Lab 3

CMPT 432 - Spring 2023 | Dr. Labouseur

Josh Seligman | joshua.seligman1@marist.edu

February 19, 2023

1 Crafting a Compiler

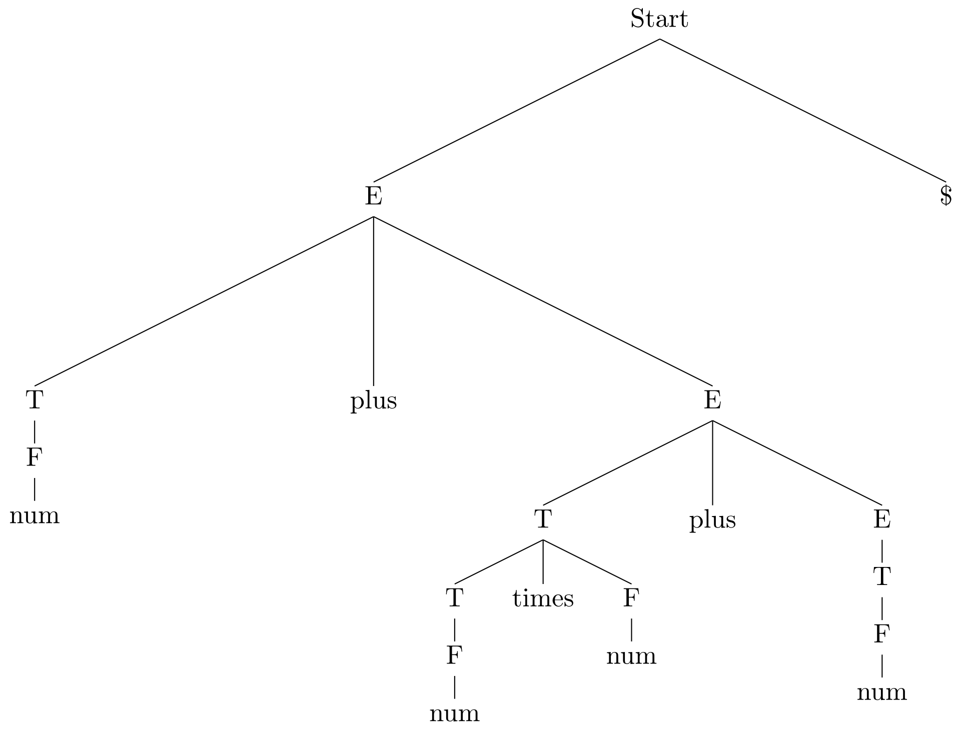
1.1 Exercise 4.7

Given the following grammar:

1. Start –> E $
2. E –> T plus E
3. | T
4. T –> T times F
5. | F
6. F –> (E)
7. | num

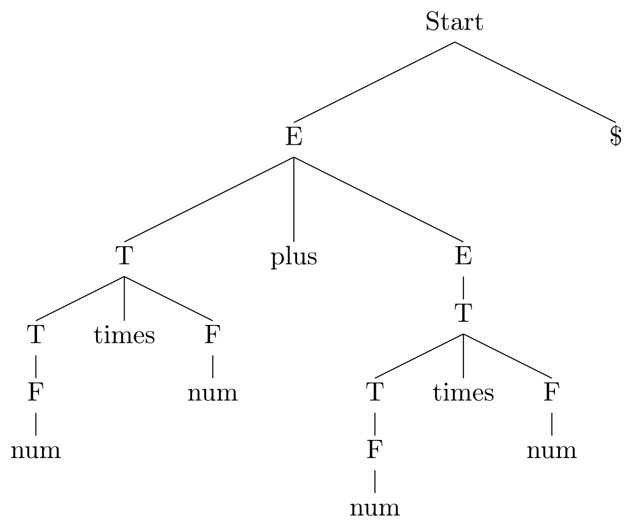
1. Show the leftmost derivation of the following string: num plus num times num plus num $

   S. Start
   1. E $
   2. T plus E $
   5. F plus E $
   7. num plus E $
   2. num plus T plus E $
   4. num plus T times F plus E $
   5. num plus F times F plus E $
   7. num plus num times F plus E $
   7. num plus num times num plus E $
   3. num plus num times num plus T $
   5. num plus num times num plus F $
   7. num plus num times num plus num $



2. Show the rightmost derivation of the following string: num times num plus num times num $

   S. Start
   1. E $
   2. T plus E $
   3. T plus T $
   4. T plus T times F $
   7. T plus T times num $
   5. T plus F times num $
   7. T plus num times num $
   4. T times F plus num times num $
   7. T times num plus num times num $
   5. F times num plus num times num $
   7. num times num plus num times num $



3. Describe how this grammar structures expressions, in terms of the precedence and left- or right- associativity of operators.

   This grammar structures expressions by forcing the higher precedence operators to be placed lower in the parse tree. The grammar creates this precedence by having lower precedence operators go through the productions for higher-precedence operators before reaching a terminal, which will enable the higher-precedence operator to be evaluated first because it is lower in the CST. Right-most operators also get precedence over left-most operators because production rule 2 says that E –> T plus E, which means that if there is another plus, the E on the right would have to expand unless the T goes down eventually to production rule 6 and adds the parentheses.

1.2 Exercise 5.2c

Given the following LL(1) grammar, write a recursive descent parser.

1. Start –> Value $
2. Value –> num
3. | lparen Expr rparen
4. Expr –> plus Value Value
5. | prod Values
6. Values –> Value Values
7. | λ

```
func parseStart() {
    parseValue()
    match($)
}

func parseValue() {
    if peek() == lparen {
        match(lparen)
        parseExpr()
        match(rparen)
    } else {
        match(num)
    }
}

func parseExpr() {
    if peek() == plus {
        match(plus)
        parseValue()
        parseValue()
    } else {
        match(prod)
        parseValues()
    }
}

func parseValues() {
    // Check for starter tokens for Value
    if peek() == num || peek() == lparen {
        parseValue()
        parseValues()
    } else {
        // Nothing to do here, epsilon condition
    }
}
```

2 Dragon

2.1 Exercise 4.2.1

Given the following CFG and the string aa+a*:

1. S –> S S +
2. | S S *
3. | a

1. Give a leftmost derivation of the string.

   Start. S
   2. S S *
   1. S S + S *
   3. a S + S *
   3. a a + S *
   3. a a + a *

2. Give a rightmost derivation of the string.

   Start. S
   2. S S *

3. S a *
1. S S + a *
3. S a + a *
3. a a + a *

3. Give a parse tree for the string.