CPSC 326: Homework 3 *Due: Feb. 26*

Goals:

- Continue with recursive descent parsing, extending HW-2 to build the Abstract Syntax Tree (AST).
- Practice using the Visitor pattern by implementing visitor functions to "pretty print" the AST.
- Continue practicing working with unit tests, creating tests, etc.

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 all of the files from your HW2 `src/main/java/cpsc326` directory, *except* for `MyPL.java`, into your HW3 `src/main/java/cpsc326` directory.
3. Complete the `ASTParser` recursive descent implementation in `ASTParser.java`. For this, you should first copy your recursive descent functions from HW-2 into `ASTParser`. Then you should extend your functions to build the AST (as was done in-class throught the simpler examples).
4. Ensure your code passes the unit tests provided in `ASTParserTests.java` and `ASTParserSyntaxTests.java`. (Note you will want to do steps 2 and 3 iteratively.) The `ASTParserTests` check that the correct AST objects are built. The `ASTParserSyntaxTests` are regression tests from HW-2 for just checking syntax.
5. Complete the `PrintVisitor` implementation in `PrintVisitor.java`. For this task, you will need to implement each of the visitor functions.
6. Ensure your AST parser and Print Visitor implementations correctly handle the example files within the `examples` subdirectory. Note that, like in HW-1, the correct output for each file is given as a `.out` file.
7. Create additional unit tests as specified in the TODO comment at the end of `ASTParserTests.java`.
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 the unit tests you created and any challenges and/or issues you faced in finishing the assignment and how you addressed them. The description of the tests can be short, but should state why you designed the tests the way you did (i.e., justify why the test is non-trivial / interesting, and what it is actually testing).
9. Submit your program by ensuring all of your code 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. For example: `var` ␣ *id*`:` ␣ *type* ␣ `=` ␣ *expr* for the "full" variant of variable declarations.

(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 field and aligned with `struct`. There should be one blank line after a struct declaration. Here is an example:

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

(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 (like with structs). Fields should be formatted with the id, the colon (with no space between id and colon), a space, then the type. Here is an example:

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

(g) Format while statements such that `while`, the boolean expression, and the opening brace occur on the same line, there is one space before and after the conditional expression, 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 = j
}
```

(h) Format for statements such that `for`, the variable, `from`, the first expression, `to`, the second condition, and the open brace are on the same line and each separated by a space. 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 i from 0 to n {
  println(i)
}
```

(i) Format if-else statements similar to while statements. Statements with an `else if` should be formatted similarly to an `if`. Here is a simple example:

```
if flag1 {
  print(0)
}
else if flag2 {
  print(1)
}
else {
  print(2)
}
```

(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`. Note the following regarding parentheses.

(l) Parenthesize all binary expressions. For example, `3 + 4` should be printed as `(3 + 4)`. Similarly, `3 + 4 + 5` should be printed as `(3 + (4 + 5))`.

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

(n) Boolean expressions should follow the same rules as for expressions above except for the case of `not`, which should have the entire expression (after the `not`) parenthesized (even if it will already be parenthesized as a binary expression). For example, `not flag` would print as `not (flag)`, `not(true or false)` as `not ((true or false))`, and `not (x>1) and (y>1)` as `not ((x > 1) and (y > 1))`.

(o) 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 (below).

(p) 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.

(q) Array types should be formatted without spaces after the square brackets. For example, the input `[ int ]` should be printed as `[int]`.

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 must use the helper functions provided in the `PrintVisitor` class for adding newlines, outputing pretty-printed code (i.e., by using `write`), incrementing and decrementing indentation (these are very helpful to use!), and printing the current indent (also very useful). Hint: before you print a block of code, you will want to call `incIndent()` and then once finished `decIndent()`. Similarly, you will need to write an indent via `write(indent())` prior to printing statements, etc.

4. Be sure to have the AST diagram for MyPL handy. This is useful as you both create AST objects and as you write the pretty printer (visitor functions).

5. 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. This assignment is one of the harder ones of the class and so it is also worth more points.

6. 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 30 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 4 points. If 6 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. Because assignments build on each other, in most cases you will need all tests to pass before moving to the next assignment.

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.