2021-01

프로그래밍언어론(001분반)

Lab05

보고서

**학과:소프트웨어융합전공**

**학번: 1912339**

**이름: 신지우**

**제출일: 2021/5/23**

1.구현한 프로그램에 대한 설명

AST.java는 빈칸인 부분이 다 이전 과제였던 부분이어서 다시 채웠고 설명은 생략하겠습니다.

함수 정의 <function> -> fun <type> id(<params>) <stmt>

함수 리턴 <stmt> -> return <expr>;

함수 호출 | id(<expr>{,<expr>});

<factor> -> id(<expr>{,<expr>});

Parser.java에서 함수 정의, 함수 리턴은 이미 구현되어있어서 함수 호출 부분을 구현하였다. 총 3가지 함수를 건드렸는데, 현재 문장이 어느 문장인지 구분하는 stmt()와 함수호출 파싱을 하는call()과 어느 요소인지 구분하는 factor()이다. stmt()를 수정한 이유는 함수 호출의 앞머리가 id인데, 이를 대입문의 앞머리 id와 구분할 필요가 있기 때문이다. stmt()의 case ID에서 Identifier 타입의 id에 토큰(식별자)을 미리 저장해주고, 그 다음 토큰이 ‘(’이면 함수 호출인 것으로 예상하고 call()로 값을 받아오게 했다. ‘(‘이 아니라면, 대입문(‘=’)인 것으로 예상하고 assignment()로 값을 받아오게 했다. 그리고 본래에는 assignment는 인수를 받지 않았는데, id를 넘겨주기 위해 Identifier타입 변수를 인수로 받도록 했다.

call(Identifier id)에서는 ‘(‘ 토큰을 match시켜 다음 토큰으로 넘겨준 후, 함수이름 id와 소괄호 사이에 들은 인수들 arguments()를 인수로 넘겨 Call타입 변수 c를 정의했다. arguments()에서는 토큰이 ‘)’가 아닐 동안, 계속해서 expr의 arraylist인 Exprs 타입의 es에 저장하여 es를 리턴한다. 이후 call()로 돌아와서, ‘)’토큰을 match, ‘;’토큰을 match 후, Call 타입 변수 c를 리턴한다.

factor()에서는 stmt()와 마찬가지로 함수호출의 id와 대입문의 id를 구분할 필요가 있기 때문에 case ID에서 if문을 통해 식별자를 받고난 다음 토큰이 ‘(‘인지 확인하고 함수호출로 파싱해주는 경우를 추가하였다.

Sint.java에서는 총 3가지를 구현하였는데, ‘함수 호출’ 부분의 리턴값이 없는 함수를 계산하는 State Eval(Call c, State state)와, newFrame(), deleteFrame()을 구현하였다.

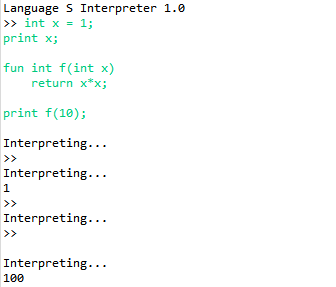
Eval(Call, State)은 리턴문이 없다는 점 하나 빼고는 리턴값이 있는 함수를 계산하는 V(Call c, State state)와 같다. 따라서 똑같이, Call 변수의 fid를 통해 스택에서 실행코드의 ast를 찾고, newFrame()을 통해 스택에 새로운 프레임을 구성하고, 함수의 AST(f.stmt)를 이용해 함수 내용을 계산한 후, deleteFrame()을 통해 프레임을 제거한다. 그리고 State 변수 s를 리턴한다.

newFrame(State, Call, Function)은 state 스택에 Function 정의의 매개변수들(params)을 넣어준 후, Call에서 입력받은 인자들을 계산해서, state에 저장된 매개변수들에 대입해주는 과정이다. 따라서, Value 배열인 val[]을 매개변수 개수 크기로 선언 후, Call 인자의 계산값을 차례대로 넣어준다. 그 다음, 매개변수를 스택에 allocate()해준다. 이후, 스택에 저장된 params에 val[]값을 차례대로 저장한 후, state를 리턴한다.

