Lab Three

Alex Smith

alex.smith1@Marist.edu

February 20, 2019

1 Crafting a Compiler

1.1 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> 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 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 with nonterminals on the left hand side and terminals on the right hand side. It uses a right associativity of its operators for <E> as <E> has the potential for the most expansion in the expression <T> plus <E>, but the grammar uses left associativity for <T> as <T> is the most expansive in the expression <T> times <F>.

1.2 Exercise 5.2

Consider the following grammar, which is already suitable for LL(1) parsing:

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

(c) Construct a recursive-descent parser based on the grammar.

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

parseValue(){
if(currentToken is num){match(num)}
else{
match(()
parseExpr()
match ())
}
}
```

```
parseExpr(){
if(currentToken is plus){
match(plus)
parseValue()
parseValue()
}
else{
match(prod)
parseValues()
}
}

parseValues(){
if(currentToken is number or ( ){
parseValue()
parseValues()
}
else {
/* epsilon production */
}
}

match(expectedToken){
retVal = false
if(currentToken in expectedToken){
retVal = true
}
return retVal
}
```

2 The Dragon Book

2.1 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>
<S><S>*
<S><S> + <S>*
a<S> + <S>*
aa + <S>*
aa + a*

b) Give a rightmost derivation for the string.

<S>
<S><S>*
<S>a*
<S><S> + a*
<S>a + a*
aa + a*

c) Give a parse tree for the string.