CPSC 326: Homework 3 *Due: Tuesday, Feb. 16*

Reading Assignment. Read the following section in the textbook:

- Section 2.4: Compilers

Programming Homework. The goal of this assignment is to implement an initial Parser (recognized) for MyPL. To complete the assignment, finish the following steps. Note that you should get started on this assignment early to give yourself enough time to ask questions (and receive a response) before the due date. If you wait until the last minute and have issues, you will likely have to turn your homework in with the late penalty. If you have questions, please ask them over Piazza or else via office hours with the instructor or graders.

1. *Download the HW-3 Starter Code.* Use the link provided in Piazza to accept the GitHub Classroom assignment for HW-3. When accepting the assignment, please be sure to link your account to your name in the roster (if you haven't done so already). Accepting the assignment will create a copy of the starter code and place it in a hw-3 repository for you. You will then need to clone this repository to your local machine. As always, be sure to frequently add, commit, and push your changes.

2. *Add your finished* `token.h` *and* `lexer.h` *file to your repository.* You will need to copy your `token.h` and `lexer.h` files as well as `mypl_exception.h` from HW-2 into your HW-3 repository (and working directory). These files are required to compile and build the `hw3` executable.

3. *Define and implement the recursive descent functions in* `parser.h`. Please see the discussion and notes from class for more details. As a guide, use the basic MyPL grammar below. Note that when defining your recursive descent functions, you will generally want to create one function per non-terminal. However, there are cases where you may want to simplify or slightly modify this assumption based on the MyPL grammar given (i.e., the grammar doesn't necessarily provide a direct mapping to recursive descent functions). A set of basic test files are also provided in the `test` subdirectory. As you implement the parser functions, you will want to use and extend these files as per the "`TODO`" comments. Note that you must provide a robust set of tests for your parser, and the graders will be using additional tests as well.

4. *Testing your implementation.* In addition to the test files, you will also want to test cases that represent invalid MyPL syntax. Again, this can be done using additional test files, from the comand line, or using standard in from `hw3`. The graders will also be testing your code for error cases.

5. *Submit your code.* Be sure you have submitted all of your source code to your GitHub repo for the assignment (again, you should get into the habit of frequently adding, committing, and pushing your code). In addition to your source code, you must also submit each of

your extended and any additional test files you used for testing. For error cases, submit a file that contains the error cases you tried (or else a script with the error cases themselves). Note that *all necessary files to compile and build* your program must be checked in to your repository. *If your homework doesn't compile, the graders won't be able to test it for correctness or completeness.*

Homework Submission. All homework must be submitted through GitHub Classroom. A link for each assignment will be posted on Piazza when the homework is assigned. Be sure all of your code is pushed by the due date (which you can double check for your repository using the GitHub website). Each programming assignment is worth **35 points**. The points are allocated based on the following.

- **Correct and Complete (25 points)**. Your code must correctly and completely do the requested tasks using the requested techniques. Note that for most assignments you will be provided a *partial* set of test cases to help you determine a *minimal level* of correctness. If your program fails any of the provided test cases you will only receive partial credit. *Note that passing the given test cases does not mean your work is complete nor correct.* Your assignment will also be graded with additional test cases (not provided to you) that will help the graders determine the extent of your solution and your final score. Note that for C++ code, correctness also implies properly handling the creation and deletion of dynamic memory (i.e., the absence of memory leaks).

- **Evidence and Quality of Testing (5 points)**. As part of your homework assignments you must develop additional test cases beyond those given to you to ensure your program is correct and complete. These test cases must be turned in with your assignment. You will be graded on the scope and quality of the additional test cases you provide.

- **Formatting and Comments (5 points)**. Your code must be formatted consistently and appropriately for the language used. For C++, you must follow the provided style guide (see the course webpage). You must also comment your code and test cases, which at a minimum must include a file heading (see examples provided), function comments, and meaningfully selected variable, class, and function names.

MyPL Grammar. The following grammar rules define the MyPL syntax. See comments above regarding the grammar rules.