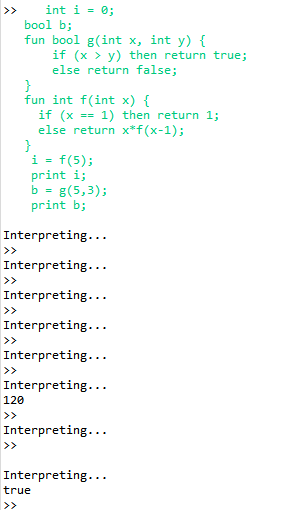
deleteFrame()은 newFrame()에서 추가한 함수 프레임을 삭제하는 과정이다. Decls 타입인 f.params와 현재 상태 스택인 state를 free()의 인수로 넘겨 주고, state를 리턴한다. 그럼 free()에서 params의 decl 개수만큼 스택을 pop()하므로 state가 정리된다.

2. 실행 결과 캡처 사진

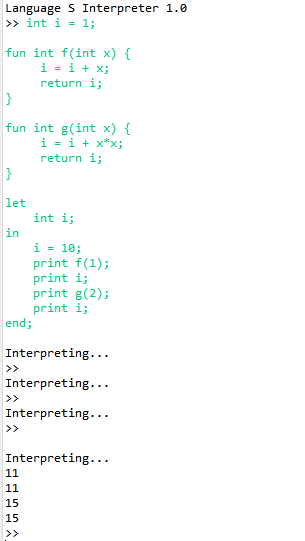
hi8.s



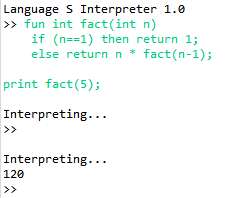
9.s



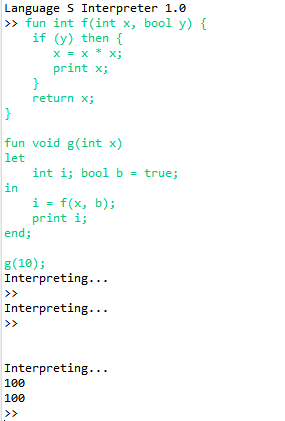
hi10.s



11.s



12.s



3. 프로그램 소스 코드 (복사-붙여넣기)

// AST.java

// AST for S

import java.util.\*;

abstract class Command {

// Command = Decl | Function | Stmt

Type type =Type.UNDEF;

}

class Decls extends ArrayList<Decl> {

// Decls = Decl\*

Decls() { super(); };

Decls(Decl d) {

this.add(d);

}

}

class Decl extends Command {

// Decl = Type type; Identifier id

Identifier id;

Expr expr = null;

int arraysize = 0;

Decl (String s, Type t) {

id = new Identifier(s); type = t;

} // declaration

Decl (String s, Type t, int n) {

id = new Identifier(s); type = t; arraysize = n;

} // array declaration

Decl (String s, Type t, Expr e) {

id = new Identifier(s); type = t; expr = e;

} // declaration

}

class Functions extends ArrayList<Function> {

// Functions = Function\*

}

class Function extends Command {

// Function = Type type; Identifier id; Decls params; Stmt stmt

Identifier id;

Decls params;

Stmt stmt;

Function(String s, Type t) {

id = new Identifier(s); type = t; params = null; stmt = null;

}

public String toString ( ) {

return id.toString()+params.toString();

}

}

class Type {

// Type = int | bool | string | fun | array | void

final static Type INT = new Type("int");

final static Type BOOL = new Type("bool");

final static Type STRING = new Type("string");

final static Type VOID = new Type("void");

final static Type FUN = new Type("fun");

final static Type ARRAY = new Type("array");

final static Type UNDEF = new Type("undef");

final static Type ERROR = new Type("error");

protected String id;

protected Type(String s) { id = s; }

public String toString ( ) { return id; }

}

class ProtoType extends Type {

// defines the type of a function and its parameters

Type result;

Decls params;

ProtoType (Type t, Decls ds) {

super(t.id);

result = t;

params = ds;

}

}

abstract class Stmt extends Command {

// Stmt = Empty | Stmts | Assignment | If | While | Let | Read | Print

}

class Empty extends Stmt {

}

class Stmts extends Stmt {

// Stmts = Stmt\*

public ArrayList<Stmt> stmts = new ArrayList<Stmt>();

Stmts() {

super();

}

Stmts(Stmt s) {

stmts.add(s);

}

}

class Assignment extends Stmt {

// Assignment = Identifier id; Expr expr

Identifier id;

