Author: Jose Gomez

Nova Southeastern University

July 31st 2015

CISC 630

Assignment 7

**Programming Exercise**

**Newly implemented features.**

**- Semantics for multi-argument function definition and calls.**

(call (fun n1 n2 (\* n1 n2)) 4 5)

=> 20.0

**- Semantics for multi-argument let declarations.**

(let [n1 5

n2 6]

(\* n1 n2))

=> 30.0

**- Semantics for naming and defining new functions in a single expression.**

(defun volume s1 s2 s3 (\* s1 s2 s3))

=> <fun>

(call volume 2 3 4)

=> 24.0

**- Semantics for new constructs for raising and handling exceptions.**

(begin (raise SillyError) (+ 2 3))

=> <SillyError>

(try (raise SillyError1)

SillyError2 (+ 2 3))

=> <SillyError1>

(try (raise SillyError1)

SillyError1 (+ 2 3))

=> 5.0

**- Add a REPL with exception handling and `(quit)` expression to exit the REPL.**

(quit)

=> <quit>

(+ 2 2)

(quit)

=> 4.0

=> <quit>

**Scheme.g4:**

grammar Scheme;

prog: expr+

;

expr: DOUBLE # Double

| BOOLEAN # Boolean

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

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

| '(' 'if' expr expr expr ')' # IfExpr

| '(' 'print' expr ')' # PrintExpr

| '(' 'while' expr expr ')' # WhileExpr

| '(' 'begin' expr+ ')' # BlockExpr

| '(' 'let' bind expr ')' # LetExpr

| '(' 'fun' ID+ expr ')' # FuncExpr

| '(' 'defun' ID ID+ expr ')' # DefFuncExpr

| '(' 'call' expr expr+ ')' # CallExpr

| '(' 'try' expr (ID expr)+ ')' # TryExpr

| '(' 'raise' ID ')' # RaiseExpr

| '(' 'quit' ')' # QuitExpr

| ID # RefExpr

;

bind: '[' (ID expr)+ ']'

;

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

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

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

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

BOOLEAN: 'true' | 'false' ;

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

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

WS: [ \t\r\n] -> skip ;

fragment

DIGIT: [0-9] ;

fragment

LETTER: [a-zA-Z] ;

**Environment.java:**

import java.util.HashMap;

public class Environment {

private Environment parent;

private HashMap<String, Val> symbols;

public Environment(Environment parent) {

this.parent = parent;

this.symbols = new HashMap<String, Val>();

}

public Environment getParent() {

return this.parent;

}

public void set(String id, Val value) {

symbols.put(id, value);

}

public Val get(String id) {

Val result = symbols.get(id);

if (result == null

&& parent != null) {

result = parent.get(id);

}

return result;

}

}

**Function.java:**

import java.util.List;

public class Function {

private List<String> args;

private SchemeParser.ExprContext expr;

public Function(List<String> args, SchemeParser.ExprContext expr) {

this.args = args;

this.expr = expr;

}

public List<String> args() {

return this.args;

}

public SchemeParser.ExprContext expr() {

return this.expr;

}

public String toString() {

return "<fun>";

}

}

**Val.java:**

public class Val {

private Object value;

public Val(Object value) {

this.value = value;

}

public <T> T get(Class<T> t) {

Class<?> vt = value.getClass();

if (!t.isAssignableFrom(vt))

throw new RuntimeException(

"Type mismatch: expected "

+ t.getName()

+ " instead of "

+ vt.getName());

return t.cast(value);

}

public Object get() {

return value;

}

}

**SchemeException.java:**

public class SchemeException extends RuntimeException {

public SchemeException(String message) {

super(message);

}

public String toString() {

return "<" + getMessage() + ">";

}

}

**SchemeQuitException.java:**

public class SchemeQuitException extends RuntimeException {

}

**Scheme.java:**

import org.antlr.v4.runtime.ANTLRInputStream;

import org.antlr.v4.runtime.CommonTokenStream;

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

import java.io.FileInputStream;

import java.io.InputStream;

import java.util.Scanner;

import java.util.NoSuchElementException;

import java.util.regex.Pattern;

public class Scheme {

private static Scanner getScanner() {

Scanner scanner = new Scanner(System.in);

Pattern delim = Pattern.compile("( \r?\n)|(\r?\n){2}");

scanner.useDelimiter(delim);

return scanner;

}

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

ValVisitor eval = new ValVisitor();

Scanner scanner = getScanner();

while (!eval.isDone()) {

try {

String src = scanner.next();

ANTLRInputStream input = new ANTLRInputStream(src);

SchemeLexer lexer = new SchemeLexer(input);

CommonTokenStream tokens = new CommonTokenStream(lexer);

SchemeParser parser = new SchemeParser(tokens);

ParseTree tree = parser.prog();

eval.visit(tree);

}

catch (NoSuchElementException ee) {

break;

}

}

}

}

**ValVisitor.java:**

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.Stack;

public class ValVisitor extends SchemeBaseVisitor<Val> {

private Environment env;

private Boolean done;

public ValVisitor() {

done = false;

env = new Environment(null);

}

public Boolean isDone() {

return done;

}

private void output(Val result) {

if (result != null)

System.out.println("=> " + result.get());

}

@Override public Val visitDefExpr(SchemeParser.DefExprContext ctx) {

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

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

env.set(id, value);

return value;

}

@Override public Val visitRefExpr(SchemeParser.RefExprContext ctx) {

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

Val value = env.get(id);

if (value == null)

