2021-01

프로그래밍언어론(001분반)

Lab03

보고서

**학과:소프트웨어융합전공**

**학번: 1912339**

**이름: 신지우**

**제출일: 2021/4/13**

1. 구현한 프로그램에 대한 설명

(1-1) allocate 함수

변수 선언의 집합 Decls의 타입이 arraylist임을 이용하여, Decls에 들어있는 모든 Decl을 순차적으로 방문하여 차례대로 state 스택에 집어넣도록 구현했다.

만약 Decl의 수식(값)이 존재한다면, 즉 사용자가 정한 값으로 초기화한다면, state에 식별자로 Decl의 식별자(id)를 넣어주고, 값으로서 Decl의 수식을 넣어주었다.

만약 Decl의 수식이 존재하지 않다면, 즉 자동 초기화할 때는, state의 식별자는 Decl의 식별자(id)를 넣어주고, 값으로서는 Decl의 type에 따라(Decl의 도입부가 int/bool/string 중 무엇으로 시작하는지에 따라) int는 0, bool은 false, string은 빈 문자열로 초기화해주었다.

(1-2) free 함수

현재 위치한 let문을 빠져나갈 때, 해당 let문의 변수 선언의 집합 Decls에 들어있는 모든 Decl을 순차적으로 방문하는 것을 카운트로 하여 그 수만큼 state 스택을 pop()해주었다.

(2-1) 관계 연산

정수 관계 연산과 스트링 관계 연산의 구분은 binaryOperation함수의 인수 v1과 v2의 type을 확인함으로써 구현했다.

정수 관계 연산이라면 v1, v2의 타입을 int로 변환시켜 관계 연산을 해준 후, 그 값을 리턴했다.

스트링 관계 연산은 사전순으로 봤을 때, 더 뒷쪽에 등장하는 스트링이 더 큰 값이라고 간주하고 구현했다. compareTo함수를 사용했을 때, 현 스트링이 사전순으로 더 앞쪽이면 음수, 스트링이 같으면 0, 사전순으로 더 뒷쪽이면 양수인 것을 이용하여 그 값에 따라 true, false를 리턴해주었다.

(2-2) 논리연산 구현

v1, v2의 타입을 bool로 변환 시켜 논리연산을 해준 후, 그 값을 리턴했다.

(\*) Eval(Read r, State state){} 구현

//input string이라는 주석이 있길래 이것까지 해야하는 줄 알고 구현했다.

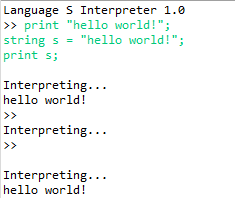
AST.java에서 class Read와 class Identifier를 참고하면, 현재 read문에 쓰인 식별자의 이름은 알 수 있지만, 그 식별자의 type은 알 수 없다는 걸 확인할 수 있다. 따라서 원래 주어진 형식을 이용하지 않고 조금 다르게 작성했다.

state 스택에서 read문에서 쓰인 식별자와 같은 이름을 가진, 가장 최근에 만들어진 식별자(id)의 값(Value)을 찾아내기 위해 state의 get함수를 이용했다.

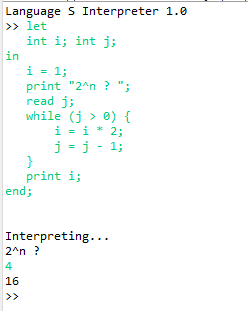
일련의 if문들은 read문의 id의 타입을 구분하기 위해 state 스택의 식별자의 값의 타입을 구분용으로 사용한 것이다. type이 int인 경우는 int 변수로 사용자 입력값을 받아 set함수의 인수로 사용하여 식별자의 값을 교체해주었다. type이 bool인 경우는 bool 변수로 사용자 입력값을 받아 set함수의 인수로 사용하여 식별자의 값을 교체해주었다. type이 string인 경우는 string 변수로 사용자 입력값을 받아 set함수의 인수로 사용하여 식별자의 값을 교체해주었다.

2. 실행 결과 캡처 사진

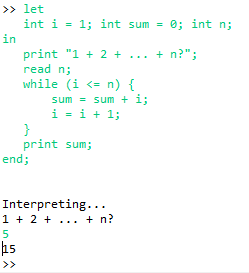
hi0.s : interpreting...이 여러 번 출력되긴 하지만 “hello world!”가 2번(정상횟수) 출력됨