Array ar = null;

Expr expr;

Assignment (Identifier t, Expr e) {

id = t;

expr = e;

}

Assignment (Array a, Expr e) { //

ar = a;

expr = e;

}

}

class If extends Stmt {

// If = Expr expr; Stmt stmt1, stmt2;

Expr expr;

Stmt stmt1, stmt2;

If (Expr t, Stmt tp) {

expr = t; stmt1 = tp; stmt2 = new Empty( );

}

If (Expr t, Stmt tp, Stmt ep) {

expr = t; stmt1 = tp; stmt2 = ep;

}

}

class While extends Stmt {

// While = Expr expr; Stmt stmt;

Expr expr;

Stmt stmt;

While (Expr t, Stmt b) {

expr = t; stmt = b;

}

}

class Let extends Stmt {

// Let = Decls decls; Functions funs; Stmts stmts;

Decls decls;

Functions funs;

Stmts stmts;

Let(Decls ds, Functions fs, Stmts ss) {

decls = ds;

funs = fs;

stmts = ss;

}

}

class Read extends Stmt {

// Read = Identifier id

Identifier id;

Read (Identifier v) {

id = v;

}

}

class Print extends Stmt {

// Print = Expr expr

Expr expr;

Print (Expr e) {

expr = e;

}

}

class Return extends Stmt {

Identifier fid;

Expr expr;

Return (String s, Expr e) {

fid = new Identifier(s);

expr = e;

}

}

class Try extends Stmt {

// Try = Identifier id; Stmt stmt1; Stmt stmt2;

Identifier eid;

Stmt stmt1;

Stmt stmt2;

Try(Identifier id, Stmt s1, Stmt s2) {

eid = id;

stmt1 = s1;

stmt2 = s2;

}

}

class Raise extends Stmt {

Identifier eid;

Raise(Identifier id) {

eid = id;

}

}

class Exprs extends ArrayList<Expr> {

// Exprs = Expr\*

}

abstract class Expr extends Stmt {

// Expr = Identifier | Value | Binary | Unary | Call

}

class Call extends Expr {

Identifier fid;

Exprs args;

Call(Identifier id, Exprs a) {

fid = id;

args = a;

}

}

class Identifier extends Expr {

// Identifier = String id

private String id;

Identifier(String s) { id = s; }

public String toString( ) { return id; }

public boolean equals (Object obj) {

String s = ((Identifier) obj).id;

return id.equals(s);

}

}

class Array extends Expr {

// Array = Identifier id; Expr expr

Identifier id;

Expr expr = null;

Array(Identifier s, Expr e) {id = s; expr = e;}

public String toString( ) { return id.toString(); }

public boolean equals (Object obj) {

String s = ((Array) obj).id.toString();

return id.equals(s);

}

}

class Value extends Expr {

// Value = int | bool | string | array | function

protected boolean undef = true;

Object value = null; // Type type;

Value(Type t) {

type = t;

if (type == Type.INT) value = new Integer(0);

if (type == Type.BOOL) value = new Boolean(false);

if (type == Type.STRING) value = "";

undef = false;

}

Value(Object v) {

if (v instanceof Integer) type = Type.INT;

if (v instanceof Boolean) type = Type.BOOL;

if (v instanceof String) type = Type.STRING;

if (v instanceof Function) type = Type.FUN;

if (v instanceof Value[]) type = Type.ARRAY;

value = v; undef = false;

}

Object value() { return value; }

int intValue( ) {

if (value instanceof Integer)

return ((Integer) value).intValue();

else return 0;

}

boolean boolValue( ) {

if (value instanceof Boolean)

return ((Boolean) value).booleanValue();

else return false;

}

String stringValue ( ) {

if (value instanceof String)

return (String) value;

else return "";

}

Function funValue ( ) {

if (value instanceof Function)

return (Function) value;

else return null;

}

Value[] arrValue ( ) {

if (value instanceof Value[])

return (Value[]) value;

else return null;

}

Type type ( ) { return type; }

public String toString( ) {

//if (undef) return "undef";

if (type == Type.INT) return "" + intValue();

if (type == Type.BOOL) return "" + boolValue();

if (type == Type.STRING) return "" + stringValue();

if (type == Type.FUN) return "" + funValue();

if (type == Type.ARRAY) return "" + arrValue();

return "undef";

