CPSC 326: Homework 2 *Due: Feb. 12*

Goals:

- Understand how to build a recursive descent parser.
- Implement the MyPL "simple" parser (for checking syntax).
- Practice 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 HW1 `src/main/java/cpsc326` directory, *except* for `MyPL.java`, into your HW2 `src/main/java/cpsc326` directory.
3. Complete the `SimpleParser` recursive descent implementation in `SimpleParser.java`.
4. Ensure your code passes the unit tests provided in `SimpleParserTests.java`. (Note you will want to do steps 2 and 3 iteratively.)
5. Ensure your parser implementation correctly handles the example file within the `examples` subdirectory. Note that the file is syntactically well-formed.
6. Create additional unit tests as specified in the TODO comment at the end of `SimpleParserTests.java`.
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 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).
8. 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. Before starting, be sure you read through and understand the MyPL grammar, which is given on the last page.
2. Before implementing your recursive descent functions, be sure you fully understand all of the helper functions provided in the `SimpleParser.java` file. You must use these in your implementation, as opposed to calling `nextToken()`, etc.
3. You must use the `eat()` helper function when necessary, but should not over use it. In many cases `advance()` is all that is needed, namely, when you already know the type of token in the stream (e.g., after a `match()` call).

4. You should generally follow the pattern of one recursive descent function per non-terminal. It is okay to deviate *slightly* in terms of the choice of recursive descent functions, but overall they should follow the general pattern described in class and induced by the grammar.

5. You may not add additional helper functions in your implementation. You also cannot deviate from the general recursive descent approach in your implementation (see previous point).

6. The MyPL grammar is not $LL(1)$, and so there are a few cases that are a bit trickier. Since the MyPL grammar requires more than one lookahead in a few places, you will need to adapt some of your recursive descent functions respectively. In my implementation, e.g., there are a few functions that assume than an `ID` token has already been read. Be sure to start early so that you leave yourself enough time to finish the entire parser.

7. The example file (in the `examples` folder) is syntactically well-formed. Running the parser on it should result in no output. To run the parser, use the following:

```
./mypl -m PARSE examples/hw1_parser.mypl
```

8. Make sure you add your name to the top of the `SimpleParser.java` file.

9. 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 (20 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 5 or more tests fail, but some tests pass, 4 points (out of the 20) will be awarded as partial credit. Note that all 20 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.

The MyPL Syntax Rules

```
    <program>  ::=  ( <struct_def> | <fun_def> )*
 <struct_def>  ::=  STRUCT ID LBRACE <fields> RBRACE
     <fields>  ::=  <field> ( COMMA <field> )* | ε
      <field>  ::=  ID : <data_type>
    <fun_def>  ::=  <return_type> ID LPAREN <params> RPAREN <block>
      <block>  ::=  LBRACE ( <stmt> )* RBRACE
<return_type>  ::=  <data_type> | VOID_TYPE
     <params>  ::=  <param> ( COMMA <param> )* | ε
      <param>  ::=  ID : <data_type>
  <data_type>  ::=  <base_type> | ID | LBRACKET ( <base_type> | ID ) RBRACKET
  <base_type>  ::=  INT_TYPE | DOUBLE_TYPE | STRING_TYPE | BOOL_TYPE
       <stmt>  ::=  <var_stmt> | <while_stmt> | <if_stmt> | <for_stmt> | <return_stmt> |
                    <assign_stmt> | <fun_call>
   <var_stmt>  ::=  VAR ID ( <var_init> | <var_type> ( <var_init> | ε ) )
   <var_type>  ::=  COLON <data_type>
   <var_init>  ::=  ASSIGN <expr>
 <while_stmt>  ::=  WHILE <expr> <block>
    <if_stmt>  ::=  IF <expr> <block> ( ELSE ( <if_stmt> | <block> ) | ε )
   <for_stmt>  ::=  FOR ID FROM <expr> TO <expr> <block>
<return_stmt>  ::=  RETURN <expr>
<assign_stmt>  ::=  <lvalue> ASSIGN <expr>
     <lvalue>  ::=  ID ( LBRACKET <expr> RBRACKET | ε ) ( DOT ID ( LBRACKET <expr> RBRACKET | ε ) )*
   <fun_call>  ::=  ID LPAREN <args> RPAREN
       <args>  ::=  <expr> ( COMMA <expr> )* | ε
       <expr>  ::=  ( <rvalue> | NOT <expr> | LPAREN <expr> RPAREN ) ( <bin_op> <expr> | ε )
     <bin_op>  ::=  PLUS | MINUS | TIMES | DIVIDE | AND | OR | EQUAL | LESS | GREATER |
                    LESS_EQ | GREATER_EQ | NOT_EQUAL
     <rvalue>  ::=  <literal> | <new_rvalue> | <var_rvalue> | <fun_call>
 <new_rvalue>  ::=  NEW ID LPAREN <args> RPAREN | NEW ( ID | <base_type> ) LBRACKET <expr> RBRACKET
    <literal>  ::=  INT_VAL | DOUBLE_VAL | STRING_VAL | BOOL_VAL | NULL_VAL
 <var_rvalue>  ::=  ID ( LBRACKET <expr> RBRACKET | ε ) ( DOT ID ( LBRACKET <expr> RBRACKET | ε ) )*
```