Author: Jose Gomez

Nova Southeastern University

June 20th 2015

CISC 630

Assignment 3

**Written Assignment 3.**

**1. (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 -> SS\* -> SS+S\* -> aS+S\* -> aa+S\* -> aa+a\*

**b) Give a rightmost derivation for the string.**

S -> SS\* -> Sa\* -> SS+a\* -> Sa+a\* -> aa+a\*

**c) Give a parse tree for the string.**

\_S\_

/ | \

/ | \

S S \*

/|\ |

S S + a

| |

a a

**d) Is the grammar ambiguous or unambiguous?** Justify your answer.

The grammar is unambiguous, it does not produce multiple trees when parsed.

**e) Describe the language generated by this grammar.**

The grammar generates a language consisting of postfix addition or multiplication expressions of the letter a.

**2. (4.2.2) Repeat Exercise 4.2.1 for each of the following grammars and strings.**

**a) S -> 0 S 1 | 0 1 with string 000111**

- Leftmost derivation:

S -> 0 S 1 -> 00 S 11 -> 000111

- Rightmost derivation:

S -> 0 S 1 -> 00 S 11 -> 000111

- Parse tree:

\_S\_

/ | \

0 S 1

/|\

0 S 0

|

01

- The grammar is umanbiguous.

- The language generated by the grammar is the set of all strings of 0s followed by an equal number of 1s.

**3. (4.2.3) Design grammars for the following languages.**

**a) The set of all strings of 0s and 1s such that every 0 is immediately followed by at least one 1.**

S -> (0?1)\*

**b) The set of all strings of 0s and 1s that are palindromes; that is, the string reads the same backard as forward.**

S -> 0 S 0 | 1 S 1 | 0 | 1 | epsilon

**c) The set of all strings of 0s and 1s with an equal number of 0s and 1s.**

S -> 0 S 1 S | 1 S 0 S | epsilon

**4. (4.3.1) The following is a grammar for regular expressions over symbols a and b only, using + in place of | for union, to avoid conflict with the use of vertical bar as a metasymbol in grammars:**

rexpr -> rexpr + rterm | rterm

rterm -> rterm rfactor | rfactor

rfactor -> rfactor \* | rprimary

rprimary -> a | b

**a) Left factor this grammar.**

rexpr -> rexpr rexpr' | rterm

rexpr' -> + rterm

rterm -> rterm rterm' | rfactor

rterm' -> + rfactor

rfactor -> rfactor \* | rprimary

rprimary -> a | b

**b) Does left factoring make the grammar suitable for top-down parsing?**

This is not suitable, because the grammar contains left recursion.

**c) In addition to left factoring, eliminate left recursion from the original grammar.**

rexpr -> rterm rexpr'

rexpr' -> + rterm rexpr' | epsilon

rterm -> rfactor rterm'

rterm' -> rfactor rterm' | epsilon

rfactor -> rprimary rfactor'

rfactor' -> \* rfactor' | epsilon

rprimary -> a | b

**d) Is the resulting grammar suitable for top-down parsing?**

The grammar is suitable for top-down parsing.

**Programming Assignment 3.**

**MSS.g4:**

grammar MSS;

prog: (expr NEWLINE)+ ;

expr: DOUBLE # double

| BOOLEAN # boolean

| '(' RATOR expr\* ')' # opexpr

| '(' 'def' ID expr ')' # defvar

| '(' 'if' expr expr expr ')' # ifexpr

| ID # refvar ;

RATOR: ARITHMETIC\_OP | RELATIONAL\_OP | BOOLEAN\_OP ;

ARITHMETIC\_OP: '^' | '\*' | '/' | '+' | '-' ;

RELATIONAL\_OP: '=' | '>' | '<' ;

BOOLEAN\_OP: '&' | '|' | '!' ;

BOOLEAN: 'true' | 'false' ;

DOUBLE: ('-')? DIGIT+ ( '.' DIGIT+)? ;

ID: LETTER (LETTER|DIGIT|'\_')\* ;

NEWLINE: '\r'?'\n' ;

WS: [ \t] -> skip ;

fragment

DIGIT: [0-9] ;

fragment

LETTER: [a-zA-Z] ;

**MSS.java:**

import org.antlr.v4.runtime.ANTLRInputStream;

import org.antlr.v4.runtime.CommonTokenStream;

import org.antlr.v4.runtime.tree.ParseTree;

public class MSS {

public static void main(String[] args) throws Exception {

ANTLRInputStream input = new ANTLRInputStream(System.in);

MSSLexer lexer = new MSSLexer(input);

CommonTokenStream tokens = new CommonTokenStream(lexer);

MSSParser parser = new MSSParser(tokens);

ParseTree tree = parser.prog();

SchemeVisitor calc = new SchemeVisitor();

calc.visit(tree);

}

}

**SchemeVisitor.java:**

import java.util.HashMap;

import java.util.Map;

public class SchemeVisitor extends MSSBaseVisitor<Val> {

private Map<String, Val> variables;

public SchemeVisitor()

