# CSC488
# ASSIGNMENT 2
# GRAMMAR DESIGN

DANIEL BLOEMENDAL

## Contents

## 1. Overview

We began by translating the production rules in the reference grammar into Java CUP syntax. However, the reference grammar is certainly not LALR. Furthermore, the reference grammar does not observe any of the operator precedence rules described in the language documentation. So the next goal was to begin transforming the production rules into LALR friendly rules, observing precedence in expressions. The challenges encountered during this transformation will be outlined below.

## 2. Shift/Reduce conflicts

The reference grammar uses the following style to accept lists. It is used for instance to recognize lists of statements or declarations.

```
statement
    ::= ...
    |   ...
    |   statement statement
    ;
```

Unfortunately with LALR parsers, as soon as a statement is shifted on to the stack, there are two valid actions. The statement on stack can be reduced to the "statement" non-terminal, or an additional statement can be shifted on to the stack to satisfy the alternative

```
    |   statement statement
```

The solution was to add explicit list production rules as follows.

```
statement_list
    ::= statement_list statement
    |   statement
    ;
```

This resolves the issue as a reduction to "statement_list" is only possible if there are no remaining statements left to shift on to the stack. This transformation was applied for statements, declarations and other such lists.

## 3. Reduce/Reduce conflicts

Another problem encountered during the process of transforming the grammar was an ambiguity when it came to variables and parameters. According to the reference grammar both a "variablename" and and "parametername" can be reduced to a "variable". However, this is ambiguous as both "variablename" and "parametername" reduce from "IDENT". So in the case of "variable" if an "IDENT" is shifted on to the stack there is more than one valid reduction. It can be reduced to "variablename" or "parametername". This is a reduce/reduce conflict.

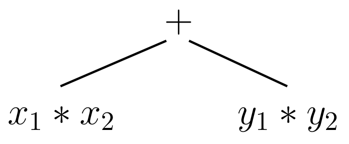
```
variable
    ::= variablename
    |   parametername
    |   variablename L_SQUARE expression R_SQUARE
    |   variablename L_SQUARE expression COMMA expression R_SQUARE
    ;

variablename  ::= IDENT;
parametername ::= IDENT;
```

## 4. Expressions

4.1. **Design.** The expression rules were built as a set of left recursive productions obeying the precedence rules set out in the reference grammar documentation. When viewing an expression as a tree, it is easy to see that the higher the precedence of an operation the closer it is to the bottom, or the leaves of the tree.

Figure 1. Expression: $x_1 * x_2 + y_1 * y_2$



To illustrate how the expression productions were designed, we will explore a simplified grammar involving only addition and multiplication. As in the reference grammar, addition has lower precedence than multiplication. As seen in figure 1, the higher precedence multiplication operations are at the leaves of the tree. The production rules can be constructed to reflect this tree like structure of expressions.

```
expressions
    ::= add_expr
    ;

add_expr
    ::= add_expr PLUS mul_expr
    |   mul_expr
    ;

```

```
10 mul_expr
11     ::= mul_expr TIMES terminal
12     |   terminal
13     ;
```

Working from the top down, in our simplified grammar the root of an expression begins at an addition expression, as seen in line 2. This reflects the fact that addition has the lowest precedence. The addition production then eats up as many terms as exist in the input sum. This is found in line 6 of the simplified grammar. When no terms are left in the sum "add_expr" falls through to "mul_expr" as seen in line 7 of the simplified grammar. If there is only one term, where no "+" operator exists in the expression, "add_expr" immediately falls through to "mul_expr". The "mul_expr" eats up all product operations and ends when there are none left. At the end only terminals remain and the process is complete.

It should be noted, that we explored the grammar from the top down. The LALR parser of course works from the bottom up. However, it is simpler to describe the design of the grammar working from the top down. The final product is a slightly more complicated application of concepts in the simplified grammar.

4.2. **Difficulties.** While designing test cases we encountered a string that belongs to the reference language but is not recognized by our parser.

```
b := 1 * not false
```

While the string is not semantically valid due to the type mismatch, where a boolean is one of the operands of a numeric operator, it is a valid string in the reference language. The expression productions rules were designed to obey operator precedence rules. The way the "not" rule was setup was to expect a predicate expression after a list of "not" terminals. While this is appropriate for practical purposes as it will accept all semantically valid expressions we have tested, it unfortunately will not recognize all syntactically valid expressions. We decided not to solve this problem as our only solutions increased grammar complexity for no practical gain.