2021-01

프로그래밍언어론(001분반)

Lab04

보고서

**학과:소프트웨어융합전공**

**학번: 1912339**

**이름: 신지우**

**제출일: 2021/5/9**

1. 구현한 프로그램에 대한 설명

Parser.java와 Sint.java를 수정하여 구현하였다.

- Parser.java

1. decl() // <decl> -> <type> id [=<expr>]; | <type> id[<n>];

선언문을 파싱하면서, 식별자 id를 파싱한 후, 왼쪽 대괄호 LBRACKET( [ )가 token으로 들어오면, 배열을 선언한 것으로 판단하여, 그 다음 토큰은 숫자로 예상을 하고 int 변수 n에 저장해주었다.

return을 어떻게 해줄 것인가의 결정은 3가지의 경우로 나뉘었는데, (1) int변수 n이 0이 아니라면, n에 새 입력이 들어왔으므로 배열 선언인 것으로 판단하여 배열 선언 AST 함수로 보내었다.

(2)n이 0이고, 토큰이 “=”인 경우, (3)else(n이 0이고, 토큰이 “=”이 아닌 경우)는 이전과제에서 구현한 내용이다.

2. assignment() // <assignment> -> id = <expr>; | id[<expr>] = <expr>;

대입문을 파싱하면서, 식별자 id를 파싱한 후, 왼쪽 대괄호 LBRACKET( [ )가 token으로 들어오면, 배열을 선언한 것으로 판단하여, 그 다음 토큰은 expr()로 파싱 후, Expr 변수 n에 저장해주었다.(배열원소 인덱스인 n은 식별자 id일수도, 값 Value일수도 있다.) 이후 남은 토큰(”=”, 대입값expr,”;”)을 파싱 후, return을 결정한다.

return의 결정은 2가지 경우로, (1) Array변수 a가 null이 아니라면, a가 생성되었으므로 배열원소의 대입문으로 판단하여 Assignment(array, expr)로 리턴하였다. (2) a가 null이라면, a가 생성되지 않았으므로, 일반 대입문으로 판단하여 Assignment(id, expr)로 리턴하였다.

3. <factor> //<factor> -> id | id[<expr>]

switch(token)문에서 case ID 파트만 수정하였다.

식별자를 Identifier변수 v에 저장한 후, if문에서 (1)왼쪽 대괄호 LBRACKET( [ )가 token으로 들어오면 배열원소 인덱스를 파싱하여 Expr변수 n에 저장한다. 식별자 v와 Expr n을 바탕으로 Array(v,n)을 Array변수 a에 대입한다. 그리고 이 a를 최종리턴되는 변수 e에 대입함으로써, 배열원소를 리턴한다. (2) 왼쪽 대괄호가 token으로 들어오지 않았다면, 최종리턴되는 변수 e에 식별자 변수 v를 대입함으로써, 식별자를 리턴한다.

- Sint.java (함수가 코드 위치 순서로는 Eval, allocate, V인데 선언, 값 가져오기, 대입 순으로 설명)

1. allocate()

선언문 내용에 따라 실제로 state 스택에 변수와 값을 저장하는 함수이다. Decl의 arraylist인 ds에 Decl 선언문이 없을 때까지 계속 state 스택에 변수와 값을 저장 후, ds의 다음 Decl 선언문을 불러온다.

경우의 수 1. expr 있고(사용자 초기화) arraysize ==0인 경우 => type id = expr;

2. expr 없고 arraysize==0인 경우 => type id;

3. arraysize != 0 인 경우 => type id[n];

1,2번 경우는 이전 과제에서 작성한 내용이고, 3번에서는 Decl 변수 d를 바탕으로 d.id를 식별자로 하여 d.arraysize 크기의 Value 배열을 만들어 스택에 저장한다.

2. V(Expr e, State state)

V(e,state)문은 expr e값을 현재 state에 따라 실제로 계산하여 리턴해주는 함수이다. 배열원소를 리턴해주도록 하기 위해, if(e instanceof Array)부분만 추가하였다. Array e의 id를 받아 i에 저장하고, Array e의 expr를 V(e,state)문으로 보내어 계산을 끝마친 후 값의 형태로 받아주었다.(식별자인 경우를 해결하기 위하여) 이후 state문에서 state.get(i)로 new Value()로 감싸인 배열을 찾아내어 arrValue()로 배열을 꺼내준후 n.intValue()를 인덱스로 하여 실제 배열원소값을 리턴해주었다.

