Lab Three

John DeFalco
john.defalco1@marist.edu

September 2019

Goal

Using the parser to turn tokens into sentences

Examples from the Readings

1 Crafting a Compiler

Below are the examples listed on the requirements document for the lab from the *Crafting a Compiler* textbook.

1.1 Problem 4.7

Grammar provided:

```
Start -> E $
E     -> T plus E
       | T
T     -> T times F
       | F
F     -> ( E )
       | num
```

(A) Show left-most derivation for provided string: "num plus num times num plus num $"

```
Start   => E $
        => T plus E $
        => F plus E $
        => F plus T plus E $
        => F plus T times F plus E $
        => num plus T times F plus E $
        => num plus F times F 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 $
```

(B) Show the right-most derivation for provided 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.

The grammar provided in this example structures expressions in such a way that the times operator has a higher precedence than the plus operator, as seen from its position being lower within the grammar. The times operator is left-associative, as the production rule including the operator is left-recursive. The plus operator is right-associative, as the production rule including the operator is right-recursive.

1.2 Problem 5.2 (c)

Grammar provided:

```
Start   -> Value $
Value   -> num
         | lparen Expr rparen
Expr    -> plus Value Value
         | prod Values
Values  -> Value Values
         | λ
```

Recursive-descent parser based on the grammar:

Match

```
procedure Match(tokenStream, token)
    if (tokenStream.peek() == token)
        then tokenStream.advance()
    else
        error(Expected token)
    end if
end
```

Start

```
procedure Start()
    call Value()
    call match($)
end
```

Value

```
procedure Value()
    switch (...)
        case tokenStream.peek() == num
            call Match(num)
        case tokenStream.peek() == lparen
            call Match(lparen)
            call Expr()
            call Match(rparen)
    end switch
end
```

Expr

```
procedure Expr()
    switch (...)
        case tokenStream.peek() == plus
            call Match(plus)
            call Value()
            call Value()
        case tokenStream.peek() == prod
            call Match(prod)
            call Values()
    end switch
end
```

Values

```
procedure Values()
    switch (...)
        case tokenStream.peek() == Value
            call Value()
            call Values()
        case tokenStream.peek() == λ
            // no error, empty string
    end switch
end
```

2 Dragon Book

Below is the example listed on the requirements document for the lab from the *Compilers (Dragon)* textbook.

2.1 Problem 4.2.1

Given grammar:

```
S    -> S S +
      | S S *
      | a
```

Given string: "aa + a*"

(A) Give leftmost derivation for the string

```
Start   => S S *
        => S S + S *
        => a S + S *
        => a a + S *
        => a a + a *
```

(B) Give rightmost derivation for the string

```
Start    => S S *
         => S a *
         => S S + a *
         => S a + a *
         => a a + a *
```

(C) Give a parse tree for the string

S
-S
–a
-S
–S
—a
–+
–S
—a
-*