{

variables = new HashMap<String, Val>();

}

@Override public Val visitDefvar(MSSParser.DefvarContext ctx) {

String id = ctx.ID().getText();

Val value = visit(ctx.expr());

variables.put(id, value);

return value;

}

@Override public Val visitRefvar(MSSParser.RefvarContext ctx) {

String id = ctx.ID().getText();

Val value = variables.get(id);

if (value == null)

throw new RuntimeException(id + " is not defined.");

return value;

}

@Override public Val visitProg(MSSParser.ProgContext ctx) {

Val result = null;

for(MSSParser.ExprContext ectx : ctx.expr()) {

result = visit(ectx);

}

if (result != null)

System.out.println(result.getValue());

return result;

}

@Override public Val visitBoolean(MSSParser.BooleanContext ctx) {

Boolean value = new Boolean(ctx.BOOLEAN().getText());

return new Val(value);

}

@Override public Val visitDouble(MSSParser.DoubleContext ctx) {

Double value = new Double(ctx.DOUBLE().getText());

return new Val(value);

}

@Override public Val visitIfexpr(MSSParser.IfexprContext ctx) {

if (visit(ctx.expr(0)).getBoolean()) {

return visit(ctx.expr(1));

} else {

return visit(ctx.expr(2));

}

}

@Override public Val visitOpexpr(MSSParser.OpexprContext ctx) {

String op = ctx.RATOR().getText();

switch(op) {

case "+":

{

Double result = 0.0;

for (MSSParser.ExprContext expr : ctx.expr()) {

result = result + visit(expr).getDouble();

}

return new Val(result);

}

case "\*":

{

Double result = 1.0;

for (MSSParser.ExprContext expr : ctx.expr()) {

result = result \* visit(expr).getDouble();

}

return new Val(result);

}

case "^":

{

Double result = null;

if (ctx.expr().size() < 2) {

result = 1.0;

}

for (MSSParser.ExprContext expr : ctx.expr()) {

if (result == null)

{

result = visit(expr).getDouble();

continue;

}

result = Math.pow(result, visit(expr).getDouble());

}

return new Val(result);

}

case "/":

{

if (ctx.expr().isEmpty())

throw new RuntimeException("illegal: (/ )");

Double result = null;

if (ctx.expr().size() < 2) {

result = 1.0;

}

for (MSSParser.ExprContext expr : ctx.expr()) {

if (result == null)

{

result = visit(expr).getDouble();

continue;

}

result = result / visit(expr).getDouble();

}

return new Val(result);

}

case "-":

{

if (ctx.expr().isEmpty())

throw new RuntimeException("illegal: (- )");

Double result = null;

if (ctx.expr().size() < 2) {

result = 0.0;

}

for (MSSParser.ExprContext expr : ctx.expr()) {

if (result == null)

{

result = visit(expr).getDouble();

continue;

}

result = result - visit(expr).getDouble();

}

return new Val(result);

}

case "&":

{

Boolean result = true;

for (MSSParser.ExprContext expr : ctx.expr()) {

result = result && visit(expr).getBoolean();

if (!result) break;

}

return new Val(result);

}

case "|":

{

Boolean result = false;

for (MSSParser.ExprContext expr : ctx.expr()) {

result = visit(expr).getBoolean();

if (result) break;

}

return new Val(result);

}

case "!":

{

if (ctx.expr().size() != 1)

throw new RuntimeException("illegal: ! operator must have exactly 1 expr argument.");

Boolean result = visit(ctx.expr(0)).getBoolean();

return new Val(!result);

}

case "=":

{

Boolean result = true;

Object pvalue = null;

for (MSSParser.ExprContext expr : ctx.expr()) {

if (pvalue == null)

{

pvalue = visit(expr).getValue();

continue;

}

Object cvalue = visit(expr).getValue();

result = result && (pvalue.equals(cvalue));

pvalue = cvalue;

if (!result) break;

}

return new Val(result);

}

case ">":

{

Boolean result = true;

Double pvalue = null;

for (MSSParser.ExprContext expr : ctx.expr()) {

if (pvalue == null)

{

pvalue = visit(expr).getDouble();

continue;

}

Double cvalue = visit(expr).getDouble();

result = result && (pvalue > cvalue);

if (!result) break;

}

return new Val(result);

}

case "<":

{

Boolean result = true;

Double pvalue = null;

for (MSSParser.ExprContext expr : ctx.expr()) {

if (pvalue == null)

{

pvalue = visit(expr).getDouble();

continue;

}

Double cvalue = visit(expr).getDouble();

result = result && (pvalue < cvalue);

if (!result) break;

}

return new Val(result);

}

default:

throw new RuntimeException("illegal operator: " + op);

}

}

}

**References**

Aho, A., Lam, M., Sethi, R., & Ullman, J. (2007). *Compilers Principles, Techniques, & Tools* (2nd ed.). Boston, MA: Addison-Wesley.