3. Eval(Assignment a, State state)

ar==null이면 평범한 대입문이라고 판정하였다.

ar!=null이면 배열원소 대입문이라고 판정하였다.

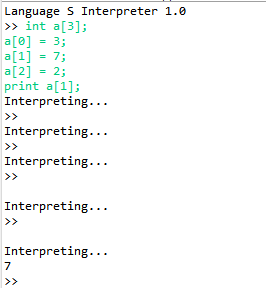
=> 이경우에는 a의 활용할 내부 필드가 Array ar(배열원소 정보), Expr expr(바꿀값)

Value v = V(a.expr,state)를 통해 상태 state에서 a.expr의 의미(값)를 계산하고,

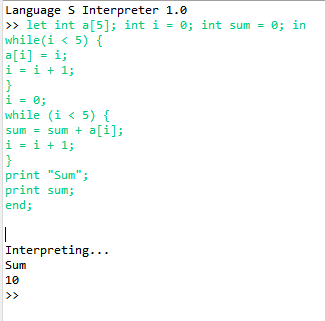
그 값을 v에 대입한다. a.ar.id로 state에서 배열을 찾고, a.ar.expr를 인덱스로 하여 배열 원소를 특정해낸 후, 배열 원소에 v를 대입해주었다.

2. 실행 결과 캡처 사진

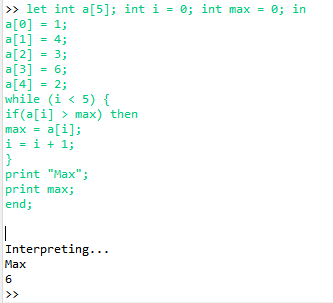
q1



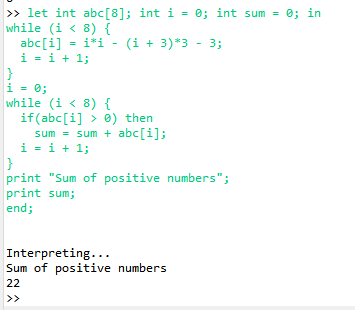
q2



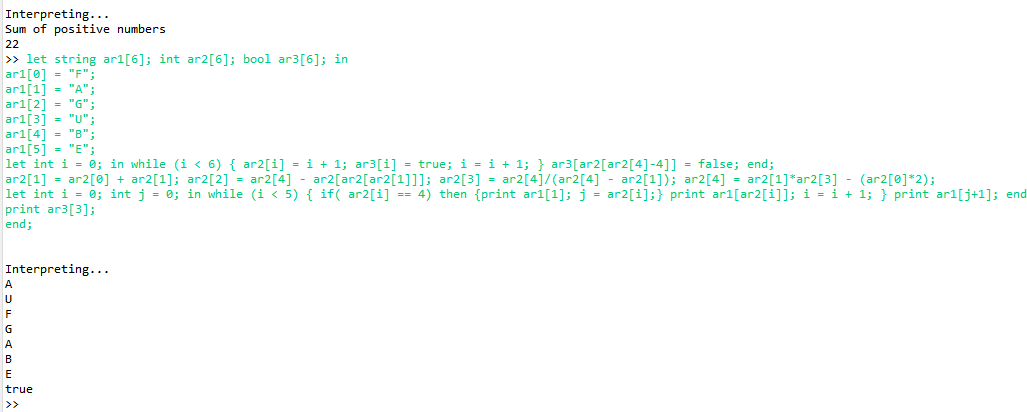
q3



q4



q5



3. 프로그램 소스 코드 (복사-붙여넣기)

Sint.java와 Parser.java

// 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) {

//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 {

/\*if(d.type == Type.INT) state.push(d.id, new Value(new int[d.arraysize]));

if(d.type == Type.BOOL) state.push(d.id, new Value(new boolean[d.arraysize]));

if(d.type == Type.STRING)\*/ 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);

}

}

}

// 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** 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 statement

s = printStmt(); **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** Stmt assignment() {

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

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

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** 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 | id[<expr>] | <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***));

**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** 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();

if (command != null)

command.display(0);

} 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();

if (command != null)

command.display(0);

} catch (Exception e) {

System.err.println(e);

}

} while (command != null);

}

} //main\*/

} //Parser