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. A link to Java CUP is available on the course web site. While parsing, your compiler will build the AST and the symbol tables.

  You will use the grammar from the IC language specification as a starting point for you CUP parser specification. You must modify this grammar to make it LALR(1) and get no conflicts when you run it through Java CUP. The operator precedence and associativity must be as indicated in the IC specification. 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 JFlex documentation, and Section 5 (Scanner Interface) of the Java CUP documentation. You must replace the `sym.java` file in the lexer module with the `sym.java` automatically generated by Java CUP. Also, you must either replace the `Token` class with `java_cup.runtime.Symbol`, or make `Token` a subclass of `java_cup.runtime.Symbol`.

  In addition to parsing the program file, you must also read and parse the library signature file `libic.sig`. The syntax of this file is much simpler. We recommend that you get started by writing this simpler parser first.

- **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**. Then design the symbol table structures and the hierarchy of program 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.

Your constructed symbol tables should be available to all remaining phases of the compiler. We recommend that all subsequent compilation phases refer to program symbols (e.g., variables, methods, etc) using references to their symbol table entries, not using their names.

- **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 requirements described in the language specification.

- **Error Handling**. Whenever syntax or semantic errors are encountered, the program must terminate immediately, and print an succinct, but informative message describing the problem. Syntax errors must clearly indicate the line number and token where they occur. Type and semantic errors must indicate at least the class and method where they occur. One should be able to fix the problem immediately after reading the error message.

Command line invocation. Your compiler will be invoked with the program file name as an argument. Optionally, one can also specify the location of the library signature file `libic.sig`:

```
java IC.Compiler <file.ic> [ -L </path/to/libic.sig> ]
```

The compiler will parse the input file and the signature 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 graphical image (in PostScript format, jpg, or other image formats). 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; and 4) the representation of types.

1 What to turn in

Turn in your code electronically using the Course Management System (CMS) on the due date. You must submit your source code as `pa2.zip` using CMS, before 11pm on the due date. 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 electronically:

- All of your source code and test cases (in directories `/src` and `/test`). As in the previous assignment, make sure your code is well-documented. We will generate javadoc documentation and browse through your comments.
- A brief, clear, and concise document describing the your code structure and testing strategy. Place this document in `/writeup`. Include in this document a description of your AST and symbol table hierarchies, and a list of all the semantic checks that your compiler performs (other than type-checking). Also describe how you broke up the assignment between group members, and describe the interfaces that you have used between the smaller pieces that different members have been working on.

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 `checkpt.zip` by February 17, 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 be an indicator for how early you started working on the assignment: if we determine that you left most of the work for the last minute, then you will be penalized more severely.