CS 301 Spring 2016
Project Phase 2
February 16 – March 17

IC **Compiler Front End**

Plan

In this programming assignment, you will implement the syntax and semantic analysis phases for IC, completing the front end of your IC compiler. You will write the parser, the AST and symbol table packages, and the type checker. This is the longest project phrase for the semester, due shortly before Spring Break. There are weekly checkpoints (listed at the end of this document) to ensure continued forward progress.

Directions for acquiring starter code have been updated after printing. Changes appear in red.

Implementation Details

You are required to implement the following:

- **Parser**. To generate the parser, use Java CUP, an LALR(1) automatic parser generator for Java. `http://www2.cs.tum.edu/projects/cup/` (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.)

  Use the grammar from the IC Language Specification as a starting point for your CUP parser specification. Modify this grammar to make it LALR(1) and free of conflicts when you run it through Java CUP. The operator precedence and associativity must be as indicated in the IC specification. You are allowed and encouraged to use Java CUP precedence and associativity declarations.

  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 rough guideline for designing the AST. Once you have designed the AST class hierarchy, extend your parser to construct ASTs.

  Your AST nodes should be implemented as case classes.

- **Symbol Tables and Types**. 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. 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 when you resolve that name. 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 in scope 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 later on. For example, if you have a `FieldDecl` class to represent field declarations and a `FieldAccess` class to represent a field access, then `FieldAccess` should have an instance variable `decl` of type `FieldDecl` that will will be linked to the appropriate field declaration during type checking. This is described briefly, along with alternatives, in EC 5.7.

- **Semantic Checks**. After constructing the AST and the symbol tables, your compiler will analyze the program and perform semantic checks. These semantic checks include type-checking, scoping rules not enforced while building the symbol tables, and all of the other requirements described in the language specification.

- **Error Handling**. 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. Your compiler will be invoked with the program file name as an argument, with an optional "`-d`" flag:

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

You must include the `java-cup-11a.jar` file in the class path. This JAR file contains definitions used by the CUP-generated parser. The following script will simplify running your code so you don't need to type this in every time:

```
#!/bin/bash
scala -classpath bin:tools/java-cup-11a.jar ic.Compiler $*
```

I have included it as `icc` in the starter folder, so you can simply run "`./icc <file.ic>`"

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:

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

or

```
./icc <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.

Output Format. As in the previous assignment, the last line of the output must be either

```
Success.
```

or

```
Failed.
```

depending on whether any errors were found. Some of the intermediate checkpoints will require other data to be printed, but the final submission should print no information other than error messages and that one last line.

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;
}
```

After generating such a file, in `foo.dot`, run `dot -Tpdf < foo.dot > foo.pdf` to create a PDF graph or tree. *If using the wx appliance, run* `sudo dnf upgrade wx-cs301` *once to ensure you have* `graphviz` *and* `dot` *installed.* Using dot is encouraged but not required.

Getting Started

I provide code to help you connect JFlex and Java CUP and configure Eclipse/Scala IDE for automatically generating the parser based on your specification. Contact me if you are done with Phase 1 early and want access to this code.

Acquire starter code. These steps will acquire starter code and move your Phase 1 work aside and replace it with mine. `Compiler.scala` and the `ic.lex` package are moved into the `old` directory and replaced by my versions. You may use my lexer (no modifications required) or make small modifications to yours to integrate it with the parser. (See below for instructions to link your lexer with the parser.)

One member of the team should do the following **after submitting Phase 1**.

1. Close Eclipse/Scala IDE.
2. Make sure you have the latest version of your team's code committed in your repository working copy. Commit any local changes. Pull, and update or merge-and-commit any new revisions from your team. Make sure `hg status` shows no uncommitted changes.
3. `hg rm bin` and, unless Mercurial said it did not remove anything, `hg commit`.
4. `hg pull starters` to pull new starter code.

5. `hg merge 82ae7fed05fc` (There should be no conflicts. Consult with me if there are conflicts.)
6. `hg commit -m "move phase 1 aside"`
7. `hg merge` (There should be no conflicts. Consult with me if there are conflicts.)
8. `hg commit -m "merge phase 2 starter"`
9. Run `make`.
10. Launch Scala IDE.
11. Refresh the `icc` project.
12. Choose Project > Clean... and clean the `icc` project.

Other teammates. Once the one updater teammate has finished the steps above, they can push at any later time to share with the rest of the time. The first time any teammate gets the updated version, they must do steps 9–12.

