CPSC 326: Homework 4 *Due: Thursday, Feb. 25*

Reading Assignment. Read the following sections in the textbook:

- Chapter 4: Names and the Environment

Programming Homework. The goal of this assignment is to modify your Parser implementation for HW-3 to generate an abstract syntax tree (AST). 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-4 Starter Code.* Use the link provided in Piazza to accept the GitHub Classroom assignment for HW-4. 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-4 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 previous files HW to your repository.* You will need to copy your header and cpp files from HW-3 into your HW-4 repository (and working directory). These files are required to compile and build the `hw4` executable.

3. *Extend your Parser (in* `parser.h`*) to build the corresponding AST objects (defined in* `ast.h`*).* Please see the discussion and notes from class for more details. As a guide, use the basic MyPL AST diagram below. Note that you may need to modify your recursive descent functions from HW-3 as you instrument your code to build the AST objects. A set of basic test files are also provided in the `test` subdirectory. As you work on your implementation, you will want to use these files as well as develop additional tests to check your work. Note that you must also provide a robust set of tests for the assignment, and the graders will be using additional tests as well.

4. *Implement the* `Printer` *AST visitor to "pretty print" the original source code based on the AST generated by the Parser.* When writing your "pretty printer", you must use the following `MyPL` code styling rules (also see the test cases provided separately).

   (a) Indent all statements within a block by three spaces.

   (b) Each statement should be on a separate line without blank lines before or after the statement. The exceptions to this rule are user-defined type and function declarations.

   (c) Format variable declarations without explicit types using one space between `var` and the identifier, and one space before and after the assignment symbol, e.g., `var` *id* = *expr*.

(d) Format variable declarations with explicit types the same as for implicit types, but with a colon directly after the variable name (no extra space), followed by a space, followed by the type, followed by a space, e.g., `var` *id: type* = *expr*.

(e) Format variable assignments with one space between the identifier and the assignment symbol, e.g., *id* = *expr*.

(f) Format type declarations such that the reserved word `type` and the type name appears on one line, with one space between the two, each variable declaration indented (two spaces from the start of `type`) and on a separate line (with no blanks between), and `end` on the next immediate line after the last variable declaration and aligned with `type`. There should be one blank line after a type declaration. Here is an example:

```
type Person
  var age = 0
  var name = ""
  var mother: Person = nil
  var father: Person = nil
end
```

(g) Format function declarations such that `fun`, the return type, the function name, and the parameter list are all on the same line, the body of the function is indendented appropriately, and the `end` is on a separate line, immediately following the last body statement, aligned with the `fun` reserved word. There should be one space between `fun` and the type, one space between the type and the function name, and so on. There should be one blank line after each function declaration. Here is an example:

```
fun int add(x: int, y: int)
  sum = x + y
  return sum
end
```

(h) Format while statements such that `while` and `do` occur on the same line, there is one space between the start and end of the Boolean expression, the body of the while loop is appropriately indendented, and `end` is aligned with `while` and occurs on the line immediately after the last statement of the body.

(i) Format for statements such that `for` and `do` occur on the same line, each part of the loop specification is separated by a single space, the body of the for loop is appropriately indendented, and `end` is aligned with `for` and occurs on the line immediately after the last statement of the body.

(j) 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), `then` appears on the same line as its corresponding `if` or `elseif`, and `end` statements appear on separate lines with no preceding blank lines.

(k) Format simple expressions without any extra spaces. Path expressions should not contain spaces between corresponding dots (e.g., `x.y.z`).

(l) Format complex expressions with spaces between their corresponding parts and fully enclose them in parentheses (regardless of whether there were parentheses in the original source code). For example, if the original was written as `3+4+5` the pretty-printed version should be written as `(3 + (4 + 5))`.

(m) Boolean expressions should follow the same rules as for complex expressions. For example, `not (x>1) and (y>1)` should print as `not ((x > 1) and (y > 1))`.

(n) Format structured type object creation such that there is one space between `new` and the struct type name.

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

Note that because we are not going to be implementing proper associativity and precedence of operators, some `MyPL` expressions can end up with some strange looking parenthesizations. Thus, associativity and precendence will need to be explicitly stated by a programmer via explicit use of parentheses.

5. *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 `hw4`. The graders will also be testing your code for error cases.

