PA 2: IC Syntax and Semantic Analysis

due: Friday, Oct. 28

CSCI 434T
Fall, 2011

Overview

In this programming assignment, you will implement the syntax and semantic analysis phases for IC. These phases require you to write the parser, the AST and the symbol table packages, and the type checker.

Details

You are required to implement the following:

- **The Parser**. To generate the parser, you will use Java CUP, a LALR(1) automatic parser generator for Java. (LALR(1) parsers are essentially LR(1) parsers, except that they typically have a much smaller automaton because they merge states containing the same items when possible.) A link to Java CUP is available on the course web site.

  You will use the grammar from the IC Language Specification as a starting point for your 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 (and should!) use Java CUP precedence and associativity declarations.

  You should use the parser only to build the AST. Separate passes over the AST will build the symbol tables and perform the semantic checks, after the program has been parsed and the AST has been constructed.

- **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 have designed the AST class hierarchy, extend your parser to construct ASTs.

  Your AST nodes should implement the Visitor pattern, as explored in the homework. Both the standard visitor and the propagating visitor will be very useful.

- **Symbol Tables and Types**. Then design the symbol table structures and any additional structures you wish to use for representing 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).

  Your constructed symbol tables should be available to all remaining compilation phases, and I recommend that all subsequent phases refer to program symbols (e.g., variables, methods, class names, etc.) using references to their symbol table entries, and not using their string names. In other words, each AST node containing a name should be given a second field that you fill in during symbol resolution. This second field should contain enough information to uniquely identify the declaration, type, etc. of the symbol being referenced. There are many ways to accomplish this. Perhaps the most straightforward is to have the symbol table structure map each string name to the AST node corresponding to the declaration of that name. Given that structure, you can simply augment each AST node containing a name with a pointer to the AST node for that name's declaration, and fill in this pointer during name resolution. (This is briefly described, along with alternatives, in Cooper and Torczon, Chapter 5.7.)

- **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**. 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 line number and a message describing the violation. You are not required to report more than one error; the execution may terminate after the first lexical, syntactic, or semantic error. One should be able to fix the problem immediately after reading the error message.

Command Line Invocation. As in PA 1, your compiler will be invoked with the program file name as an argument, with an optional "`-d`" flag:

```
java -classpath .:tools/java-cup-11a.jar ic.Compiler <file.ic>
```

You must include the `java-cup-11a.jar` file in the clas path. This JAR file contains definitions used by the CUP-generated parser.

The compiler will parse the input file, construct the AST and symbol tables, perform the semantic checks, and 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 "`-printAST`" option: prints a textual description of the constructed AST to `System.out`.
2. The "`-printSymTab`" option: prints a textual description of the symbol tables to `System.out`.

These options should appear after the filename, as in:

```
java -classpath .:tools/java-cup-11a.jar .ic.Compiler <file.ic> -printAST
```

You can design your own textual description of the AST and symbol table structures, but make sure your output provides all important information and is easy to read.

Package Structure. You should implement the new components of the compiler as the following sub-packages of the `ic` package:

- the `error` module for error exceptions;
- the `lex` module for the `ic.lex` specification and associated classes;
- the `parser` module for the `ic.cup` specification and associated classes;
- the `ast` module for the AST class hierarchy;
- the `symtab` module for symbol tables; and
- the `tc` module for the type checker.

The `dot` Utility. You may find it helpful to use the graph visualization tools in the `graphviz` suite of tools for printing out information about the AST and the hierarchy of symbol tables. You can find information about this tool on the course web site. The most useful tool would be the `dot` program, which reads a textual specification for a graph and outputs a graphical image (in PDF format, jpg, or other image formats). For instance, the dot specification for the AST of the statement `x = y + 1` is:

```
digraph G {
  expr [label="="];
  lhs [label="x"];
  rhs [label="+"];
  leftop [label="y"];
  rightop [label="1"];
  expr -> lhs;
  expr -> rhs;
  rhs -> leftop;
  rhs -> rightop;
}
```

Running the `dot` tool on a file containing this description, as described in HW 4, will produce a graphical tree. However, using dot is encouraged but not required.

Getting Started

For details on how to integrate your parser with the PA 1 lexer, you may wish to read Section 2.2.8 (Java CUP Compatibility) of the JFlex documentation, and Section 5 (Scanner Interface) of the Java CUP documentation. In essence, 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`.

To simplify these steps, I have provided a PA2 Starter project on the website. Even if you choose to use your own lexer code, you may wish to begin with this project because it contains additional configuration details to enable it to generate CUP parsers. Simply copy your source in `ic.flex` once you have it set up.

To set up the project, have one person from the group download and import it into Eclipse. *Be sure to select "*`Copy projects into Workspace`*" in the Import Dialog Box.* Once imported, right-click on the project name, select "`Team -> Share Project...`", and add it to your SVN repository. After you perform a commit, others will be able to check that file out through the SVN Repository Explorer. (Recall that you get to that perspective by choosing "`Open Perspective`" from the menu and selection "`Other...`" and then "`SVN Repository Explorer`."

While debugging you parser, you may find it useful to run CUP in a mode that dumps the automaton and parse table. To do so, run the following from the command line:

```
java -jar tools/java-cup-11a.jar -destdir ic/parser -dump ic/parser/ic.cup
```

(Running "`make dump`" in the PA2-Starter directory will do the same.)

Using Scala. If you wish to use Scala:

- Download the Scala version of the starter project.
- You can run your IC compiler on the command line as follows:

  ```
  scala -classpath .:tools/java-cup-11a.jar ic.Compiler <file.ic>
  ```

  The "-d" flag works as above.
- You may use `case` classes instead of the visitor pattern.
- As in PA 1, if Eclipse gives you spurious error messages, run "make" from the command line and then "Clean..." and "Refresh" the Eclipse project.

Submission

This is a *substantial* programming project. You have roughly 4 weeks to complete it — use your time wisely. There will be intermediate checkpoints along the way. Please be sure that your SVN repository contains up-to-date versions of the following by the submission deadlines listed below:

- All of your source code and test cases (in directories `/ic` and `/test`). As in the previous assignment, make sure your code is well-documented. I will browse through your comments and code a *each checkpoint*.
- A brief, clear, and concise design document. Place this document in `/writeup`. For the checkpoint submissions, this document should contain a description of the overall design and major data structures, a summary of your testing strategy, a list of known bugs or issues remaining in the code, and so on. Simply continue to add to the document each week, so that by the final submission deadline, it is a complete design document.

Schedule

These milestones are the minimal requirements for the checkpoints. You are of course welcome (if not strongly encouraged) to do more by each deadline. The Mt. Day deadline will be pushed back until Sunday evening.

Friday, Oct. 7: Your parser must successfully parse valid IC programs and report syntax errors in bad ones. Your writeup must include an initial design of your AST package, including: 1) The list of operations present in your root node class, and 2) a brief overview of the class hierarchy of node types. This need not be very detailed, but it should at least demonstrate that you have begun to think about how to lay out your ASTs.

Friday, Oct. 14: Your parser must generate ASTs for programs, and you must support the `-printAST` command-line option.

Friday, Oct. 21: All of the above, plus your compiler must generate the symbol tables and perform name resolution. At this point your compiler should implement the `-printSymTab` option. Additionally, you may wish to extend your AST printer to print out some indication of how each name in the program has been resolved.

Friday, Oct. 30: PA 2 due.