**2023학년도 1학기 [프로그래밍언어론]**

**Lab03 과제 보고서**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 제출일 | 5월 3일 | 분반 | 002 | 학과 | 소프트웨어학부  컴퓨터과학전공 |
| 학번 | 2016133 | 이름 | 이유진 | | |

1. 구현한 프로그램에 대한 설명

Language S의 인터프리터를 구현한다.

1. Let 문 구현을 위한 allocate 함수와 free 함수 구현

Sint.java 수정

* 1. 선언된 변수들(decls)을 위한 엔트리를 상태 state에 추가하는 allocate 함수

전역변수와 지역변수의 중복선언이 가능하도록 구현

(값이 덮어씌워지지 않게)

* 1. 선언된 변수들(decls)을 위한 엔트리를 상태 state에서 제거하는 free 함수

stack에서 pop하는 역할

1. Language S의 문법에 따라 관계 및 논리 연산 수행 기능 구현

Sint.java의 binaryOperation()울 확장

* 1. Relational operations

==, !=, >, >=, <, <=

String의 대소 비교는 사전 순 : compareTo 이용

* 1. Logical operations

&, |

1. Language S의 문장에 do-while문, for문을 추가, 이를 해석하는 인터프리터 작성

Parser.java 수정

* 1. Do-while문

private Stmt stmt()의 switch~case문에 ***DO*** 추가

private Stmts dowhileStmt() 생성, while문을 이용해 구현

* 1. For 문

private Stmt stmt()의 switch~case문에 ***FOR*** 추가

private Let forStmt() 생성, let과 while문을 이용해 구현

이외에 lab03에서 사용하지 않는 코드 적절히 주석처리 하였음.

2. 실행 결과 캡처 사진

1) hi0.s

텍스트이(가) 표시된 사진

자동 생성된 설명

2) hi2.s

텍스트, 모니터, 스크린샷이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 모니터이(가) 표시된 사진

