Syntax Analysis (through CUP)

In this exercise, you will:

- Implement a parser for Ice Coffee using the cup.
- Construct an *abstract syntax tree* of a program.
- Detect and report syntax errors.
- Learn about ambiguities in context-free grammars and they ways to resolve them.

1 Practical Section

IC Grammar Refer to the IC Specification, Sections 14 and 16, where a complete grammar of IC is presented. Notice that the grammar comes in two flavors:

- *Program grammar* — is used by all IC programs (Section 14).
- *Library grammar* — used to specify the IC standard library (Section 16). This is a simplified version of the *program grammar* since it only contains method declarations.

Parsing To generate the parser, you will use Java CUP, a LALR(1) automatic parser generator for Java. A link to Java CUP is (also) available on the course moodle 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 you CUP parser specification. You must modify this grammar to make it conflict free 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 8:1 (JFlex and CUP) 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 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 subdirectory.
The application of JavaCup to Library.cup and IC.cup should result with no errors and no conflicts.
For more hints you can view the slp implementation.

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 have designed the AST class hierarchy, extend your parser such that it also constructs the AST.
NOTE: In the website, 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 a 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.

Input Format The input to the parser module consists of:

1. A *program* file,
2. Optionally, a *library* signature file.

The program should conform to the *program grammar*, and the library, usually just the file `libic.sig` presented in Section 16 of the specifications, conforms to the *library grammar*.

Output Format If the program is found well-formed by the syntax analyzer, the output should contain the resulting AST. To output the tree use the PrettyPrinter class in the skeleton.

If an error is encountered (either lexical or syntactic), no output should be generated other than an error message (to either `System.out` or `System.err`, but be consistent):

> *line*`:`*column* `:` *type-of-error*`;` *error-description*

For example:

```
2:8 :  syntax error; expected '(' or ',' or ';', but found '='
```

Notice: for consistency, when several tokens are in the "expected" set as in the example above, sort them lexicographically according to their ASCII values.

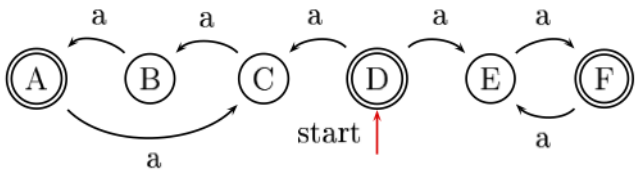
Command-line Interface You should provide a jar file with pre-configured `Main` in the default package. The usage will be:

`java -jar` *bin* ⟨*program-filename*⟩ [`-L`⟨*library-filename*⟩]

The ⟨*program-filename*⟩ is a required parameter containing the name of the program file to be parsed. The second parameter, [`-L`⟨*library-filename*⟩], is optional, and when present, contains the name of the library signature file (usually `libic.sig`) — which should be processed according to the *library grammar*. Note that there is no space between `-L` and the filename.

2 Theoretical Section

(a) Write a **context-free, right-linear grammar** that generates the language described by the following NFA:



(if you do not remember what a right-linear grammar is, refer to Wikipedia.)

(b) Describe a general method by which **any** NFA can be converted into a grammar of this form.

(c) Recall that the naïve expression grammar:

$$E \to E + E \mid E - E \mid E * E \mid E \ / \ E \mid \# \mid (\ E\ )$$

(where # represents a number token) is ambiguous and therefore not suitable for parsers.

Someone suggested that we use layering to write an equivalent unambiguous grammar:

$$\begin{aligned} E &\to T + E \mid T - E \mid T \\ T &\to F * T \mid F \ / \ T \mid F \\ F &\to \# \mid (\ E\ ) \end{aligned}$$

What is the problem with this formulation?

Illustrate your claim by an example for an expression and its respective parse tree.

(d) You are presented with the following CFG:

$$\begin{aligned} S &\to A\ C \\ A &\to \texttt{a}\ A \mid \texttt{b} \\ C &\to C\ D \mid \texttt{c} \\ D &\to \texttt{d} \end{aligned}$$

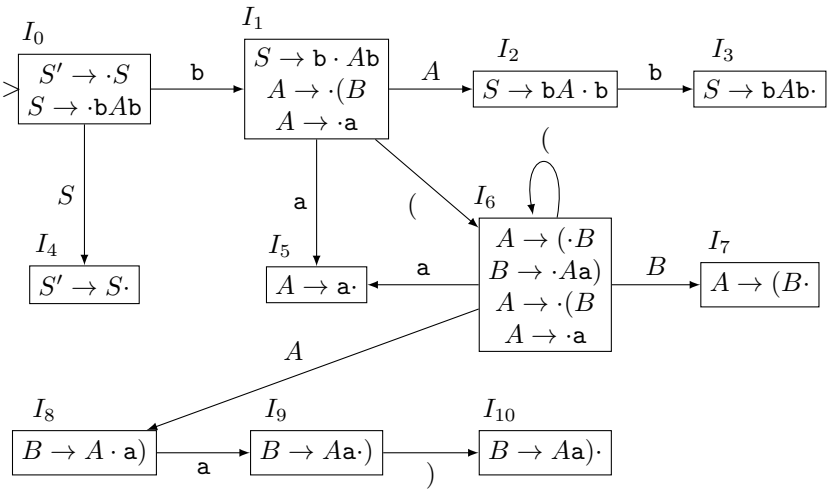
where `a`, `b`, `c`, `d` are terminals.

Is this grammar $LL(1)$? If not, fix it so that it would be.

Construct the FIRST and FOLLOW sets, fill the $LL(1)$ parse table, and show how an $LL(1)$ parser would process the input "`abcdd`".

Here is a nice tutorial that can help you get started.

(e) We found this Characteristic Finite State Machine in the classroom.



All the states are accepting states.

(i) What is the CFG for which this CFSM was constructed?

(ii) The sequence "`b a b`" is a valid word in the language generated by this grammar. Why, therefore, is there no accepting run of the CFSM on this word?

(iii) Write down the sequence of shift/reduce actions an LR(0) parser will make on the input —

`b ( ( a a ) a ) b`

and the contents of the parser stack after each action takes place.

3 Bonus Objectives

(a) The IC language has a small design flaw, that was overlooked in order to simplify the syntax. Since variable declaration (ie, int x;) is considered as one of the "Statement" non-terminals, the following code, for example, is syntactically legal:

```
if (...)  int x;
```

This type of code — a conditional statement followed immediately by a variable declaration not inside curly braces — is considered a syntax error in any real C-like language (can you guess why?).

Objective: Make the necessary changes to the IC language syntax and to your parser so that this situation is detected as a syntax error. (8 points)

Important!!

The error must be detected **by the parser** — that is, the program above should be considered grammatically incorrect — **not** by the AST construction phase. So in a program containing these two lines:

```
if (x) int y;
a b;
```

A syntax error should be reported on the first line, not the second.

(b) CUP allows you to implement a limited form of error recovery, using the error keyword. This allows the parser to essentially skip over buggy branches of the syntax tree and continue parsing the rest of it, possibly discovering more errors. Consult the CUP manual for more information.
Objective: Implement error recovery functionality as best as you can. (7 points)