Project Two

Collin Drake
Collin.Drake1@Marist.edu

March 6, 2024

Lab 3

Crafting a Compiler Exercises:

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

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

   S. Start
   1. E $
   2. T plus E $
   5. F plus E $
   7. num plus E $
   2. num plus T plus E $
   4. num plus T times F plus E $
   5. num plus F times F plus E $
   7. num plus num times F plus E $
   7. num plus num times num plus E $
   3. num plus num times num plus T $
   5. num plus num times num plus F $
   7. num plus num times num plus num $

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

S. Start
1. E $
2. T plus E $
3. T plus T $
4. T plus T times F $
7. T plus T times num $
5. T plus F times num $
7. T plus num times num $
4. T times F plus num times num $
7. T times num plus num times num $
5. F times num plus num times num $
7. num times num plus num times num $

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

   The grammar provided above structures expressions with a rightmost operator precedence. I came to this result because of production 2. I noticed that within this production we can see that E is located on the right side of the expression meaning that larger expansions would have to occur on the right side of the production. Also, we know operators of higher importance are placed lower on the parse tree when performing a depth-first traversal.

Exercise: 5.2c

Construct a recursive-descent parser based on 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 | Λ

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

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

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

parseValues(){
    if(Token == lparen || Token == num){
        parseValue();
        parseValues();
    }
    else{
        match($\Lambda$);
    }
}
```

Dragon Exercises:

Exercise: 4.2.1 a,b,c

Consider the context-free grammar:

1. S $\rightarrow$ S S +
2. | S S *
3. | a

and the string aa + a*

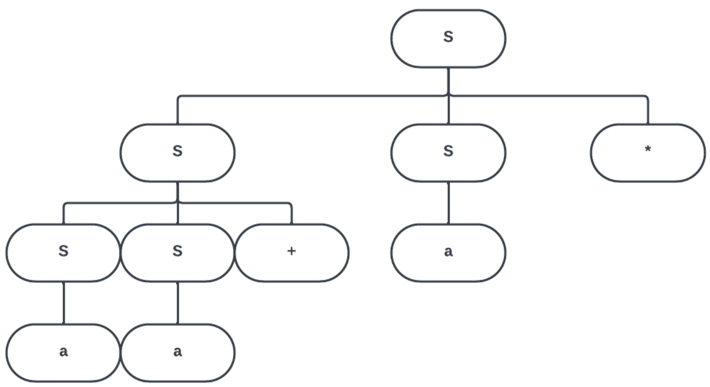
a) Give a leftmost derivation for the string

S. S
2. S S *
1. S S + S *
3. a S + S *
3. a a + S *
3. a a + a *

b) Give a rightmost derivation for the string