}

}

class Binary extends Expr {

// Binary = Operator op; Expr expr1; Expr expr2;

Operator op;

Expr expr1, expr2;

Binary (Operator o, Expr e1, Expr e2) {

op = o; expr1 = e1; expr2 = e2;

} // binary

}

class Unary extends Expr {

// Unary = Operator op; Expr expr

Operator op;

Expr expr;

Unary (Operator o, Expr e) {

op = o; //(o.val == "-") ? new Operator("neg"): o;

expr = e;

} // unary

}

class Operator {

String val;

Operator (String s) {

val = s;

}

public String toString( ) {

return val;

}

public boolean equals(Object obj) {

return val.equals(obj);

}

}

// Parser.java

// Parser for language S

public class Parser {

Token token; // current token

Lexer lexer;

String funId = "";

public Parser(Lexer scan) {

lexer = scan;

token = lexer.getToken(); // get the first token

}

private String match(Token t) {

String value = token.value();

if (token == t)

token = lexer.getToken();

else

error(t);

return value;

}

private void error(Token tok) {

System.err.println("Syntax error: " + tok + " --> " + token);

token=lexer.getToken();

}

private void error(String tok) {

System.err.println("Syntax error: " + tok + " --> " + token);

token=lexer.getToken();

}

public Command command() {

// <command> -> <decl> | <function> | <stmt>

if (isType()) {

Decl d = decl();

return d;

}

if (token == Token.FUN) {

Function f = function();

return f;

}

if (token != Token.EOF) {

Stmt s = stmt();

return s;

}

return null;

}

private Decl decl() {

// <decl> -> <type> id [=<expr>]; | <type> id[<n>];

Type t = type();

String id = match(Token.ID);

int n=0;

Decl d = null;

if(token==Token.LBRACKET) {

match(Token.LBRACKET);

n = (int) literal().value;

match(Token.RBRACKET);

}

if (n!=0)

d = new Decl(id, t, n);

else if (token == Token.ASSIGN) {

match(Token.ASSIGN);

Expr e = expr();

d = new Decl(id, t, e);

} else

d = new Decl(id, t);

match(Token.SEMICOLON);

return d;

}

private Decls decls () {

// <decls> -> {<decl>}

Decls ds = new Decls ();

while (isType()) {

Decl d = decl();

ds.add(d);

}

return ds;

}

private Function function() {

// <function> -> fun <type> id(<params>) <stmt>

match(Token.FUN);

Type t = type();

String str = match(Token.ID);

funId = str;

Function f = new Function(str, t);

match(Token.LPAREN);

if (token != Token.RPAREN)

f.params = params();

match(Token.RPAREN);

Stmt s = stmt();

f.stmt = s;

return f;

}

private Decls params() {

Decls params = new Decls();

Type t = type();

String id = match(Token.ID);

params.add(new Decl(id, t));

while (token == Token.COMMA) {

match(Token.COMMA);

t = type();

id = match(Token.ID);

params.add(new Decl(id, t));

}

return params;

}

private Type type () {

// <type> -> int | bool | void | string

Type t = null;

switch (token) {

case INT:

t = Type.INT; break;

case BOOL:

t = Type.BOOL; break;

case VOID:

t = Type.VOID; break;

case STRING:

t = Type.STRING; break;

default:

error("int | bool | void | string");

}

match(token);

return t;

}

private Stmt stmt() {

// <stmt> -> <block> | <assignment> | <ifStmt> | <whileStmt> | ...

Stmt s = new Empty();

switch (token) {

case SEMICOLON:

match(token.SEMICOLON); return s;

case LBRACE:

match(Token.LBRACE);

s = stmts();

match(Token.RBRACE);

return s;

case IF: // if statement

s = ifStmt(); return s;

case WHILE: // while statement

s = whileStmt(); return s;

case ID: // assignment

Identifier id = new Identifier(match(Token.ID));

if(token == Token.LPAREN) s = call(id);

else s = assignment(id);

return s;

case LET: // let statement

s = letStmt(); return s;

case READ: // read statement

s = readStmt(); return s;

case PRINT: // print statement

s = printStmt(); return s;

case RETURN: // return statement

s = returnStmt(); return s;

default:

error("Illegal stmt"); return null;

}

}

private Stmts stmts () {

// <block> -> {<stmt>}

Stmts ss = new Stmts();

while((token != Token.RBRACE) && (token != Token.END))

ss.stmts.add(stmt());

return ss;