자동 생성된 설명

3) hi3.s

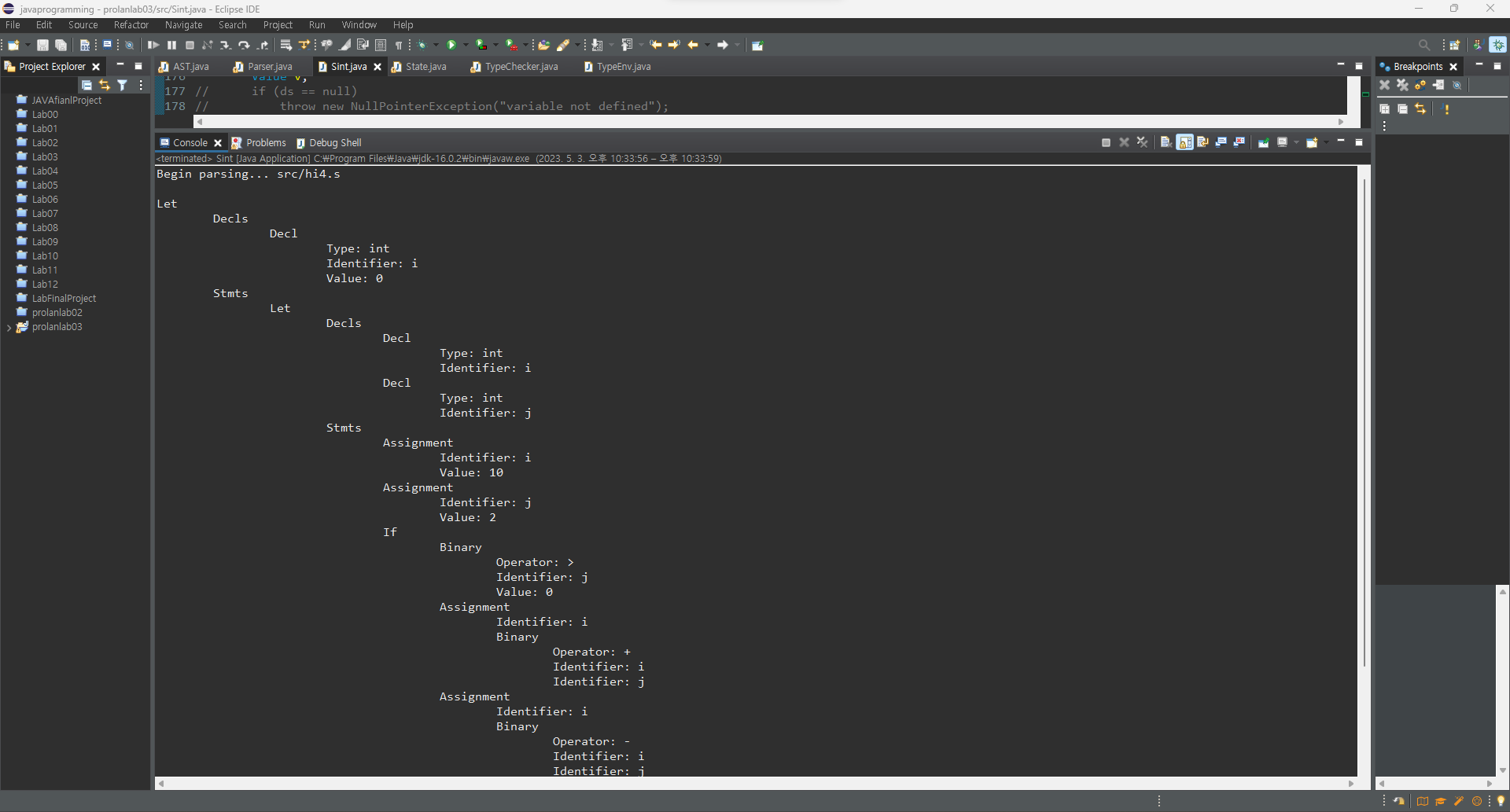
텍스트, 모니터, 스크린샷, 화면이(가) 표시된 사진

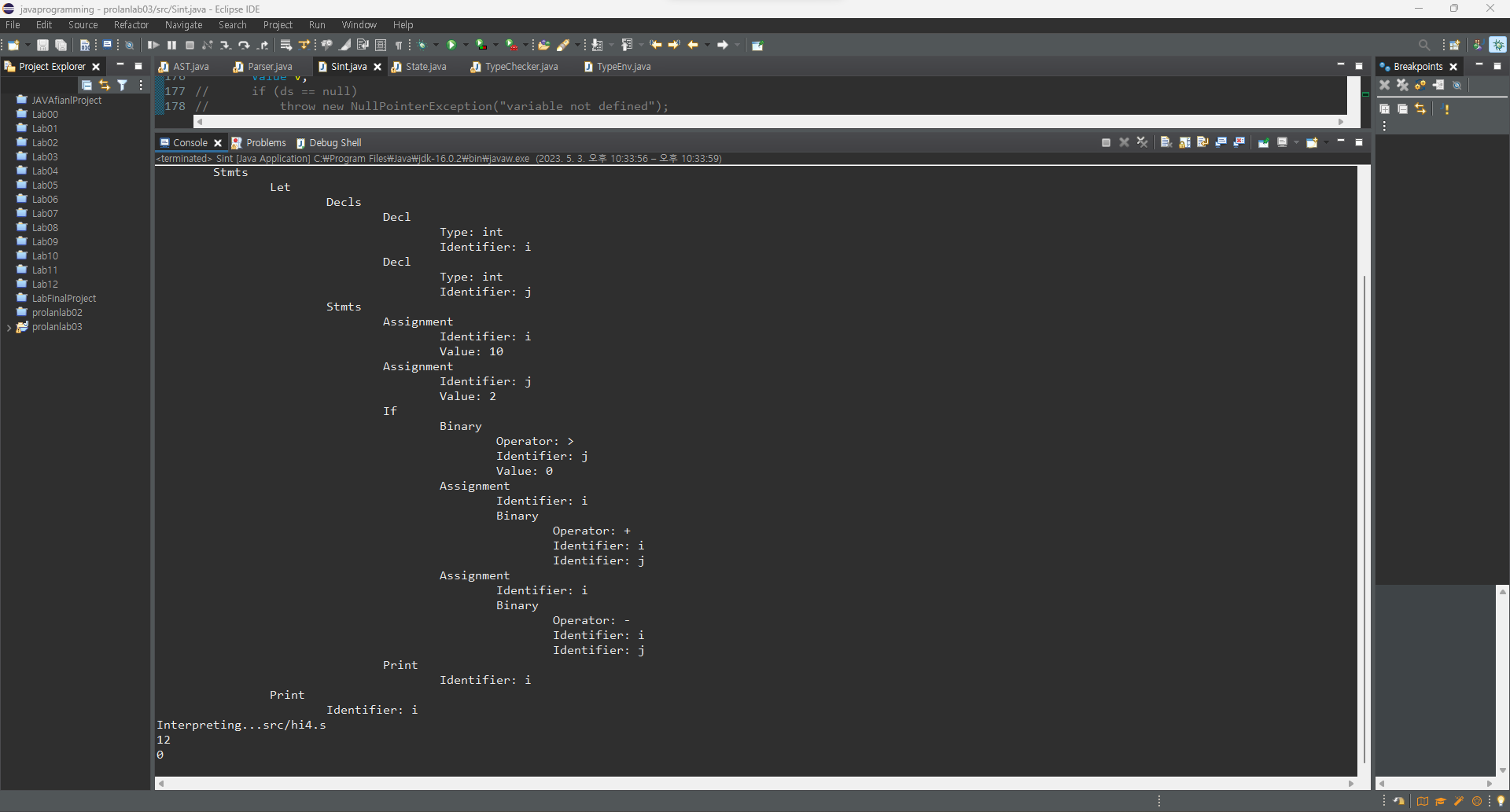
자동 생성된 설명

텍스트, 스크린샷, 모니터이(가) 표시된 사진

자동 생성된 설명

4) hi4.s





5) hi5.s

텍스트, 스크린샷, 모니터, 화면이(가) 표시된 사진

자동 생성된 설명

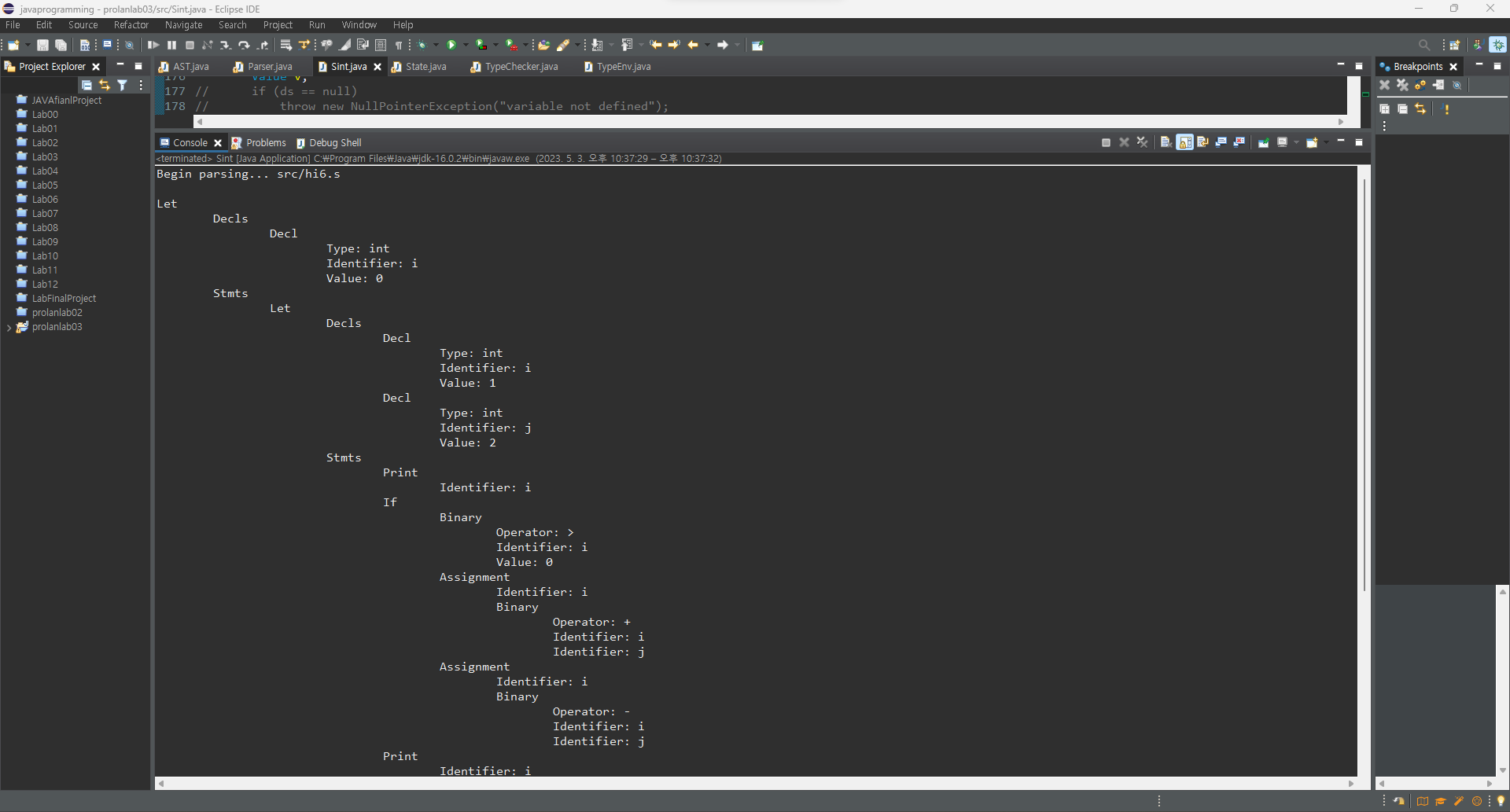
텍스트, 스크린샷, 모니터이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 모니터이(가) 표시된 사진

자동 생성된 설명

6) hi6.s



텍스트, 스크린샷, 모니터이(가) 표시된 사진

자동 생성된 설명

7) hi7.s

텍스트, 스크린샷, 모니터이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 모니터이(가) 표시된 사진