S. S
2. S S *
3. S a *
1. S S + a *
3. S a + a *
3. a a + a *

c) Give a parse tree for the string



Lab 4

Crafting a Compiler Exercises:

Exercise: 4.9

Compute First and Follow sets for the nonterminals of the following grammar

1. S → a S e
2. |B
3. B → b B e
4. |C
5. C → c C e
6. |d

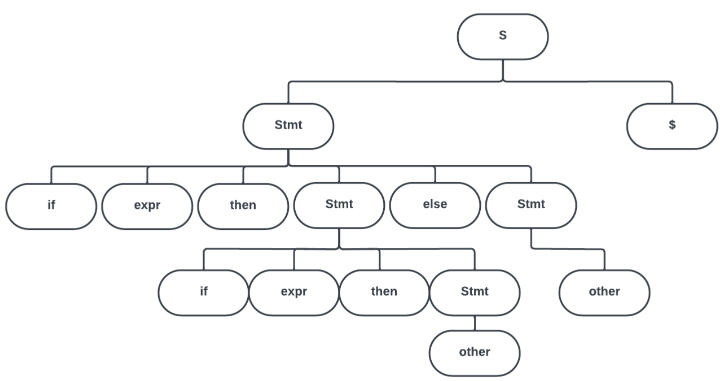
S: First{a,b,c,d} – Follow{$,e}
B: First{b,c,d} – Follow{$,e}
C: First{c,d} – Follow{$,e}

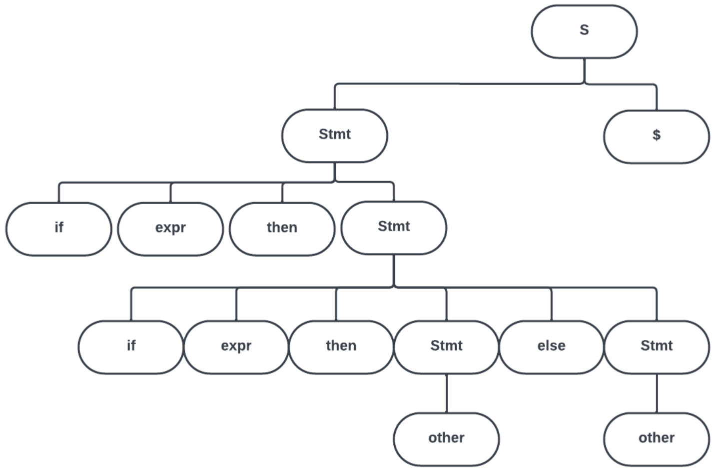
Exercise: 5.10

Show the two distinct parse trees that can be constructed for

if expr then if expr then other else other

1 S → Stmt $
2 Stmt → if expr then Stmt else Stmt
3 | if expr then Stmt
4 | other





using the grammar given in Figure 5.17. For each parse tree, explain the correspondence of then and else

In regards to the first parse tree, we notice that an "if" is always followed by a "then", however, an "else" does not always follow behind a "then". The first parse tree contains an "else" within the first production expansion, but within the second tree, it occurs in the second expansion. Therefore, we understand that an "else" node can only occur after the presence of a "then", but it is not required.

Dragon Exercises:

Exercise: 4.4.3

Compute FIRST and FOLLOW for the grammar of Exercise 4.2.1

1. S $\rightarrow$ S S +
2. | S S *
3. | a

S: First{a} – Follow{$,+,*,a}

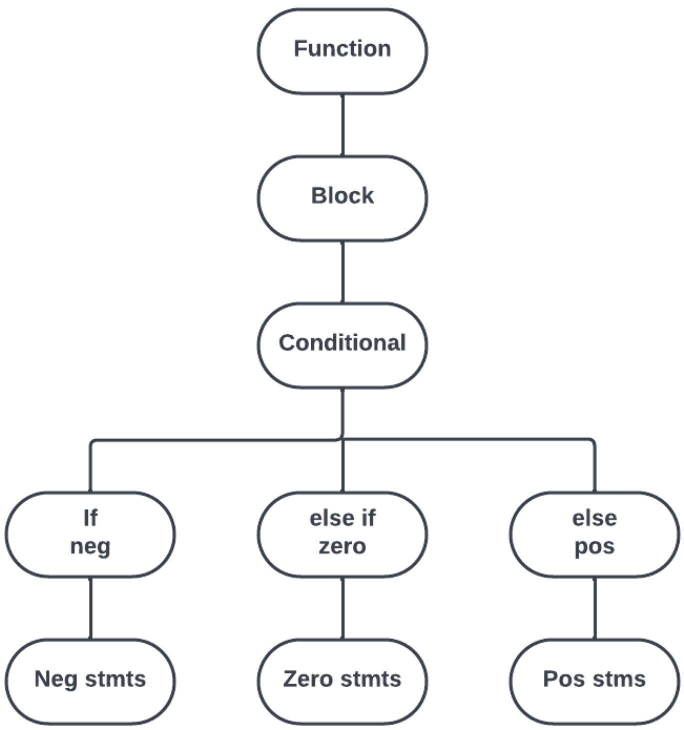
Lab 5

Crafting a Compiler Exercises:

Exercise: 9.2

Assume that we add a new kind of conditional statement to C or Java, the signtest. Its structure is:

signtest (exp) {
neg: stmts
zero: stmts
pos: stmts
}



1 Project 2

1.1 Syntax Analysis

Syntax Analysis, otherwise known as Parsing is the second step within the Compilation process. In the Parser we accept the Token Stream produced by the Lexer and validate each token with the language grammar. As the Parser verifies that each token aligns properly with the grammar it builds a Concrete Syntax Tree, containing root, branch, and leaf nodes representing each terminal and non-terminal within the grammar. The result of the Parsing process is a parse tree.

To test my Parser I used the test cases below:

Test Case 1

