CSE 6341, Programming Project 1
Due Thursday, September 12, 11:59 pm (10 points)

The goal of this project is to implement a type checker for a very simple language. The checking is based on material discussed in class but with a few modifications to the underlying context-free grammar. Your implementation will use the supplied code for parsing and AST building (AST = abstract syntax tree; this is a more compact version of the parse tree). Before starting, make sure your Java version and environment variables from Project 0 are set up correctly.

Step 1: Set up
Say you have created `/home/buckeye.8/6341` and have Project 0 in it. Run `./plan t1` again.

Step 2: Understand the AST
The context-free grammar for the targeted language is as follows:
<program> ::= <unitList>
<unitList> ::= <unit><unitList> | <unit>
<unit> ::= <decl> | <stmt>
<decl> ::= <varDecl> **;** | <varDecl> **=** <expr> **;**
<varDecl> ::= **int ident** | **float ident**
<stmt> ::= **ident =** <expr> **;** | **print** <expr> **;**
<expr> ::= **intconst** | **floatconst** | **ident** | <expr> **+** <expr> | **(** <expr> **)**

Terminal symbols are shown in blue. For example, **intconst** is a single terminal symbol representing an integer constant. If you want to see the details of how the terminal symbols are defined, see `parser/Scanner.jflex` (the definition of regular expression Ident and everything after that). If you want to see the details of how the parser is defined, see `parser/Parser.cup` (the definition of terminals INT, etc. and everything after that). You do **not** need to understand these details to complete the project successfully. Do **not** change the parser or the scanner.

Read the code in `p1/ast` to see how the AST nodes are defined. The root of the AST is a node that is an instance of class `Program`. Each type of AST node corresponds to some Java class. For example, class `AssignStmt` implements a <stmt> node for the following production:

<stmt> ::= **ident =** <expr> **;**

Field `ident` in class `AssignStmt` corresponds to terminal **ident** in the production, while field `expr` corresponds to non-terminal <expr>. Also see `p1/interpreter` for the entry point of the project. Calling `toString` on any AST node will give you a string representation of the subtree rooted at that node; printing these strings may help you understand the AST structure.

It is essential to do this reading early and to ask any clarification questions as soon as possible. If you do not have experience with object-oriented programming in Java, please proactively reach out to me for clarifications when necessary.

Step 3: Implement type checking
You need to implement a type checker to check for the following conditions:

1) Any variable appearing in an <expr> must have a declaration in some earlier <decl>. For example, **int x = 1; int y = x + w;** is not allowed because **w** is not declared. As another example, **int x = x+1;** is also not allowed (the **x** in **x+1** is not declared).

2) Each variable can be declared only once. For example, **int x; int y = 1; int x = y+1;** is not allowed.

3) In an assignment **ident =** <expr> **;** the variable on the left-hand side of the assignment must be already declared. For example, **int x; y = x+1;** is not allowed.

4) In an assignment **ident =** <expr> **;** or a declaration with initialization <varDecl> **=** <expr> **;** the type of the variable on the left-hand side of **=** must be the same as the type of the expression on the right-hand side. For example, **int x; float y = 1.; x = 3.14 + y;** is not allowed; neither is **int x = 3.14;**

5) Both operands of **+** must be of the same type. For example, **int x = 1; float y = 3.14; float z = y + 1.1; int w = x + z;** is not allowed because of **x + z**.

If the program violates any of these checks, call *Interpreter.fatalError* with exit code *Interpreter.EXIT_STATIC_CHECKING_ERROR*. The test script will check this exit code, so make sure your implementation uses it. The text message associated with the error should be something simple that describes which specific check was violated. Your code should call *Interpreter.fatalError* as soon as it detects a violation. If the program contains several typechecking errors, only the earliest one will be detected and reported.

Additional details:

1) Do **not** change the constructors of any AST node classes. These constructors are used by the parser while creating the AST. Your type checking should be performed **after** the AST has been completely created by the parser. The starting point of your type checking is shown by a TODO comment in *Interpreter.java*.

2) A natural way to implement the checking is to add a new method *check* to the classes in package *ast*. This method will perform checking for the corresponding AST node, which would typically involve calling *check* on the children nodes. For example, *check* for *PlusExpr* would call *check* on its two children (accessible through fields *expr1* and *expr2*). In a typical object-oriented style, you would declare a new abstract method *check* in class *ASTNode* (this class is the superclass of all classes in *ast*) and then implement methods *check* in the relevant classes.

3) If you are not familiar with inheritance and abstract methods in object-oriented languages, examine how method *print* is defined and implemented for all classes in package *ast*.

4) You will need to store information about the types of identifiers that have been declared so far. A simple solution is to have a single map (e.g., a `java.util.Map`) that maps identifiers to types. This is the "single global table" from the slides. One very easy approach is to make the map directly accessible everywhere, by using a public static field in class `Program`. This field will be initialized with an empty map at the very beginning of method `Program.check`.

5) To deal with checking violations, you can simply call `Interpreter.fatalError` directly from the location where the violation was first detected.

6) To implement attribute *type* for AST nodes for expressions, a simple approach could be to add new fields `type` to relevant classes from package `ast` to store the values of this attribute.

Step 4: Testing

Write many test cases and test your checker with them. Submit at least 5 test cases with your submission. The test cases you submit will not affect your score for the project. Put them in the same location as the provided file t1 and name them s1, s2, …

Step 5: Submission

After completing your project, run

```
cd p1; make clean; cd ..
tar -cvzf p1.tar.gz p1
```

Then submit `p1.tar.gz` in Carmen.

General rules (copied from the course syllabus)

Submissions must be uploaded via Carmen by 11:59 pm on the due date. The projects must compile and run on **coelinux**. Some students prefer to implement the projects on a different machine. If you decide to do this, it is your responsibility to make the code compile and run correctly on coelinux before the deadline. Many students have tried to port to coelinux too close to the deadline, leading to last-minute problems and missed deadlines.

Projects should be done independently. General high-level discussion of projects with other students in the class is allowed, but **you must do all design, programming, testing, and debugging independently.** Projects that show excessive similarities will be taken as evidence of cheating and dealt with accordingly. Code plagiarism tools may be used to detect cheating. See the syllabus under "Academic Integrity".

You can submit up to 24 hours after the deadline; if you do so, your score will be reduced by 10%. **ONLY THE LAST SUBMITTED VERSION WILL BE CONSIDERED. Triple-check carefully that you have submitted the correct version. If you submit the wrong version of your code, and you get a low score (or zero score), I will NOT consider resubmissions – the original low/zero score will be assigned WITHOUT DISCUSSION.**

If you submit more than 24 hours after the deadline, the submission will not be accepted. NO EXCEPTIONS TO THIS RULE WILL BE CONSIDERED. NO REQUESTS FOR RESUBMISSION WILL BE CONSIDERED. MAKE SURE YOU SUBMIT THE CORRECT CODE VERSION.

Read the project description **very carefully, several times, start-to-end.** If you need any clarifications, contact me immediately (do **not** wait until the last minute). **Test extensively.**

Accommodations for sickness and other special circumstances will be made based on university guidelines. Please contact me **ahead of time** to arrange for such accommodations.