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. You can construct the symbol tables either during the parsing phase, or separately, in a subsequent phase.

  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 must be invoked with a single file name as argument:

```
java IC.Compiler <file.ic>
```

With this command, 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. Your compiler must also support two command-line options to dump internal information about the AST and the symbol tables:

1. The "`-dump-ast`" option: the compiler will print at `System.out` a textual description of the AST for the input file. Each AST node will be printed on a separate line, and must have information about what the children nodes are (for instance, using numerical identifiers for the nodes).
2. The "`-dump-symtab`" option: the compiler will print a textual description of the symbol tables. Each entry in the symbol table will be printed on a separate line; the information for each node should include the name of the identifier, its kind (variable, method, etc), and its type. Different symbol tables will be separated by blank lines. You must indicate, for each symbol table, what are its children tables.

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.

Makefile: You must also provide a makefile at the top level of your hierarchy and support the following commands: "`make compiler`" will build your compiler from the source code, and "`make clean`" will remove all the class files and will leave only the source code.

What to turn in

As in any other large program, much of the value in a compiler is in how easily it can be maintained. For this reason, a high value is placed here on both clarity and brevity – both in documentation and code.

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 about the assignment: 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.

Electronic submission:

Your electronic submission is expected at the same time as your written submission: at the beginning of class on the due date. Electronic submissions after 10AM will be considered a day late. Place your files in `\\crane\courses\cs412-sp03\grpX\pa2`, where `grpX` is your group identifier. Please organize your top-level directory structure as follows :

- `src` : all of your source code. Do not submit the class files – we should be able to build your compiler from its source code using `make compiler`.
- `doc` : javadoc output that we can browse on-line. We should be able to browse through all your classes and interfaces, including all their methods and fields. Also include in this directory your write-up (`writeup.ps` or `writeup.txt`) and a `readme.txt` file containing any known bugs and other information that we might find useful when grading your assignment.
- `test` : any test cases you used in testing your project.

Note: Failure to submit your assignment in the proper format may result in deductions from your grade.