자동 생성된 설명

8) stringrelop1.s

텍스트이(가) 표시된 사진

자동 생성된 설명

9) stringrelop2.s

텍스트이(가) 표시된 사진

자동 생성된 설명

10) logicalop1.s

텍스트이(가) 표시된 사진

자동 생성된 설명

11) logicalop2.s

텍스트이(가) 표시된 사진

자동 생성된 설명

12) for.s

텍스트이(가) 표시된 사진

자동 생성된 설명

13) dowhile.s

텍스트, 스크린샷, 모니터이(가) 표시된 사진

자동 생성된 설명

3. 프로그램 소스 코드 (복사-붙여넣기)

1) AST.java

// AST.java

// AST for S

// 프로그래밍언어론 lab03 소프트웨어학부 컴퓨터과학전공 2016133 이유진

import java.util.\*;

class Indent {

public static void display(int level, String s) {

String tab = "";

System.***out***.println();

for (int i = 0; i < level; i++)

tab = tab + " ";

System.***out***.print(tab + s);

}

}

abstract class Command {//done

// Command = Decl | Function | Stmt

Type type = Type.***UNDEF***;

public void display(int l) {

Indent.*display*(l,getClass().toString().substring(6));

}

}

class Decls extends ArrayList<Decl> { //done

// Decls = Decl\*

public ArrayList<Decl> ds = new ArrayList<Decl>();

Decls() { super(); };

Decls(Decl d) {

this.add(d);