}

private Let letStmt () {

// <letStmt> -> let <decls> in <block> end

match(Token.LET);

Decls ds = decls();

match(Token.IN);

Stmts ss = stmts();

match(Token.END);

match(Token.SEMICOLON);

return new Let(ds, null, ss);

}

private Read readStmt() {

// <readStmt> -> read id;

// parse read statement

match(Token.READ);

Identifier id = new Identifier(match(Token.ID));

match(Token.SEMICOLON);

return new Read(id);

}

private Print printStmt() {

// <printStmt> -> print <expr>;

// parse print statement

match(Token.PRINT);

Expr e = expr();

match(Token.SEMICOLON);

return new Print(e);

}

private Return returnStmt() {

// <returnStmt> -> return <expr>;

match(Token.RETURN);

Expr e = expr();

match(Token.SEMICOLON);

return new Return(funId, e);

}

private Stmt assignment(Identifier id) {

// <assignment> -> id = <expr>; | id[<expr>] = <expr>;

Expr n = null;

Array a = null;

if(token==Token.LBRACKET) {

match(Token.LBRACKET);

n = expr();

match(Token.RBRACKET);

a = new Array(id, n);

}

match(Token.ASSIGN);

Expr e = expr();

match(Token.SEMICOLON);

if(a != null)

return new Assignment(a,e);

else

return new Assignment(id, e);

}

private Call call(Identifier id) {

// <call> -> id(<expr>{,<expr>});

//

// parse function call

match(Token.LPAREN);

Call c = new Call(id, arguments());

match(Token.RPAREN);

match(Token.SEMICOLON);

return c;

}

private If ifStmt () {

// <ifStmt> -> if (<expr>) then <stmt> [else <stmt>]

match(Token.IF);

match(Token.LPAREN);

Expr e = expr();

match(Token.RPAREN);

match(Token.THEN);

Stmt s1 = stmt();

Stmt s2 = new Empty();

if (token == Token.ELSE){

match(Token.ELSE);

s2 = stmt();

}

return new If(e, s1, s2);

}

private While whileStmt () {

// <whileStmt> -> while (<expr>) <stmt>

// parse while statement

match(Token.WHILE);

match(Token.LPAREN);

Expr e = expr();

match(Token.RPAREN);

Stmt s = stmt();

return new While(e,s);

}

private Expr expr () {

// <expr> -> <bexp> {& <bexp> | '|'<bexp>} | !<expr> | true | false

switch (token) {

case NOT:

Operator op = new Operator(match(token));

Expr e = expr();

return new Unary(op, e);

case TRUE:

match(Token.TRUE);

return new Value(true);

case FALSE:

match(Token.FALSE);

return new Value(false);

}

Expr e = bexp();

//

// parse logical operations

while (token == Token.AND || token == Token.OR) {

Operator op = new Operator(match(token));

Expr b = bexp();

e = new Binary(op, e, b);

}

return e;

}

private Expr bexp() {

// <bexp> -> <aexp> [ (< | <= | > | >= | == | !=) <aexp> ]

Expr e = aexp();

//

// parse relational operations

if(token == Token.EQUAL || token == Token.LTEQ || token == Token.LT || token == Token.GTEQ || token == Token.GT|| token == Token.NOTEQ) {

Operator op = new Operator(match(token));

Expr a = aexp();

e = new Binary(op,e,a);

}

return e;

}

private Expr aexp () {

// <aexp> -> <term> { + <term> | - <term> }

Expr e = term();

while (token == Token.PLUS || token == Token.MINUS) {

Operator op = new Operator(match(token));

Expr t = term();

e = new Binary(op, e, t);

}

return e;

}

private Expr term () {

// <term> -> <factor> { \* <factor> | / <factor>}

Expr t = factor();

while (token == Token.MULTIPLY || token == Token.DIVIDE) {

Operator op = new Operator(match(token));

Expr f = factor();

t = new Binary(op, t, f);

}

return t;

}

private Expr factor() {

// <factor> -> [-](id | <call> | id[<expr>] | literal | '('<aexp> ')')

Operator op = null;

if (token == Token.MINUS)

op = new Operator(match(Token.MINUS));

Expr e = null;

switch(token) {

case ID:

