Gianna Julio

CMPT432 - Design of Compilers
Lab 3

1. [***Crafting a Compiler* - Exercise 4.7**] A grammar for infix expressions follows:

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

(a) Show the leftmost derivation of the following string:

```
'num plus num times num plus num $'
<Start>
<E> $
<T> plus <E> $
<F> plus <E> $
num plus <E> $
num plus <T> $
num plus <T> times <F> $
num plus <F> times <F> $
num plus num times <F> $
num plus num times (<E>) $
num plus num times (<T> plus <E>) $
num plus num times (<F> plus <E>) $
num plus num times (num plus <E>) $
num plus num times (num plus <T>) $
num plus num times (num plus <F>) $
num plus num times (num plus num) $
num plus num times num plus num $
```

(b) Show the rightmost derivation of the following string:

```
'num times num plus num times num $'
<Start>
<E> $
<T> plus <E> $
<T> plus <T>$
<T> plus <T> times <F> $
<T> plus <T> times num $
<T> plus <F> times num $
<T> plus num times num $
<T> times <F> plus num times num $
<T> times num plus num times num $
<F> times num plus num times num $
num times num plus num times num $
```

(c) Describe how this grammar structures expressions, in terms of the precedence and left- or right- associativity of operators.
This grammar structures expressions so that multiplication has precedence over addition. Because T, which expands into a multiplication expression, comes first in each expression in the grammar, T is the first nonterminal to be expanded. Therefore, multiplication is the first operation to be performed. The grammar is also structured to be left-associative for both addition and multiplication. The leftmost term is expanded first, so within a sequence of the same operator(all plus or all times), calculations are performed from left to right.

2. [***Crafting a Compiler* - Exercise 5.2c**] Construct a recursive-descent parser based on the grammar:

1	Start	$\rightarrow$	Value $
2	Value	$\rightarrow$	num
3		|	lparen Expr rparen
4	Expr	$\rightarrow$	plus Value Value
5		|	prod Values
6	Values	$\rightarrow$	Value Values
7		|	λ

```
parseStart() {
  parseValue()
  match('$')
)

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

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

parseValues() {
  if(current != emptyString)
  parseValue()
  parseValues()
}

match(token) {
  return (token in expectedTokens)
}
```

3. [*"Dragon" Textbook* - **Exercise 4.2.1**] Consider the context-free grammar:
`S → S S + | S S * | a`
and the string `aa+a*`

(a) Give a leftmost derivation for the string.
```
S
SS*
SS+S*
aS+S*
aa+S*
aa+a*
```

(b) Give a rightmost derivation for the string.
```
S
SS*
Sa*
SS+a*
Sa+a*
aa+a*
```

(c) Give a parse tree for the string.