Link your lexer and parser. These instructions indicate how to modify your lexer to work with the parser. If you wish to use my lexer, you can probably ignore this. For details on how to integrate your parser with your 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. If you wish to use your lexer, here are the basic steps required.

1. Add the following line in the top section of your `ic.flex` file:

   ```
   import ic.parser.sym;
   ```

   This will link the lexer module with the `sym.java` automatically generated by Java CUP. If you wish to remove your old handwritten `sym.java`, run: `hg forget src/ic/lex/sym.java` or do the equivalent in the GUI.
2. Update your `sym` references in `ic.flex` to match the terminal declarations in `src/ic/parser/ic.cup` or vice versa.
3. Make `Token` a subclass of `java_cup.runtime.Symbol`.

   ```
   class Token(id : Int, value : Object, line : Int)
   extends Symbol(id,line,line,value)
   ```

   Alternatively, replace your uses of `Token` with uses of `java_cup.runtime.Symbol`.
4. Make sure your lexer always handles the minus sign ("-") separately from inteteger literals.

   In other words, allow only positive integer literal tokens. The minus sign can be interpreted in two ways in ICC programs, as the binary minus operator or the unary minus operator. It will be the parser's job to determine which is the appropriate interpretation. Thus "-10" should now be scanned as two tokens, "-" and "10", and the parser will represent that as the unary minus operator applied to 10 in the programs' parse tree.

Working with CUP. You will put your grammar specification and tree building actions in the `parser.cup` file, from which CUP will generate `parser.java` and `sym.java`. Your actions will need to be written in Java, but your AST nodes and other supporting classes will be written in Scala. Creating a Scala object from Java code is typically routine — just use `new` as usual. However, creating Scala Lists, Maps, and Option values can be trickier since there is no analog to those concepts in Java. I have provided a helper file `ParserUtil.scala` that provides methods for constructing lists and options. The methods in that file can be invoked from your Java action code to, insert an element into a list, append lists, and so on. Feel free to add any other useful interoperability methods to that file as well.

Debugging Your Grammar. 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 `icc` directory will do the same.)

Documentation and Testing

Documentation and testing are crucial moving forward. A few basic thoughts:

Comments. Document the role of each class, what each non-trivial method does, and how each important block of code works. You need not document every line or class. For example, you will write perhaps 20–30 AST node classes. They all are roughly the same. So, write a top-level architectural comment in the root class and note conventions followed in the rest of the classes. Then comment only the most salient details or tricky parts in the rest of the files (*e.g.*, the fields linking identifiers to declarations).

Testing. You will not survive this project if you only write one or two test files. The best way to test is to write many (eg, dozens) of small cases to cover as many cases of the IC specification as possible and test as you go. To this end, plan on writing 20–30 test cases as you begin each part, rather than waiting until the night before it is due. Run all of your tests often to avoid unwittingly breaking correct behavior. The script from the last phase should serve as a good starting point for this, and I can assist in setting up additional scripts if you like. You should also document how you test your compiler so that I can reproduce your results.

Submission

By each checkpoint or due date below, be sure that your shared Bitbucket repository contains up-to-date versions of the following:

- 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. I will try to run your compilers, browse through your code, and give some feedback *at each checkpoint*.
- A brief, clear, and concise summary of where you are at each checkpoint. Place this in the `writeup` directory. This document should describe where you are, what bugs or open issues remain, and how you tested this part. The writeup need not be long. A few short sentences or a bullet list should be sufficient. (Think of this as a quick 30 second project update for your boss.)

As in the first assignment, double check your project when you submit each checkpoint by checking out a fresh copy and verifying that it works.

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. Any time on the given date in the Eastern timezone qualifies as on time.

Thursday, February 25: Your parser must successfully parse valid IC programs and report syntax errors in bad ones. Your write up directory should also contain 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.

Your project should include at least 20–30 test cases, and ideally more.

Thursday, March 3: Your parser must generate ASTs for programs, and you must support the `-printAST` command-line option.

Thursday, March 10: All of the above, plus your compiler must generate the symbol tables and perform name resolution for variable uses. That is, each variable access should be resolved to either a local variable access or a field access. Variable name resolution should be done as a separate pass over the AST after all of the symbol table information has been constructed. **You may wish to resolve class names to their declarations at the same time as variable resolution. Otherwise, you will likely need a separate pass to do that before type checking.** (Field and method name resolution will take place during the subsequent type checking phase.)

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 variable access in the program has been resolved.

Thursday, March 17: All parts due.

Friday, March 18: Happy Spring Break!