}

public void display(int level){

Indent.*display*(level,"Decls");

for (Decl decl:this){

decl.display(level+1);

}

}

}

class Decl extends Command { //done

// 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

public void display(int level){

super.display(level);

type.display(level+1);

id.display(level+1);

if (expr!=null){

expr.display(level+1);

}

}

}

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();

//return id.toString();

}

}

class Type {

// Type = int | bool | string | fun | array | except | 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 ***EXC*** = new Type("exc");

final static Type ***RAISEDEXC*** = new Type("raisedexc");

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; }

public void display(int level){ //done

Indent.*display*(level,"Type: "+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 {//done

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

public void display(int l) {

Indent.*display*(l,getClass().toString().substring(6)+"");

}

}

class Empty extends Stmt { //done

public void display(int level){

super.display(level);

}

}

class Stmts extends Stmt {

// Stmts = Stmt\*

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

Stmts() {

super();

}

Stmts(Stmt s) {

stmts.add(s);

}

Stmts(Stmt s, While w){ //lab03 done

stmts.add(s);

stmts.add(w);

}

Stmts(Stmt s, Assignment a){ //lab03 done

stmts.add(s);

stmts.add(a);

}

public void display(int level){ //done

Indent.*display*(level,"Stmts");

for (Stmt s:stmts){

if (s!=null){

s.display(level+1);

}

}

}

}

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;

}

public void display(int level) {

Indent.*display*(level, "Assignment");

id.display(level+1);

expr.display(level+1);

}

}

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;

}

public void display(int level){//done

super.display(level);

expr.display(level+1);

stmt1.display(level+1);

stmt2.display(level+1);

}

}

class While extends Stmt { //

// While = Expr expr; Stmt stmt;

Expr expr;

Stmt stmt;

While (Expr t, Stmt b) {

expr = t; stmt = b;

}

public void display(int level){ //done

super.display(level);

expr.display(level+1);

stmt.display(level+1);

}

}

class Let extends Stmt { //

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

Decls decls;

Functions funs;

Stmts stmts;

Let(Decls ds, Stmts ss) {

decls = ds;

funs = null;

stmts = ss;

}

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

decls = ds;

funs = fs;

stmts = ss;

}

public void display(int level){ //done

super.display(level);

decls.display(level+1);

stmts.display(level+1);

}

}

class Read extends Stmt { //done

// Read = Identifier id

Identifier id;

Read (Identifier v) {

id = v;

}

public void display(int level){

super.display(level);

id.display(level+1);

}

}

class Print extends Stmt { //done

// Print = Expr expr

Expr expr;

Print (Expr e) {

expr = e;

}

public void display(int level){

super.display(level);

expr.display(level+1);

}

}

class Return extends Stmt { //done

Identifier fid;

Expr expr;

Return (String s, Expr e) {

fid = new Identifier(s);

expr = e;

}

public void display(int level){

super.display(level);

fid.display(level+1);

expr.display(level+1);

}

}

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> {//done

// Exprs = Expr\*

public ArrayList<Expr> exprs = new ArrayList<Expr>();

public void display(int level){

for (Expr e:exprs){

if (e!=null){

e.display(level+1);

}

}

}

}

abstract class Expr extends Stmt { //done

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

public void display(int l) {

Indent.*display*(l,getClass().toString().substring(6)+":");

}

}

class Call extends Expr { //done

Identifier fid;

Exprs args;

Call(Identifier id, Exprs a) {

fid = id;

args = a;

}

public void display(int level){

super.display(level);

fid.display(level+1);

args.display(level+1);

}

}

class Identifier extends Expr { //done

// 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);

}

public void display(int level){ //추가

Indent.*display*(level,"Identifier: "+id);

}

}

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 ""; //null

}

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( ) { //done

//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";

}

public void display(int level){

Indent.*display*(level,"Value: "+value);

}

}

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

public void display(int level) {

Indent.*display*(level, "Binary");

op.display(level+1);

expr1.display(level+1);

expr2.display(level+1);

