

## 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](#). Note that the `sym.java` file that you wrote in PA1 will be 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 `libc.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.

NOTE: In the web-site, we include a proposed AST hierarchy to help you get started.

- **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> ]
```

**Notice that:**

- (a) There is no space between `-L` and the path to `libic.sig`; and
- (b) The text following the `-L` contains the path to the library file **including** the name of the file (which is usually `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. 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 as well as a possible AST hierarchy 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 parser. 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

You must turn in your code electronically in the team account on the due date, including a documentation write-up. **Please include only the source files in your submission, not the compiled class files or any other temporary files.**

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 will be placed here on both clarity and brevity – both in documentation and code. Make sure your code structure is well-explained in your write-up and in your javadoc documentation.

Turn in a document `PA2-DOC.XXX` (`XXX` can be one of `txt/doc/pdf`) with the following information:

- A brief, clear, and concise description of your code structure and testing strategy.
- 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. Documented bugs will be graded more forgivingly than non-documented bugs.

- A description of the grammar you created for IC (you may use CUP's `-dump_grammar` option) and any special choices you have made when creating the grammar (special precedence options etc.).
- Feedback. We are interested in hearing your overall thoughts about the assignment: how much time you spent on it, what was the most difficult or most interesting part, and how you think it could be made better. This part is optional.

**Electronic Submission Instructions.** Please organize your top-level [directory structure](#) according to the diagram shown in the web-site.

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

GOOD LUCK!