Lab Three - Turning tokens into sentences with your Parser

Murray Coueslant

murray.coueslant1@marist.edu

October 16, 2019

1 The Dragon Book

1.1 Exercise 4.2.1

Consider the context-free grammar:

1. S -> S S +
2. | S S *
3. | a

and the string aa + a*.

1.1.1 Problem a

Problem Give a leftmost derivation for the string.

Solution

- S
- S S * (2)
- S S + S * (1)
- a S + S * (3)
- a a + S * (3)
- a a + a * (3)

1.1.2 Problem b

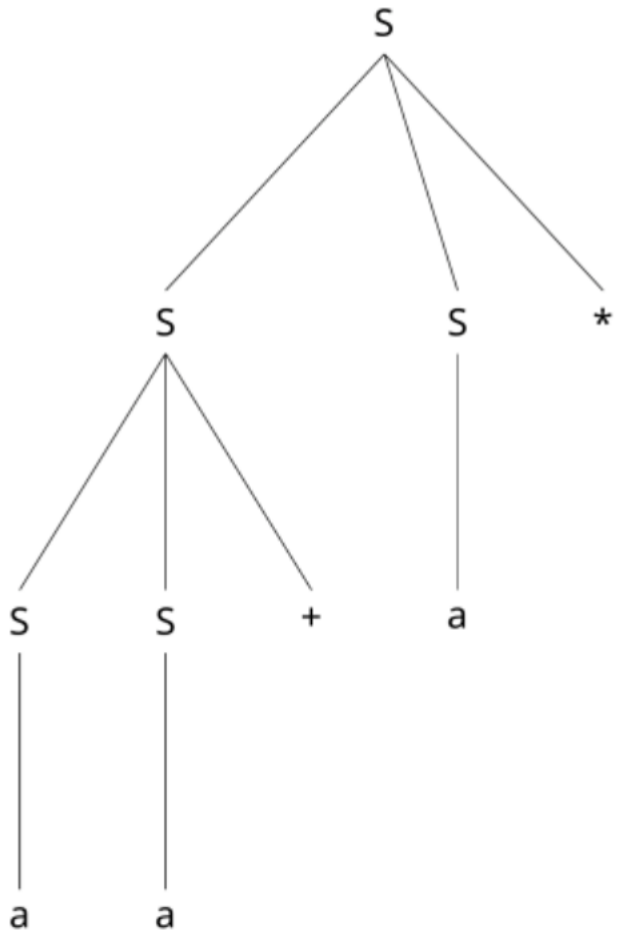
Problem Give a rightmost derivation for the string.

Solution

- S
- S S * (2)
- S S + S * (1)
- S S + a * (3)
- S a + a * (3)
- a a + a * (3)

1.1.3 Problem c

Problem Give a parse tree for the string.



Solution

2 Crafting A Compiler

2.1 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

2.1.1 Problem A

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

Solution

- E $ (1)
- T plus E $ (2)
- T plus T plus E $ (2)
- F plus T plus E $ (5)
- F plus T times F plus E $ (4)
- num plus T times F plus E $ (7)
- num plus F times F plus E $ (5)
- num plus num times F plus E $ (7)
- num plus num times num plus E $ (7)
- num plus num times num plus T $ (3)
- num plus num times num plus F $ (5)
- num plus num times num plus num $ (7)

2.1.2 Problem B

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

Solution

- E $ (1)
- T plus E $ (2)
- T plus T $ (3)
- T plus T times F $ (4)
- T plus T times num $ (7)
- T plus F times num $ (5)
- T plus num times num $ (7)
- T times F plus num times num $ (4)
- T times num plus num times num $ (7)
- F times num plus num times num $ (5)
- num times num plus num times num $ (7)

2.1.3 Problem C

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

Solution This grammar gives most precedence to the 'times' operation. This can be seen in the derivation by the fact that it tends to appear later than the 'plus' operation, which points to the fact that it would appear lower on the parse tree and therefore carries a higher precedence than the 'plus' operation. The grammar has two expression types, 'E' and 'T', and an 'F' type to handle parenthesis. The 'plus' operation is right associative, we see this in construction two when E appears again on the right hand side of the 'plus' terminal. T is left associative in this grammar as we can see in construction two the T non-terminal appears again on the left hand side of the 'times' terminal. In a reduction these associativities inherent in the grammar describe how to group the operations in order to reach a result, in a derivation they inform the shape of the parse tree.

2.2 Exercise 5.2c

Problem 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

Construct a recursive-descent parser based on the grammar.

Solution

```
def ParseProgram(){
    ParseStart();
}

def ParseStart(){
    ParseValue();
    MatchAndConsume($);
}

def ParseValue(){
    switch(currentToken){
        case num:
            MatchAndConsume(num);
            break;
        case lparen:
            MatchAndConsume(lparen);
            ParseExpr();
            MatchAndConsume(rparen);
            break;
        default:
            ReportMalformedValue();
            return;
    }
    Ascend();
    return;
}

def ParseExpr(){
    switch(currentToken){
        case plus:
            MatchAndConsume(plus);
            ParseValue();
            ParseValue();
            break;
        case prod:
            MatchAndConsume(prod);
            ParseValues();
            break;
        default:
            ReportMalformedExpr();
            return;
    }
    Ascend();
    return;
}

def ParseValues(){
    switch(currentToken){
        case Value:
            ParseValue();
            ParseValues();
            break;
        default:
            // lambda case
            break;
    }
    Ascend();
    return;
}

def MatchAndConsume(){
    // subroutine to consume the current token and add corresponding tree nodes
}
```