$\langle program \rangle$ ::= ($\langle tdecl \rangle$ | $\langle fdecl \rangle$)*

$\langle tdecl \rangle$ ::= `TYPE ID` $\langle vdecls \rangle$ `END`

$\langle vdecls \rangle$ ::= $\langle vdecl\_stmt \rangle$ $\langle vdecls \rangle$ | ϵ

$\langle fdecl \rangle$ ::= `FUN` ($\langle dtype \rangle$ | `NIL`) `ID LPAREN` $\langle params \rangle$ `RPAREN` $\langle stmts \rangle$ `END`

$\langle params \rangle$ ::= `ID COLON` $\langle dtype \rangle$ (`COMMA ID COLON` $\langle dtype \rangle$)* | ϵ

$\langle dtype \rangle$ ::= `INT_TYPE` | `DOUBLE_TYPE` | `BOOL_TYPE` | `CHAR_TYPE` | `STRING_TYPE` | `ID`

$\langle stmts \rangle$ ::= $\langle stmt \rangle$ $\langle stmts \rangle$ | ϵ

$\langle stmt \rangle$ ::= $\langle vdecl\_stmt \rangle$ | $\langle assign\_stmt \rangle$ | $\langle cond\_stmt \rangle$ | $\langle while\_stmt \rangle$ | $\langle for\_stmt \rangle$ | $\langle call\_expr \rangle$ | $\langle exit\_stmt \rangle$

$\langle vdecl\_stmt \rangle$ ::= `VAR ID` ((`COLON` $\langle dtype \rangle$) | ϵ) `ASSIGN` $\langle expr \rangle$

$\langle assign\_stmt \rangle$ ::= $\langle lvalue \rangle$ `ASSIGN` $\langle expr \rangle$

$\langle lvalue \rangle$ ::= `ID` (`DOT ID`)*

$\langle cond\_stmt \rangle$ ::= `IF` $\langle expr \rangle$ `THEN` $\langle stmts \rangle$ $\langle condt \rangle$ `END`

$\langle condt \rangle$ ::= `ELIF` $\langle expr \rangle$ `THEN` $\langle stmts \rangle$ $\langle condt \rangle$ | `ELSE` $\langle stmts \rangle$ | ϵ

$\langle while\_stmt \rangle$ ::= `WHILE` $\langle expr \rangle$ `DO` $\langle stmts \rangle$ `END`

$\langle for\_stmt \rangle$ ::= `FOR ID ASSIGN` $\langle expr \rangle$ `TO` $\langle expr \rangle$ `DO` $\langle stmts \rangle$ `END`

$\langle call\_expr \rangle$ ::= `ID LPAREN` $\langle args \rangle$ `RPAREN`

$\langle args \rangle$ ::= $\langle expr \rangle$ (`COMMA` $\langle expr \rangle$)* | ϵ

$\langle exit\_stmt \rangle$ ::= `RETURN` $\langle expr \rangle$

$\langle expr \rangle$ ::= ($\langle rvalue \rangle$ | `NOT` $\langle expr \rangle$ | `LPAREN` $\langle expr \rangle$ `RPAREN`) ($\langle operator \rangle$ $\langle expr \rangle$ | ϵ)

$\langle operator \rangle$ ::= `PLUS` | `MINUS` | `DIVIDE` | `MULTIPLY` | `MODULO` | `AND` | `OR` | `EQUAL` | `LESS` | `GREATER` | `LESS_EQUAL` | `GREATER_EQUAL` | `NOT_EQUAL`

$\langle rvalue \rangle$ ::= $\langle pval \rangle$ | `NIL` | `NEW ID` | $\langle idrval \rangle$ | $\langle call\_expr \rangle$ | `NEG` $\langle expr \rangle$

$\langle pval \rangle$ ::= `INT_VAL` | `DOUBLE_VAL` | `BOOL_VAL` | `CHAR_VAL` | `STRING_VAL`

$\langle idrval \rangle$ ::= `ID` (`DOT ID`)*