}

}

class Unary extends Expr { //done

// 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

public void display(int level){

super.display(level);

op.display(level+1);

expr.display(level+1);

}

}

class Operator { //done

String val;

Operator (String s) {

val = s;

}

public String toString( ) {

return val;

}

public boolean equals(Object obj) {

return val.equals(obj);

}

public void display(int level){

Indent.*display*(level,"Operator: "+val);

}

}

//AST.java

2) Parser.java

// Parser.java

// Parser for language S

// 프로그래밍언어론 lab03 소프트웨어학부 컴퓨터과학전공 2016133 이유진

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 t = type();

String id = match(*Token*.***ID***);

Decl d = null;

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();

// parse declrations of parameters

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

s = assignment(); return s;

case ***LET***: // let statement

s = letStmt(); return s;

case ***READ***: // read statement

s = readStmt(); return s;

case ***PRINT***: // print statment

s = printStmt(); return s;

case ***RETURN***: // return statement

s = returnStmt(); return s;

case ***DO***: //lab03 done

s = dowhileStmt(); return s;

case ***FOR***: //lab03 done

s = forStmt(); 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() { //done

// <readStmt> -> read id;

match(*Token*.***READ***);

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

match(*Token*.***SEMICOLON***);

return new Read(id);

}

private Print printStmt() { //done

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

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() {

// <assignment> -> id = <expr>;

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

/\*

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

return call(id);

\*/

match(*Token*.***ASSIGN***);

Expr e = expr();

match(*Token*.***SEMICOLON***);

return new Assignment(id, e);

}

/\*

private Call call(Identifier id) {

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

//

// parse function call

//

return null;

}

\*/

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 () { //done

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

match(*Token*.***WHILE***);

match(*Token*.***LPAREN***);

Expr e = expr();

match(*Token*.***RPAREN***);

Stmt s = stmt();

return new While(e,s);

}

private Stmts dowhileStmt() { //lab03 done

// <dowhileStmt> -> do <stmt> while (<expr>);

match(*Token*.***DO***);

Stmt s = stmt();

match(*Token*.***WHILE***);

match(*Token*.***LPAREN***);

Expr e = expr();

match(*Token*.***RPAREN***);

match(*Token*.***SEMICOLON***);

While w = new While(e,s);

return new Stmts(s,w);

}

private Let forStmt() { //lab03 done

// <forStmts> -> for(<type> id = <expr>; <expr>; id=<expr>) <stmt>

match(*Token*.***FOR***);

match(*Token*.***LPAREN***);

Decls ds = decls();

Expr e1 = expr();

match(*Token*.***SEMICOLON***);

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

match(*Token*.***ASSIGN***);

Expr e2 = expr();

match(*Token*.***RPAREN***);

Stmt s = stmt();

Assignment a = new Assignment(id,e2);

Stmts wss = new Stmts(s,a);

While w = new While(e1,wss);

Stmts ss = new Stmts(w);

return new Let(ds,null,ss);

}

private Expr expr () { //done

// <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();

Expr b = null;

Operator op = null;

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

op = new Operator(match(token));

b = bexp();

return new Binary(op,e,b);

}

return e;

}

private Expr bexp() { //done

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

Expr e = aexp();

Expr a = null;

Operator op = null;

switch (token) {

case ***LT***:

op = new Operator(match(token));

a = aexp();

return new Binary(op,e,a);

case ***LTEQ***:

op = new Operator(match(token));

a = aexp();

return new Binary(op,e,a);

case ***GT***:

op = new Operator(match(token));

a = aexp();

return new Binary(op,e,a);

case ***GTEQ***:

op = new Operator(match(token));

a = aexp();

return new Binary(op,e,a);

case ***EQUAL***:

op = new Operator(match(token));

a = aexp();

return new Binary(op,e,a);

case ***NOTEQ***:

op = new Operator(match(token));

a = aexp();

return 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> | 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***));

e = v;

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

match(*Token*.***LPAREN***);

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

match(*Token*.***RPAREN***);

e = c;

