Assignment Description

In this programming assignment, you will implement the syntax and semantic analysis phases for IC, including the AST construction and the symbol tables. The latest version of the IC language specification can be found on the course web site. We expect you to build upon the code that you wrote for Programming Assignment 1. You are required to implement the following:

- **The parser**. To generate the parser, you will use Java CUP, an LALR(1) automatic parser generator for Java, available at: `http://www.cs.princeton.edu/~appel/modern/java/CUP`. You should use Java CUP only to build the AST and the symbol tables. The compiler will perform the semantic checks in a separate phase, after the program has been parsed and the AST has been constructed.

  You will use the grammar from the IC language specification as a starting point. You must modify this grammar to make it LALR(1) and get no conflicts when you run it through Java CUP. The operators must satisfy the precedence given in the IC specification and all binary operators must be left-associative. You are allowed to use Java CUP precedence and associativity declarations.

  For details about the integration of your parser with the lexer generated in the previous assignment, read Section 2.2.8 (Java CUP Compatibility) of the JLex documentation, and Section 5 (Scanner Interface) of the Java CUP documentation. Also, replace the `sym.java` file in the lexer module with the `sym.java` automatically generated by Java CUP.

- **AST construction**. Design a class hierarchy for the abstract syntax tree (AST) nodes for the IC language. When the input program is syntactically correct, your checker will produce a corresponding AST for the program. The abstract syntax tree is the interface between the syntax and semantic analysis, so designing it carefully is important for the subsequent stages in the compiler. Note that your AST classes do not necessarily correspond to the non-terminals of the IC grammar. Use the grammar from the language specification only as a guideline for designing the AST. Once you designed the AST class hierarchy, extend your parser such that it also constructs the AST.

- **Symbol Tables and Types**. You then have to design a hierarchy of symbol tables and a hierarchy of types. Your design should allow each AST node to access the symbol table corresponding to its current scope (e.g. class, method, or block scope), and each entry in the symbol table should have information about the type of the identifier stored in that entry. Any errors which occur during symbol table construction (such as multiply declared identifiers) are considered semantic errors.

  Your constructed symbol tables should be available to all remaining phases of the compiler. In the rest of your compiler, you will refer to program symbols (e.g., variables, methods, etc) using references to their symbol table entries.

- **Semantic checks**. After you have constructed the AST and the symbol tables, your compiler will analyze the program and perform semantic checks. These semantic checks include type-checking, scope rules, and all of the other checks presented in the IC specification.

- **Error Handling**. You must extend your error package with `SyntaxError` and `SemanticError` exceptions, and have your compiler throw such exceptions whenever it encounters errors. These exceptions must carry information about the error, such as the token and the line number where the syntactic error has occurred, or a message describing the violated semantic rule. It is not required to report more than one error; the execution may terminate after the first lexical, syntactic, or semantic error.

Command line invocation. Your compiler will be invoked with a single file name as argument, as in the previous assignment. The compiler will parse the input file, construct the AST and symbol tables, will perform the semantic checks, and will report any error it encounters. In addition, your compiler must support two command-line options to print internal information about the AST and the symbol tables:

1. The "`-print-ast`" option: will print at `System.out` a textual description of the constructed AST;
2. The "`-print-symtab`" option: will print a textual description of the symbol tables.

You may find it helpful to use the graph visualization tool `graphviz` for printing out information about the AST and the hierarchy of symbol tables. You can find a web link to this tool on the course web site. As part of that package, you will find the `dot` program, which reads a textual specification for a graph and outputs a PostScript document. For instance, the dot specification for the AST of the statement `x = y + 1` is:

```
digraph G {
  Assign -> {"Id x", Plus}
  Plus -> {"Id y", "Num 1"}
}
```

However, it is not part of the requirement to use such a description. You can use your own textual description of the AST and symbol table structures. In that case, make sure your output provides enough information and is easy to read.

Package Structure: You will implement the new components of the compiler as sub-packages of the IC package. You will have a sub-package for each of the following: 1) the parser module, 2) the AST class hierarchy, 3) the symbol tables, 4) the representation of types, and 5) the semantic checks. The syntax and semantic exceptions should be part of the error package that you defined in the previous assignment.

What to turn in

Turn in your code electronically using the Course Management System (CMS) on the due date, and submit your a short write-up the next day in class. You must submit your source code as `pa2.tar.gz` using CMS, anytime on the due date (i.e. until 11:59pm). Please include only the source files and your test cases in your submission. For this assignment, we also expect a checkpoint submission halfway through the assignment – see details below.

Turn in on paper:

- A brief, clear, and concise document describing the your code structure and testing strategy. Include in this document: 1) a description of your AST and symbol table hierarchies, and 2) a list of all the semantic checks that your compiler performs, except type-checking.
- Feedback. As in the first assignment, please provide one paragraph to tell us your overall thoughts: how much time you spent on it, what was the most difficult or more interesting part, and how you think it could be made better.

Turn in electronically:

- All of your source code and test cases (in directories `/src` and `/test`).
- A Makefile for compiling your sources and generating javadoc documentation.

Checkpoint submission:

This assignment requires significantly more work than the first. To encourage you to start working on the project early, there will be a checkpoint submission halfway through the assignment. You have to submit the current state of your work as a file `checkpt1.tar.gz` by February 21, using CMS.

If you get your final assignment working in the end, we will disregard the checkpoint submission. However, if you are not able to successfully complete the assignment, then the checkpoint submission will have an impact on your grade: if we determine that little work has been done by the checkpoint, and you left most of the work for the last minute, then you will be severely penalized.