypl.py` and `mypl`. You *must* ensure that all files needed to run your code are added, committed, and pushed to your repository. If the graders try to run your code and there are missing files (which make it so that your parser can't run), you will receive a 0 for the assignment.
3. Complete the implementation of the `ASTParser` class in the given `mypl_ast_parser.py` file.
4. Ensure your code passes the unit tests provided in `hw3_tests.py`. Note that you should also copy your HW2 tests, change the parser from `SimpleParser` to `ASTParser` and make sure these tests pass as well.
5. Create additional unit tests as specified at the bottom of `hw3_tests.py`. By interesting we mean that the tests should be non-trivial, different to existing tests given, and be useful to ensure correctness of your implementation.
6. Complete the implementation of the `PrintVisitor` class in `mypl_printer.py` file. See additional information below.
7. Ensure your pretty printer implementation correctly handles the example files within the `examples` subdirectory. Note that running `./mypl --print` over these files can be compared to the corresponding `.out` files given in the `examples` directory for correctness. The easiest way to check these is by using the `diff` program.
8. Create a short write up as a **pdf file** named `hw3-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 a brief description of each of the unit tests you created and why you think they are "interesting".
9. Submit your program by ensuring all of your code, test file, and writeup is pushed 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 and in lecture notes.

1. 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), 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 after a type declaration. Note that the semicolon should occur without any space before it. 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 after each function declaration. (Note the code for doing functions is given to you.) 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. The arguments passed into a struct "constructor" should be formatted similarly to function calls.

(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)`. The arguments, which are expressions, should follow the expression rules above.

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

2. You can not deviate from the general visitor approach and functions specified in the starter code. Similarly, you may not modify any of the AST classes provided.

3. You will need to allow yourself enough time to think through some of the trickier parts of the parser and print visitor. If you start this assignment too close to the deadline you will likely run out of time.

4. If you use any print statements for debugging, you must remove these from your final solution. In addition, you must remove all commented out code from your final submission.

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 40 points. The points will be allocated according to the following.

1. **Correct and Complete (30 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 input files). Each failed test will result in a loss of 2 points. If 15 or more tests fail, but some tests pass, 6 points (out of the 30) will be awarded as partial credit. Note that all 30 points may be deducted if your code does not run, 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 (5 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, 1–4 points if the tests are only partially completed (e.g., missing tests) or the tests provided are of low quality, and 5 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, 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 (3 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. Additional items may also be requested depending on the assignment. 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.