```
        /*test case for string, int, print */
        {
            print(string)
            print(int)
            int c
            while string
            "hello"
        }$
```

Test Case 2

```
        /*test case 2 */
        {
        print(7 true a)
        while(int x = 5)
        print(x = 858)
        string a
        5 + 5
        }$
```

Test Case 3

```
        /*Long Test Case - Everything Except Boolean Declaration */
        {
        /* Int Declaration */
        int a
        int b
        a = 0
        b=0
        /* While Loop */
        while (a != 3) {
        print(a)
        while (b != 3) {
        print(b)
        b = 1 + b
        if (b == 2) {
        /* Print Statement */
        print("there␣is␣no␣spoon" /* This will do nothing */ )
        }
        }
        b = 0
        a = 1+a
        }
        }
        $
```

Test Case 4

```
        {}$
        {{{{{{}}}}}}$
        {{{{{{}}}/*comments     are     ignored */}}}}$
        { /* comments are still ignored*/int@}$
```

2 Appendix

Node.java

```java
/*
Node file
Creates Nodes to be used in the Parsers CST
*/

//import arraylist
import java.util.ArrayList;

//The Node class!
public class Node {
    String name;    //the name of the node
    Node parent;    //pointer to the nodes parent
    ArrayList<Node> children;   //list of pointers to child nodes

    //Node constructor -- creates a node and initializes its variables
    public Node(String label){
        this.name = label;
        this.parent = null;
        this.children = new ArrayList<>();
    }
}
```

CST.java

```java
/*
    Concrete Syntax tree file
    Creates CSTs to be used in Parse
    Created with help from Alan G. Labouseur, Michael Ardizzone, and Tim Smith
*/

//The CST class!
public class CST {
    Node root;  //pointer to the root node
    Node current;   //pointer to the current node
    String traversal;   //string to hold CST traversal

    //CST constructor -- creates a CST and initializes all variables
    public CST(){
        this.root = null;
        this.current = null;
        traversal = "";
    }

    //outputs the processes completed within the CST
    public void CSTLog(String output){
        System.out.println("PARSER␣-␣CST␣-␣" + output);
    }

    //adds a node to the CST
    public void addNode(String kind, String label){

        //kind - what kind of node is it?
        //label - what is the parse function?

        //create a new node to be added to the CST
        Node newNode = new Node(label);

        //check if the tree has a root node
        if(root == null){
```

```
                //update root node to new node
                root = newNode;
                newNode.parent = null;
            }
            else{
                //if there is already a root then add the newnode to child array
                newNode.parent = current;
                current.children.add(newNode);
            }

            //if the new node is not a leaf node make it the current
            if(!kind.equals("leaf")){
                current = newNode;
            }
            else{
                //output that a node was added to the tree
                CSTLog("Added␣[␣" + label + "␣]␣node");
            }
        }

        //traverses up the tree
        public void moveUp(){
            //move up to parent node if possible
            if(current.parent != null){
                current = current.parent;
            }
            else{
                // error logging
                CSTLog("ERROR!␣There␣was␣an␣error␣when␣trying␣to␣move␣up␣the␣tree...");
            }
        }

        //outputs the current programs CST if it parsed successfully
        public void outputCST(){
            expand(root, 0);
            System.out.print("\n");
            CSTLog("\n" + traversal);
        }

        public void expand(Node node, int depth){
            //space nodes out based off of the current depth
            for(int i = 0; i < depth; i++){
                traversal += "-";
            }

            //if this node is a leaf node output the name
            if(node.children.size() == 0){
                traversal += "[" + node.name + "]";
                traversal += "\n";
            }
            else{
                //this node is not a leaf node
                traversal += "<" + node.name + ">␣\n";

                //recursion!!! -- call the next child and increment the depth
                for(int j = 0; j < node.children.size(); j++){
                    expand(node.children.get(j), depth + 1);
                }
            }
        }
    }
```

3 References

3.1 Links

Below are the resources I have used to create simple, readable, and beautiful code.

- Multiple switch cases for the same result: stackoverflow
- Create lists from array: geeks4geeks
- CST and Node builds: labouseur.com
- Parser understanding and build out: https://www.labouseur.com/courses/compilers/parse.pdf
- Test cases and output checking: Labouseur.com and Gabriel Arnell
- Try-catch: w3schools
- Throwing exceptions: rollbar.com