

Assignment Description

In this assignment, you will implement the syntax analysis for IC, including the AST construction. We expect you to use the code that you wrote for PA1. You are required to implement the following:

- **The IC 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 of the program.

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 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 package with the `sym.java` automatically generated by Java CUP. Also, you must 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.

The parser specifications for IC and the library signature file must be contained in the files `IC.cup` and `Library.cup`, respectively. The generated `.java` files, including `sym.java` should all be in the `IC/Parser` sub-directory.

The application of JavaCup to `Library.cup` and `IC.cup` should result with no errors or warnings and no conflicts. That is, the printout should look similar to this:

```
0 errors and 0 warnings
X terminals, X non-terminals, and X productions declared,
producing X unique parse states.
0 terminals declared but not used.
0 non-terminals declared but not used.
0 productions never reduced.
0 conflicts detected (0 expected).
```

- **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 parser 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.
- **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. One should be able to fix the problem immediately after reading the error message. It is not required to report more than one error; the execution may terminate after the first lexical, or syntactic error.

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 will report any error it encounters. In addition, your compiler must support the following command-line option to print internal information about the AST. The `"-print-ast"` option: will print at `System.out` a textual description of the constructed AST. 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).

You may find it helpful to use the graph visualization tool [Graphviz](#) for printing out information about the AST. 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.

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 the parser module and a sub-package for the AST class hierarchy. The web site contains a link to a diagram showing the exact [directory structure](#). The course web-site also contains a [zipped file](#) with the correct directory and file structure to help you get started. The zip file includes a `build.xml` file with a target for creating the parser and other useful operations.

Testing the Parser: We expect you to perform your own testing of the scanner. You should develop a thorough test suite that tests all legal syntactical structures and as many syntax errors as you can think of. We will test your parser against our own test cases – including programs that are syntactically correct, and also programs that contain syntactical errors.

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 at class (or mailbox 268):

- A brief, clear, and concise document describing the your code structure and testing strategy. Include in this document a description of your AST.
- 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 Instructions. Turn in your code electronically in your team account on the due date. Please include only the source files and your test cases in your submission. Please organize your top-level directory structure as follows:

- `src` - all of your source code.
- `doc` - documentation, including your write-up and a `PA2-README.TXT` containing a description of the class hierarchy in your `src` directory, brief descriptions of the major classes, any known bugs, and any 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.

GOOD LUCK!