Identifier v = new Identifier(match(Token.ID));

if (token == Token.LPAREN) { // function call

match(Token.LPAREN);

Call c = new Call(v,arguments());

match(Token.RPAREN);

e = c;

}

else if(token == Token.LBRACKET) {

match(Token.LBRACKET);

Expr n = expr();

match(Token.RBRACKET);

Array a = new Array(v, n);

e = a;

}

else e = v;

break;

case NUMBER: case STRLITERAL:

e = literal();

break;

case LPAREN:

match(Token.LPAREN);

e = aexp();

match(Token.RPAREN);

break;

default:

error("Identifier | Literal");

}

if (op != null)

return new Unary(op, e);

else return e;

}

private Exprs arguments() {

// arguments -> [ <expr> {, <expr> } ]

Exprs es = new Exprs();

while (token != Token.RPAREN) {

es.add(expr());

if (token == Token.COMMA)

match(Token.COMMA);

else if (token != Token.RPAREN)

error("Exprs");

}

return es;

}

private Value literal( ) {

String s = null;

switch (token) {

case NUMBER:

s = match(Token.NUMBER);

return new Value(Integer.parseInt(s));

case STRLITERAL:

s = match(Token.STRLITERAL);

return new Value(s);

}

throw new IllegalArgumentException( "no literal");

}

private boolean isType( ) {

switch(token) {

case INT: case BOOL: case STRING:

return true;

default:

return false;

}

}

public static void main(String args[]) {

Parser parser;

if (args.length == 0) {

System.out.println("Begin parsing... ");

System.out.print(">> ");

Lexer.interactive = true;

parser = new Parser(new Lexer());

do {

if (parser.token == Token.EOF) {

parser.token = parser.lexer.getToken();

}

Command command = null;

try {

command = parser.command();

} catch (Exception e) {

System.err.println(e);

}

System.out.print("\n>> ");

} while(true);

}

else {

System.out.println("Begin parsing... " + args[0]);

parser = new Parser(new Lexer(args[0]));

Command command = null;

do {

try {

command = parser.command();

} catch (Exception e) {

System.err.println(e);

}

} while (command != null);

}

} //main

} // Parser

// Sint.java

// Interpreter for S

import java.util.Iterator;

import java.util.Scanner;

public class Sint {

static Scanner sc = new Scanner(System.in);

static State state = new State();

State Eval(Command c, State state) {

if (c instanceof Decl) {

Decls decls = new Decls();

decls.add((Decl) c);

return allocate(decls, state);

}

if (c instanceof Function) {

Function f = (Function) c;

state.push(f.id, new Value(f));

return state;

}

if (c instanceof Stmt)

return Eval((Stmt) c, state);

throw new IllegalArgumentException("no command");

}

State Eval(Stmt s, State state) {

if (s instanceof Empty)

return Eval((Empty)s, state);

if (s instanceof Assignment)

return Eval((Assignment)s, state);

if (s instanceof If)

return Eval((If)s, state);

if (s instanceof While)

return Eval((While)s, state);

if (s instanceof Stmts)

return Eval((Stmts)s, state);

if (s instanceof Let)

return Eval((Let)s, state);

if (s instanceof Read)

return Eval((Read)s, state);

if (s instanceof Print)

return Eval((Print)s, state);

if (s instanceof Call) {

//Call c = (Call) s;

//Value v = state.get(c.fid);

//if(v.funValue().type == Type.VOID)

return Eval((Call)s, state);

//else

//return V((Call)s, state);

}

if (s instanceof Return)

return Eval((Return)s, state);

throw new IllegalArgumentException("no statement");

}

// call without return value

State Eval(Call c, State state) {

//

// evaluate call without return value

Value v = state.get(c.fid); // find function

Function f = v.funValue();

State s = newFrame(state, c, f); // create new frame on the stack

s = Eval(f.stmt, s); // interpret the call

s = deleteFrame(s, c, f); // delete the frame on the stack

return s;

}

// value-returning call

Value V (Call c, State state) {

Value v = state.get(c.fid); // find function

Function f = v.funValue();

State s = newFrame(state, c, f); // create new frame on the stack

s = Eval(f.stmt, s); // interpret the call

v = s.pop().val; // remove the return value

s = deleteFrame(s, c, f); // delete the frame on the stack

return v;

}

State Eval(Return r, State state) {

Value v = V(r.expr, state);

return state.push(r.fid, v);