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

return value;

}

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

Val result = null;

try {

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

result = visit(ectx);

}

}

catch (SchemeException se) {

result = new Val(se);

output(result);

}

catch (SchemeQuitException qe) {

output(result);

output(new Val("<quit>"));

}

catch (Exception ee) {

ee.printStackTrace();

}

return result;

}

@Override public Val visitQuitExpr(SchemeParser.QuitExprContext ctx) {

done = true;

throw new SchemeQuitException();

}

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

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

return new Val(value);

}

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

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

return new Val(value);

}

@Override public Val visitIfExpr(SchemeParser.IfExprContext ctx) {

if (visit(ctx.expr(0)).get(Boolean.class)) {

return visit(ctx.expr(1));

} else {

return visit(ctx.expr(2));

}

}

@Override public Val visitPrintExpr(SchemeParser.PrintExprContext ctx) {

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

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

return result;

}

@Override public Val visitWhileExpr(SchemeParser.WhileExprContext ctx) {

Val result = new Val(0D);

while(visit(ctx.expr(0)).get(Boolean.class)) {

result = visit(ctx.expr(1));

}

return result;

}

@Override public Val visitBlockExpr(SchemeParser.BlockExprContext ctx) {

Val result = null;

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

result = visit(ectx);

}

return result;

}

@Override public Val visitLetExpr(SchemeParser.LetExprContext ctx) {

env = new Environment(env);

visit(ctx.bind());

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

env = env.getParent();

return result;

}

@Override public Val visitBind(SchemeParser.BindContext ctx) {

for(int i = 0; i < ctx.ID().size(); i++) {

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

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

env.set(id, value);

}

return null;

}

@Override public Val visitFuncExpr(SchemeParser.FuncExprContext ctx) {

List<String> args = new ArrayList<String>();

for(int i = 0; i < ctx.ID().size(); i++) {

args.add(ctx.ID(i).getText());

}

Function func = new Function(args, ctx.expr());

return new Val(func);

}

@Override public Val visitDefFuncExpr(SchemeParser.DefFuncExprContext ctx) {

String id = ctx.ID(0).getText();

List<String> args = new ArrayList<String>();

for(int i = 1; i < ctx.ID().size(); i++) {

args.add(ctx.ID(i).getText());

}

Function func = new Function(args, ctx.expr());

Val value = new Val(func);

env.set(id, value);

return value;

}

@Override public Val visitCallExpr(SchemeParser.CallExprContext ctx) {

Function funCtx = visit(ctx.expr(0)).get(Function.class);

env = new Environment(env);

for(int i = 0; i < funCtx.args().size(); i++) {

String id = funCtx.args().get(i);

Val arg = visit(ctx.expr(i+1));

env.set(id, arg);

}

Val result = visit(funCtx.expr());

env = env.getParent();

return result;

}

@Override public Val visitTryExpr(SchemeParser.TryExprContext ctx) {

try {

return visit(ctx.expr(0));

}

catch (SchemeException se) {

for(int i = 0; i < ctx.ID().size(); i++) {

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

if (id.equals(se.getMessage())) {

return visit(ctx.expr(i + 1));

}

}

throw se;

}

}

@Override public Val visitRaiseExpr(SchemeParser.RaiseExprContext ctx) {

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

throw new SchemeException(id);

}

@Override public Val visitOpExpr(SchemeParser.OpExprContext ctx) {

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

switch(op) {

case "+":

{

Double result = 0.0;

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

result = result + visit(expr).get(Double.class);

}

return new Val(result);

}

case "\*":

{

Double result = 1.0;

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

result = result \* visit(expr).get(Double.class);

}

return new Val(result);

}

case "^":

{

Double result = null;

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

result = 1.0;

}

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

if (result == null) {

result = visit(expr).get(Double.class);

continue;

}

result = Math.pow(result, visit(expr).get(Double.class));

}

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 (SchemeParser.ExprContext expr : ctx.expr()) {

if (result == null) {

result = visit(expr).get(Double.class);

continue;

}

result = result / visit(expr).get(Double.class);

}

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 (SchemeParser.ExprContext expr : ctx.expr()) {

if (result == null) {

result = visit(expr).get(Double.class);

continue;

}

result = result - visit(expr).get(Double.class);

}

return new Val(result);

}

case "&":

{

Boolean result = true;

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

result = result && visit(expr).get(Boolean.class);

if (!result) break;

}

return new Val(result);

}

case "|":

{

Boolean result = false;

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

result = visit(expr).get(Boolean.class);

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)).get(Boolean.class);

return new Val(!result);

}

case "=":

{

Boolean result = true;

Object pvalue = null;

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

if (pvalue == null) {

pvalue = visit(expr).get();

continue;

}

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

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

pvalue = cvalue;

if (!result) break;

}

return new Val(result);

}

case ">":

{

Boolean result = true;

Double pvalue = null;

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

if (pvalue == null) {

pvalue = visit(expr).get(Double.class);

continue;

}

Double cvalue = visit(expr).get(Double.class);

result = result && (pvalue > cvalue);

if (!result) break;

}

return new Val(result);

}

case "<":

{

Boolean result = true;

Double pvalue = null;

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

if (pvalue == null) {

pvalue = visit(expr).get(Double.class);

continue;

}

Double cvalue = visit(expr).get(Double.class);

result = result && (pvalue < cvalue);

if (!result) break;

}

return new Val(result);

}

default:

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

}

}

}