# CS3410: Software Engineering Lab
# Writing LR(1) parser for JavaCC - Interim Project Status Report

Aravind S CS11B033
S Akshayaram CS11B057
R Srinivasan CS11B059
S K Ramnanadan CS11B061
Adit Krishnan CS11B063

April 14, 2014

## 1 Problem Definition

JavaCC is a Java parser generator written in Java Programming Language. JavaCC is by far the most popular parser generator used with Java applications. JavaCC generates top down parsers (recursive decent) as opposed to bottom up parsers generated by YACC-like tools i.e it implements $LL(k)$ parser. This does not allow left recursive grammars to be parsed. We propose to add LR(1) parser for the JavaCC program which will enable it to parse left recursive grammars as well.

## 2 Code Details

The code for the parser generation can be found in `src/org/javacc/parser`. The various options that govern the parser like lookahead, debug-enable, etc are taken as input and pushed into a class `Options`.

We skip over the tokenization portion as it is not relevant to us. We only need to know the API updated by the tokenizer. Before the parser is called, the tokens and productions are updated in global constructs in `JavaCCGlobals`. The suitable containers for Tokens and Productions can be found in the `Token` and `NormalProduction` respectively. Productions supplied in the `.jj` file to be parsed may be Java code as well in which we have a sub-class for Productions called `JavaCodeProduction`. A similar case holds for `BNFProduction` as well.

The RHS's of productions is generically called `Expansion` which can again be of several types and hence has sub-classes:

1. `Choice` for `|` in a RegEx
2. `Non-Terminal`
3. `OneOrMore` for `+` in a RegEx
4. `RegularExpression`
5. `Sequence` for a sequence of any of these
6. `ZeroOrMore` for `*` in a RegEx
7. `ZeroOrOne` for `?` in a RegEx
8. `TryBlock` for a try-catch block

For example: `(<id>+|\n)(\t)*` would throw up firstly a `Sequence`, the first of which is a `Choice` of a `OneOrMore` and a simple token, and the second of which is a `ZeroOrMore`.

`RegularExpression` also has similar sub-classes for the same reason and these files are prefixed with an `R` in their name.

Another notable point is that the code allows for C++-specific constructs as well and this is specifically handled everywhere in the code.

Now, let's walk through a typical run. As always, the `Main` class begins the process. After some initializations, `ParseGen` is called (note that there are variants for CPP like `ParseGenCPP`) after which a Lexer is also generated (using `LexGen` or `LexGenCPP`) but we will not delve into the lexing.

`ParseGen` mainly does some tertiary code dumping amidst which it invokes `ParseEngine` which is the heart of the parser generator. This file contains functions which build First sets and generate code to do LL(k) parsing. The function `buildLookaheadChecker()` does this work which has a mini-state machine to generate the suitable lookahead sequences.

Another notable feature is the use of buffers which hold portions of the code which are dumped at requisite points of time. Also, file `JavaCC.jj` contains the grammar and actions that describe `JavaCCParser`. When passed as input to `JavaCCParser` it generates another copy of itself.

# 3 Work Division

1. Srinivasan R.: Reading the code and realizing the implementation of $LL(k)$ parser generator in the code.

2. Aravind S. and Adit Krishnan: Ensuring the code written follows design patterns and drawing out a design of the existing code and see patterns used.

3. Ramnandan S.K: Reading about $LR(1)$ parser generator and implementing the $LR(1)$ parser generator

4. Akshayaram S: Coming with left recursive grammar to test the changes made and implementation of $LR(1)$ parser generator.

# 4 Work Done and Expectation

While the exact functioning of the $LL(k)$ parser is not of relevance, this reading has provided a sufficiently strong understanding of the organization of the code which is of utmost importance in formulate our $LR(1)$ parser generator. Substantial work would go into the file ParseEngine with probably measly cleanups in the other files. But, we must mention beforehand that catering to all options offered by the current setting of the JavaCC parser like support for C++ code, etc. may seem a bit ambitious. However, we hope to ensure the working on a simple left-recursive grammar. We expect to complete by the first week of May.