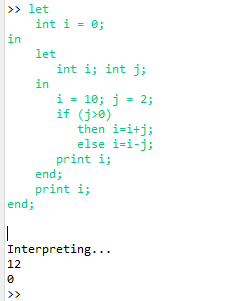
hi2.s



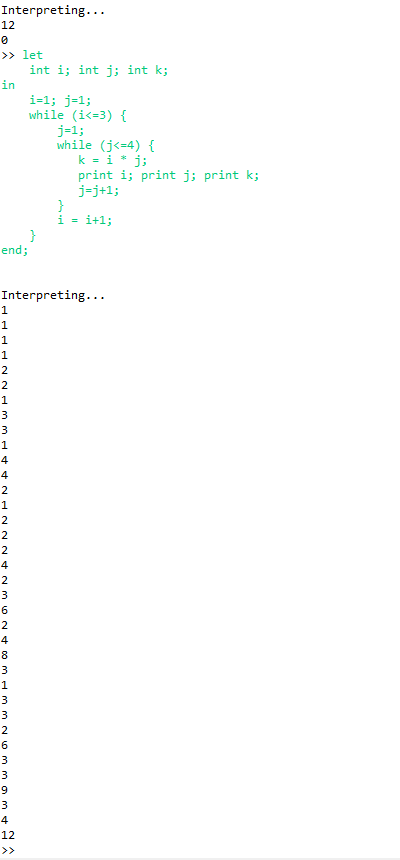
hi3.s



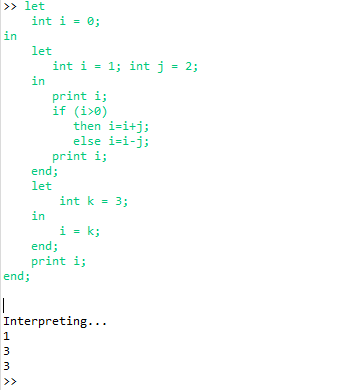
hi4.s



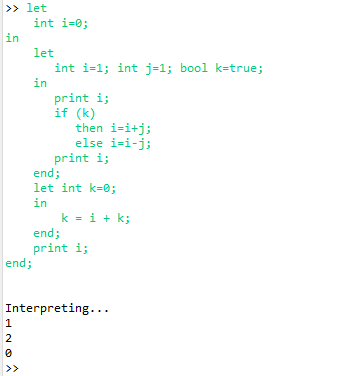
hi5.s



hi6.s : hi6.s 파일 중 빈 공백줄 삭제하고 돌림.(빈 공백줄 존재 시 컴파일 에러)

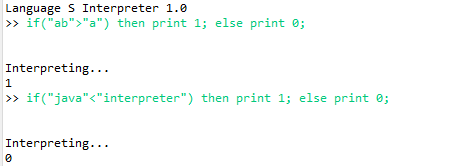


hi7.s

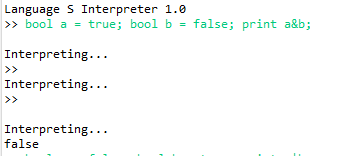


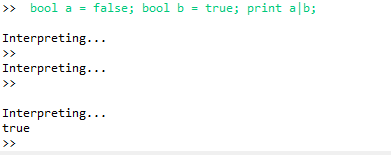
- 스트링 관계연산 테스트 2개

(사전순으로 뒤에 나오는 게 더 크다고 간주한 계산)



- 논리연산 테스트 2개





3. 프로그램 소스 코드 (복사-붙여넣기)

Sint.java와 AST.java(Sint.java가 굴러가도록 하기 위해 일부 수정 있었음)

// 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** 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");

}

State Eval(Empty s, State state) {

**return** state;

}

State Eval(Assignment a, State state) {

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

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

}

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**)

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

**else** {

**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(""));

}

}

**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);

**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**);

}

}

}

// AST.java

// AST for S

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 {

// Command = Decl | Function | Stmt

Type type =Type.UNDEF;

public void display(int l) { }

}

class Decls extends ArrayList<Decl> {

// Decls = Decl\*

ArrayList<Decl> decls = new ArrayList<Decl>();

public void display(int level) {

Indent.display(level, "Decls");

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

while(it.hasNext()) {

it.next().display(level+1);

}

}

}

class Decl extends Command {

// Decl = Type type; Identifier id

Identifier id;

Expr expr = null;

Decl (String s, Type t) {

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

} // declaration

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

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

} // declaration

public void display(int level) {

Indent.display(level, "Decl");

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();

}

}

class Type {

// Type = int | bool | string | 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 UNDEF = new Type("undefined");

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) {

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

}

}

class ProtoType extends Type {

// 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>();

public void display(int level) {

Indent.display(level, "Stmts");

Iterator<Stmt> it = stmts.iterator();

while(it.hasNext()) {

it.next().display(level+1);

}

}

}

class Assignment extends Stmt {

// Assignment = Identifier id; Expr expr

Identifier id;

Expr expr;

Assignment (Identifier t, Expr e) {

id = t;

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) {

Indent.display(level, "If");

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) {

Indent.display(level, "While");

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, Functions fs, Stmts ss) {

decls = ds;

funs = fs;

stmts = ss;

}

public void display(int level) {

Indent.display(level, "Let");

decls.display(level+1);

stmts.display(level+1);

}

}

class Read extends Stmt {

// Read = Identifier id

Identifier id;

Read (Identifier v) {

id = v;

}

public void display(int level) {

Indent.display(level, "Read");

id.display(level+1);

}

}

class Print extends Stmt {

// Print = Expr expr

Expr expr;

Print (Expr e) {

expr = e;

}

public void display(int level) {

Indent.display(level, "Print");

expr.display(level+1);

}

}

class Return extends Stmt {

Identifier fid;

Expr expr;

Return (String s, Expr e) {

fid = new Identifier(s);

expr = e;

}

public void display(int level) {

Indent.display(level, "Return");

fid.display(level+1);

expr.display(level+1);

}

}

class Exprs extends ArrayList<Expr> {

// Exprs = Expr\*

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

}

abstract class Expr extends Stmt {

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

public void display(int level) {}

}

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);

}

public void display(int level) {

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

}

}

class Value extends Expr {

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

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;

value = v; undef = false;

}

Object value() { return value; }

int intValue( ) {

if (value instanceof Integer)

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

return 0;

}

boolean boolValue( ) {

if (value instanceof Boolean)

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

return false;

}

String stringValue ( ) {

if (value instanceof String)

return (String) value;

return null;

}

Function funValue ( ) {

if (value instanceof Function)

return (Function) value;

return null;

}

Type type ( ) { return type; }

public String toString( ) {

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

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

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

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

return "undef";

}

public void display(int level) {

Indent.display(level, "Value: "+ this.toString());

}

}

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 {

// Unary = Operator op; Expr expr

Operator op;

Expr expr;

Unary (Operator o, Expr e) {

op = o;

expr = e;

} // unary

public void display(int level) {

Indent.display(level, "Unary");

op.display(level+1);

expr.display(level+1);

}

}

class Operator {

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: "+ this.toString());

}

}