6. *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.*

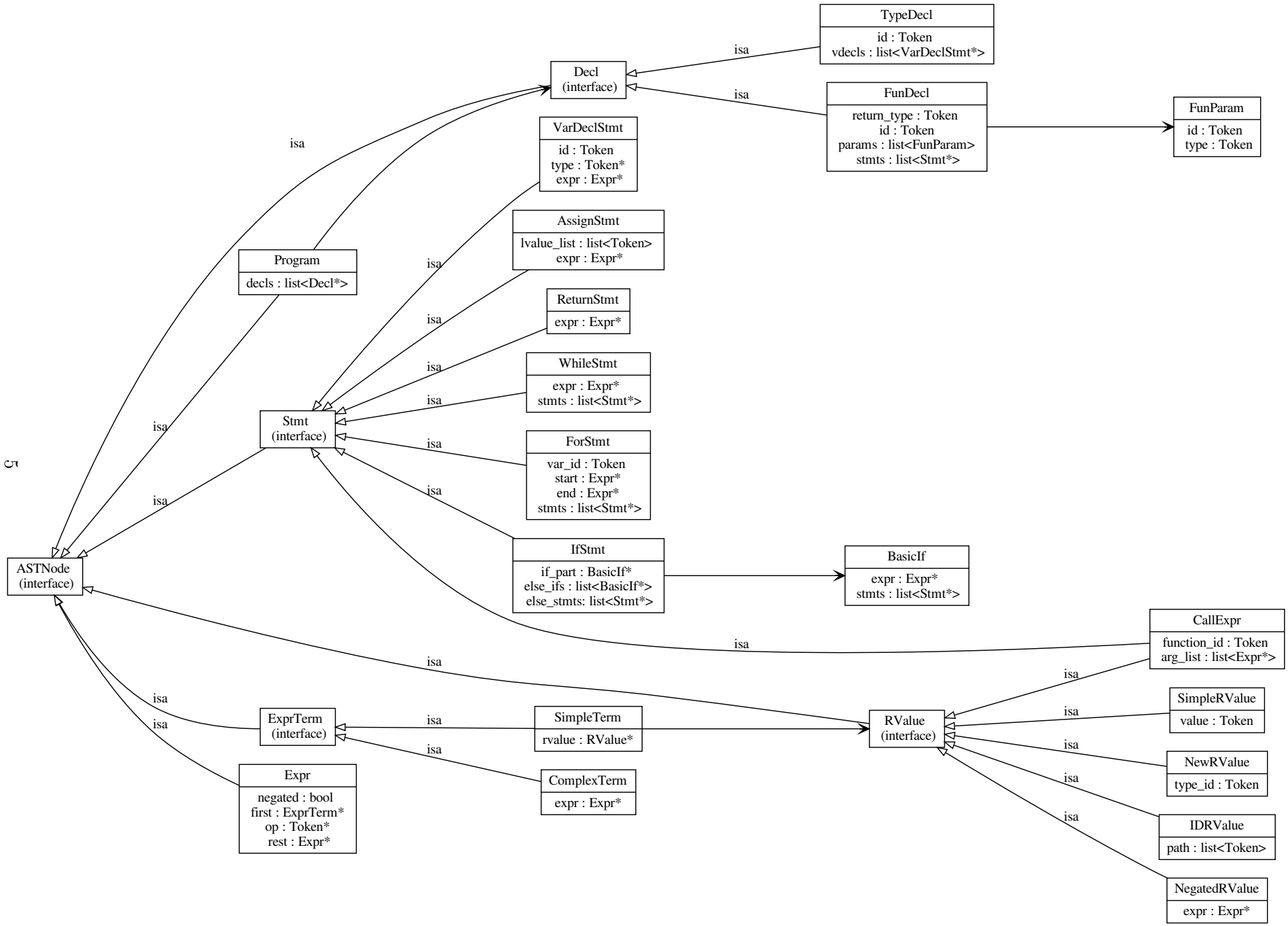
Homework Submission. All homework must be submitted through GitHub Classroom. A link for each assignment will be posted on Piazza when the homework is assigned. Be sure all of your code is pushed by the due date (which you can double check for your repository using the GitHub website). Each programming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


ASTNode
(interface)
Program
decls : list<Decl*>
Decl
(interface)
TypeDecl
id : Token
vdecls : list<VarDeclStmt*>
FunDecl
return_type : Token
id : Token
params : list<FunParam>
stmts : list<Stmt*>
FunParam
id : Token
type : Token
VarDeclStmt
id : Token
type : Token*
expr : Expr*
Stmt
(interface)
AssignStmt
lvalue_list : list<Token>
expr : Expr*
ReturnStmt
expr : Expr*
WhileStmt
expr : Expr*
stmts : list<Stmt*>
ForStmt
var_id : Token
start : Expr*
end : Expr*
stmts : list<Stmt*>
IfStmt
if_part : BasicIf*
else_ifs : list<BasicIf*>
else_stmts: list<Stmt*>
BasicIf
expr : Expr*
stmts : list<Stmt*>
ExprTerm
(interface)
Expr
negated : bool
first : ExprTerm*
op : Token*
rest : Expr*
SimpleTerm
rvalue : RValue*
ComplexTerm
expr : Expr*
RValue
(interface)
CallExpr
function_id : Token
arg_list : list<Expr*>
SimpleRValue
value : Token
NewRValue
type_id : Token
IDRValue
path : list<Token>
NegatedRValue
expr : Expr*
isa