Lab One

Blake Mackey

blake.mackey1@Marist.edu

March 13, 2021

1 Dragon Book Exercises

1.1 Exercise 4.2.1

Considering the grammar:

```
S -> SS+ | SS* | a
1. start -> S
2. S -> SS+
3.    | SS*
4.    | a
```

And the string:

```
aa + a*
```

The following derivations and Parse Tree can be made.

1.1.1 Part A - Left Most Derivation

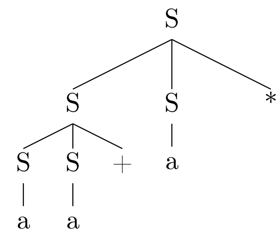
```
1. start -> S
3. S -> SS*
2. SS* -> SS+S*
4. SS+S* -> aS+S*
4. aS+S* -> aa+S*
4. aa+S* -> aa+a*
```

1.1.2 Part B - Right Most Derivation

```
1. start -> S
3. S -> SS*
4. SS* -> Sa*
2. Sa* -> SS+a*
```

4. SS+a* –> Sa+a*
4. Sa+a* –> aa+a*

1.1.3 Part C - Parse Tree



2 Crafting a Compiler Exercises

2.1 Exercise 4.7

Grammar:

```
1. start -> E $
2. E -> T plus E
3.    | T
4. T -> T times F
5.    | F
6. F -> (E)
7.    | num
```

2.1.1 Part A - Left Most Derivation

String - num plus num times num plus num

```
1. start -> E $
2. E $ -> T plus E $
5. T plus E $ -> F plus E $
7. F plus E $ -> num plus E $
2. num plus E $ -> num plus T plus E $
4. num plus T plus E $ -> num plus T times F plus E $
5. num plus T times F plus E $ -> num plus F times F plus E $
7. num plus F times F plus E $ -> num plus num times F plus E $
7. num plus num times F plus E $ -> num plus num times num plus E $
3. num plus num times num plus E $ -> num plus num times num plus T $
5. num plus num times num plus T $ -> num plus num times num plus F $
7. num plus num times num plus F -> num plus num times num plus num
```

2.1.2 Part B - Right Most Derivation

```
1. start -> E $
2. E $ -> T plus E $
3. T plus E $ -> T plus T $
5. T plus T $ -> T plus T times F $
7. T plus T times F $ -> T plus T times num $
5. T plus T times num $ -> T plus F times num $
7. T plus F times num $ -> T plus num times num $
4. T plus num times num $ -> T times F plus num times num $
7. T times F plus num times num $ ->
5. T times num plus num times num $ -> F times num plus num times num $
7. F times num plus num times num $ -> num times num plus num times num $
```

2.1.3 Part C

2.2 Exercise 5.2 C

For the grammar:

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

The following recursive descent parser can be written.

```
function ParseStart {
    ParseValue();
    match($);
}

function ParseValue {
    if token == num {
        match(num);
    }
    else {
        match(lparen);
        ParseExpr();
        march(rparen);
    }
}

function ParseExpr {
    if token == plus {
        match(plus);
        ParseValue();
        ParseValues();
    }

    else {
        match(prod);
        ParseValues();
    }
}

function ParseValues {
    if token == Value {
        match(Value);
        ParseValues();
    }

    else {
        match(lambda);
    }
}
```