}

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***.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();

if (command != null) command.display(0); // display AST

} 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 {

if (parser.token == *Token*.***EOF***) break;

try {

command = parser.command();

if (command != null) command.display(0); // display AST

} catch (Exception e) {

System.***err***.println(e);

}

} while (command != null);

}

} //main

} // Parser.java

3) Sint.java

// Sint.java

// Interpreter for S

// 프로그래밍언어론 lab03 소프트웨어학부 컴퓨터과학전공 2016133 이유진

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) {

//전역 변수 선언; 상태 변환 함수 Eval

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)

return Eval((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

//

return state;

}

// 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.peek().val; // get the return value v = s.get(new Identifier("return"));

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.set(new Identifier("return"), 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

//

state.push(new Identifier("return"), null); // allocate for return value

return state;

}

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

state.pop(); // pop the return value

//

// free a stack frame from the stack

//

return state;

}

\*/

State Eval(Empty s, State state) {

return state;

}

State Eval(Assignment a, State state) {

// if (state.lookup(a.id) == -1){

// throw new NullPointerException("variable not defined");

// } else {

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

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

// }

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

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

}

State Eval(Read r, State state) {

int n = *sc*.nextInt(); //lab03 done

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

if (r.id.type == Type.***INT***) {

int i = *sc*.nextInt();

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

}

if (r.id.type == Type.***BOOL***) {

boolean b = *sc*.nextBoolean();

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

}

// input string

if (r.id.type == Type.***STRING***) { //lab03 done

String str = *sc*.next();

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

}

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);

// if (s instanceof Return) //?

// return 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); //stack에 let~end의 엔트리 쌍을 만들어줌

s = Eval(l.stmts,s);

return free(l.decls, s);

}

State allocate (Decls ds, State state) {//lab03

//선언된 변수들(ds)을 위한 엔트리들을 상태 state에 추가

Identifier id;

Value v;

// if (ds == null)

// throw new NullPointerException("variable not defined");

if (ds != null) {

// add entries for declared variables on the state

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

id = ds.get(i).id;

v = (Value) ds.get(i).expr;

if (state.lookup(id) != -1)

state = state.push(id, v);

else

state.push(id, v);

}

}

else

throw new NullPointerException("variable not declared");

return state;

// Identifier id;

// Value v;

// if (ds != null){

// for (int i=0;i<ds.size();i++){

// id=ds.get(i).id;

// v=(Value)ds.get(i).expr;

// state.push(id,(Value)v);

// }

// } else{

// throw new NullPointerException("variable not declared");

// }

// return state;

}

State free (Decls ds, State state) {//lab03 done

//선언된 변수들(ds)의 엔트리를 상태 state에서 제거

if (ds != null && !state.isEmpty()) {

// free the entries for declared variables from the state

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

state.pop();

}

}

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 : lab03 done

case "=="://done

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

return new Value(v1.stringValue().equals(v2.stringValue()));

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

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

case "!="://done

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

return new Value(!v1.stringValue().equals(v2.stringValue()));

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

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

case ">"://done

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

return new Value(v1.stringValue().compareTo(v2.stringValue()) > 0);

}

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

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

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

case ">="://done

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

return new Value(v1.stringValue().compareTo(v2.stringValue()) >= 0);

}

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

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

}

case "<"://done

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

return new Value(v1.stringValue().compareTo(v2.stringValue()) < 0);

}

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

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

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

case "<="://done

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

return new Value(v1.stringValue().compareTo(v2.stringValue()) <= 0);

}

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

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

}

// logical operations : lab03 done

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);

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 2.0");

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();

//if (command != null) command.display(0); // display AST

if (command == null)

throw new Exception();

else {

command.display(0);

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

System.out.println("\nType: "+ command.type);\*/

}

} 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();

// if (command != null) command.display(0); // display AST

if (command == null)

throw new Exception();

else {

command.display(0);

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

System.out.println("\nType: "+ command.type); \*/

}

} 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);

}

}

}

//Sint.java