}

State newFrame (State state, Call c, Function f) {

if (c.args.size() == 0)

return state;

// evaluate arguments

// activate a new stack frame in the stack

Value val[] = new Value[f.params.size()];

int i = 0;

for(Expr e: c.args)

val[i++] = V(e,state);

allocate(f.params, state);

//인자값을 매개변수에 전달.................

for(i=0;i<f.params.size();i++)

state.set(f.params.get(i).id,val[i]);

return state;

}

State deleteFrame (State state, Call c, Function f) {

//

// free a stack frame from the stack

//

free(f.params,state);

return state;

}

State Eval(Empty s, State state) {

return state;

}

State Eval(Assignment a, State state) {

//add array assignment

if(a.ar == null) {

Value v = V(a.expr, state);

return state.set(a.id, v);

}

else {

Value v = V(a.expr, state);

Identifier i = a.ar.id;

Value n = V(a.ar.expr,state);

(state.get(i).arrValue())[n.intValue()] = v;

return state;

}

}

State Eval(Read r, State state) {

Value v = state.get(r.id);

if (v.type == Type.INT) {

int i = sc.nextInt();

state.set(r.id, new Value(i));

}

if (v.type == Type.BOOL) {

boolean b = sc.nextBoolean();

state.set(r.id, new Value(b));

}

//

// input string

if(v.type == Type.STRING) {

String s = sc.next();

state.set(r.id, new Value(s));

}

return state;

}

State Eval(Print p, State state) {

System.out.println(V(p.expr, state));

return state;

}

State Eval(Stmts ss, State state) {

for (Stmt s : ss.stmts) {

state = Eval(s, state);

}

return state;

}

State Eval(If c, State state) {

if (V(c.expr, state).boolValue( ))

return Eval(c.stmt1, state);

else

return Eval(c.stmt2, state);

}

State Eval(While l, State state) {

if (V(l.expr, state).boolValue( ))

return Eval(l, Eval(l.stmt, state));

else

return state;

}

State Eval(Let l, State state) {

State s = allocate(l.decls, state);

s = Eval(l.stmts,s);

return free(l.decls, s);

}

State allocate(Decls ds, State state) {

// add entries for declared variables on the state

//

if (ds != null)

{

Iterator<Decl> it = ds.iterator();

while(it.hasNext()) {

Decl d = it.next();

if(d.expr != null && d.arraysize == 0)

state.push(d.id, (Value)d.expr);

else if(d.expr == null && d.arraysize == 0) {

if(d.type == Type.INT) state.push(d.id, new Value(0));

if(d.type == Type.BOOL) state.push(d.id, new Value(false));

if(d.type == Type.STRING) state.push(d.id, new Value(""));

}

else state.push(d.id, new Value(new Value[d.arraysize]));

}

return state;

}

return null;

}

State free (Decls ds, State state) {

// free the entries for declared variables from the state

//

if (ds != null) {

Iterator<Decl> it = ds.iterator();

while(it.hasNext()) {

state.pop();

it.next();

}

return state;

}

return null;

}

Value binaryOperation(Operator op, Value v1, Value v2) {

check(!v1.undef && !v2.undef,"reference to undef value");

switch (op.val) {

case "+":

return new Value(v1.intValue() + v2.intValue());

case "-":

return new Value(v1.intValue() - v2.intValue());

case "\*":

return new Value(v1.intValue() \* v2.intValue());

case "/":

return new Value(v1.intValue() / v2.intValue());

//

// relational operations

case "<":

if(v1.type == Type.INT && v2.type == Type.INT) {

return new Value(v1.intValue() < v2.intValue());

}

if(v1.type == Type.STRING && v2.type == Type.STRING) {

int num = v1.toString().compareTo(v2.toString());

if(num < 0)

return new Value(true);

else

return new Value(false);

}

else

throw new IllegalArgumentException("wrong operator type");

case "<=":

if(v1.type == Type.INT && v2.type == Type.INT) {

return new Value(v1.intValue() <= v2.intValue());

}

if(v1.type == Type.STRING && v2.type == Type.STRING) {

int num = v1.toString().compareTo(v2.toString());

if(num <= 0)

return new Value(true);

else

return new Value(false);

}

else

throw new IllegalArgumentException("wrong operator type");

case ">":

if(v1.type == Type.INT && v2.type == Type.INT) {

return new Value(v1.intValue() > v2.intValue());

