CSC488 Assignment 2

Vishrant Vasavada, Tian Ze(Peter) Chen, Anthony Vandikas, Felix Lu, Badrul Chowdhury

4th February, 2015

1. Overview

Using the rules provided in CSC488S language grammar sheet, we began by converting those into Java CUP syntax. The first thing we observed was that reference grammar wasn't LALR friendly. So the next step was to begin transforming the rules into LALR friendly rules, also providing the proper precedence rules to the operations.

2. Shift/Reduce Conflicts

This was the most difficult problem that we addressed when devising our grammar. For example, the reference grammar uses following style to accept statement lists:

```
statement    ::=    ....
                |    ....
                |    statement statement          % sequence of statements
```

The first step taken by an LALR parser would be to push a "statement" on the stack. Now the parser has two alternatives:

1) Assume that the statement satisfies the grammar rule for some of the recent parse trees, and reduce them into a single tree with a new root symbol.
2) Or shift an additional statement onto the stack to satisfy "statement statement" rule.

To remove this ambiguity, we explicitly added following new rule:

```
statements    ::=    statement
                 |    statements statement
```

Reduction to "statement" is only possible when there are no more statements left to shift onto the stack, thus resolving the SHIFT/REDUCE CONFLICT. We applied this transformation to statements, declarations and other such things.

3. Precedence

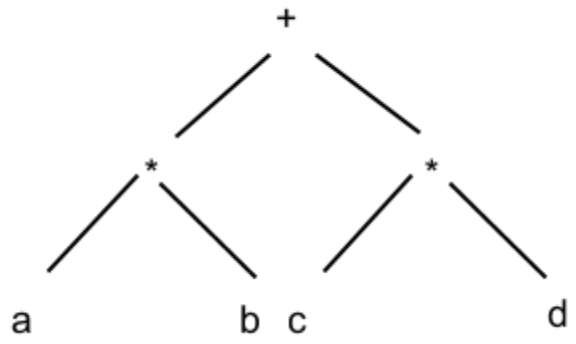
We also observed that the reference grammar rules don't really handle the precedence rules mentioned in language documentation. The following is a snippet from our parser that associates (unary '-' , '*', '/', '+', binary '-') from left-to-right:

```
expr2       ::=     expr1
                |   expr2 PLUS expr1
                |   expr2 MINUS expr1

expr1       ::=     expr0
                |   expr1 TIMES expr0
                |   expr1 DIVIDES expr0

expr0       ::=     ....    % terminal
                |   MINUS expr0
```

For instance, let us consider a simple example involving product and addition: a * b + c *d. When we view the expression as tree, it is easy to see that the higher precedence operations appear below lower precedence operations. Hence, the expression will look like:



Working top down as per our snippet code, the root of an expression begins at an addition operator. This reflects the fact that addition has the lowest precedence. Currently, we are at "expr2 PLUS", where expr2 could consist of as many terms as present in the input sum (in our case: a*b and c*d). When no terms are left in expr2, we move down the parse tree to expr1. The symbol expr1 would take as many terms as are present in input product (in our case: a, b, c, and d.). When no terms are left, we then move to the terminals and the process is complete.

4. Difficulties Faced and Conclusion

Testing was a challenge mainly because the designers had to consider the many ways users could (mis-)use the language. For example, it is not clear how the parser should handle the statements such as **b = true | -1** or **b = false & -1**. We considered these problems as issues for the semantic checker, so they are not caught by the parser.

The designed grammar handles our (extensive) test suite gracefully, thereby giving us confidence that the implementation is trustworthy. We say mostly because, as is the case with most things in life, there probably is room for improvement… only time will tell.