}

if(v1.type == Type.STRING && v2.type == Type.STRING) {

int num = v1.toString().compareTo(v2.toString());

if(num > 0)

return new Value(true);

else

return new Value(false);

}

else

throw new IllegalArgumentException("wrong operator type");

case ">=":

if(v1.type == Type.INT && v2.type == Type.INT) {

return new Value(v1.intValue() >= v2.intValue());

}

if(v1.type == Type.STRING && v2.type == Type.STRING) {

int num = v1.toString().compareTo(v2.toString());

if(num >= 0)

return new Value(true);

else

return new Value(false);

}

else

throw new IllegalArgumentException("wrong operator type");

case "==":

if(v1.type == Type.INT && v2.type == Type.INT) {

return new Value(v1.intValue() == v2.intValue());

}

if(v1.type == Type.STRING && v2.type == Type.STRING) {

int num = v1.toString().compareTo(v2.toString());

if(num == 0)

return new Value(true);

else

return new Value(false);

}

else

throw new IllegalArgumentException("wrong operator type");

case "!=":

if(v1.type == Type.INT && v2.type == Type.INT) {

return new Value(v1.intValue() != v2.intValue());

}

if(v1.type == Type.STRING && v2.type == Type.STRING) {

int num = v1.toString().compareTo(v2.toString());

if(num != 0)

return new Value(true);

else

return new Value(false);

}

else

throw new IllegalArgumentException("wrong operator type");

//

// logical operations

case "&":

return new Value(v1.boolValue() && v2.boolValue());

case "|":

return new Value(v1.boolValue() || v2.boolValue());

default:

throw new IllegalArgumentException("no operation");

}

}

Value unaryOperation(Operator op, Value v) {

check( !v.undef, "reference to undef value");

switch (op.val) {

case "!":

return new Value(!v.boolValue( ));

case "-":

return new Value(-v.intValue( ));

default:

throw new IllegalArgumentException("no operation: " + op.val);

}

}

static void check(boolean test, String msg) {

if (test) return;

System.err.println(msg);

}

Value V(Expr e, State state) {

if (e instanceof Value)

return (Value) e;

if (e instanceof Identifier) {

Identifier v = (Identifier) e;

return (Value)(state.get(v));

}

if (e instanceof Binary) {

Binary b = (Binary) e;

Value v1 = V(b.expr1, state);

Value v2 = V(b.expr2, state);

return binaryOperation (b.op, v1, v2);

}

if (e instanceof Unary) {

Unary u = (Unary) e;

Value v = V(u.expr, state);

return unaryOperation(u.op, v);

}

if (e instanceof Call)

return V((Call)e, state);

if (e instanceof Array) {

Identifier i = ((Array) e).id;

Value n = V(((Array)e).expr,state);

return (Value)((state.get(i).arrValue())[n.intValue()]);

}

throw new IllegalArgumentException("no operation");

}

public static void main(String args[]) {

if (args.length == 0) {

Sint sint = new Sint(); Lexer.interactive = true;

System.out.println("Language S Interpreter 1.0");

System.out.print(">> ");

Parser parser = new Parser(new Lexer());

do { // Program = Command\*

if (parser.token == Token.EOF)

parser.token = parser.lexer.getToken();

Command command=null;

try {

command = parser.command();

//command.type = TypeChecker.Check(command);

} catch (Exception e) {

System.out.println(e);

System.out.print(">> ");

continue;

}

if (command.type != Type.ERROR) {

System.out.println("\nInterpreting..." );

try {

state = sint.Eval(command, state);

} catch (Exception e) {

System.err.println(e);

}

}

System.out.print(">> ");

} while (true);

}

else {

System.out.println("Begin parsing... " + args[0]);

Command command = null;

Parser parser = new Parser(new Lexer(args[0]));

Sint sint = new Sint();

do { // Program = Command\*

if (parser.token == Token.EOF)

break;

try {

command = parser.command();

// command.type = TypeChecker.Check(command);

} catch (Exception e) {

System.out.println(e);

continue;

}

if (command.type!=Type.ERROR) {

System.out.println("\nInterpreting..." + args[0]);

try {

state = sint.Eval(command, state);

} catch (Exception e) {

System.err.println(e);

}

}

} while (command != null);

}

}

}