2021-01

프로그래밍언어론(001분반)

Lab02

보고서

**학과:소프트웨어융합전공**

**학번: 1912339**

**이름: 신지우**

**제출일: 2021/3/30**

1. 구현한 프로그램에 대한 설명

Lexer.java는 어휘분석기로서 토큰 단위로 분리하는 역할을 하는 파일이고, 수정이 필요없는 파일이었다. Parser.java는 Lexer.java에서 넘겨받은 토큰을 통해 재귀 하강 파싱 스타일로 파싱하는 파일이다. Parser.java에서 내가 구현한 파트는 문장 파싱의 readStmt(), printStmt(), whileStmt()와 수식 파싱의 expr()의 논리연산과 bexp()의 비교연산이다.

readStmt()의 구현:

<stmt> -> read id; 이므로 read 토큰을 match하고, id 토큰으로 Identifier 객체를 생성하고, 세미콜론 토큰을 match 후, id 토큰을 인수로 갖는 AST.java의 Read 객체를 리턴한다.

printStmt()의 구현:

<stmt> -> print expr; 이므로 print 토큰을 match하고, expr 토큰으로 expr 객체를 생성하고, 세미콜론 토큰을 match 후, expr 토큰을 인수로 갖는 AST.java의 Print 객체를 리턴한다.

whileStmt()의 구현:

<stmt>->while(expr) stmt 이므로 while 토큰, '(' 토큰을 match 후, expr 토큰으로 expr 객체를 생성하고, ')' 토큰을 match 후, stmt 토큰으로 stmt 객체를 생성한다. expr 객체와 stmt 객체를 인수로 갖는 AST.java의 While 객체를 리턴한다.

expr()의 논리연산 구현:

<expr>-><bexp>{'&'<bexp> | '|'<bexp>} 이므로 token이 '&'토큰이나 '|' 토큰인 동안 반복(while)해서 논리 연산자 토큰으로 operator객체를 생성하고, 뒤이어 오는 수식으로 bexp 객체를 생성 후, AST.java의 Binary 객체를 만들어 리턴한다.

bexp()의 비교연산 구현:

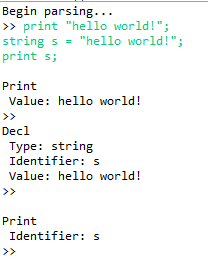
<bexp>-><aexp>[<relop><aexp>]이므로 token이 '==', '<=', '<', '>=','>','!='이면(if) 비교 연산자 토큰으로 operator 객체를 생성하고, 뒤이어 오는 수식으로 aexp 객체를 생성후, AST.java의 Binary 객체를 만들어 리턴한다.

AST.java 파일에서는 class Indent를 추가하여 들여쓰기를 통해 AST를 표현하도록 출력하는 display 함수를 구현하였다. 그리고 모든 수식 class와 문장 class에 display 함수를 상속받거나 정의하였다. level이라는 인수를 통해 들여쓰기의 정도가 결정되는데, 상위 단계 문장에서 display 함수를 수행하면 level+1을 인수로 갖는 그 하위단계의 문장과 수식들의 display 함수가 수행된다. 그래서 한 단계씩 하위로 내려갈수록 들여쓰기가 한 칸씩 많아짐으로써 AST 트리를 표현한다. command, decls, decl, type, stmts, assignment, if, while, let, read, print, return, identifier, value, binary, unary, operator 클래스의 display 함수를 구현하였다.

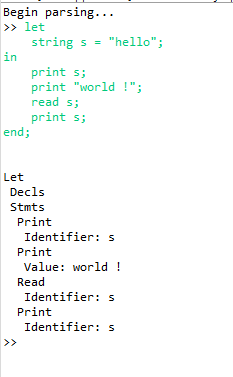
2. 실행 결과 캡처 사진

(Decls 아래에 Decl이 이어져서 파싱되어야하는데 Decls만 나오고 다음 문장으로 나와버린다.)

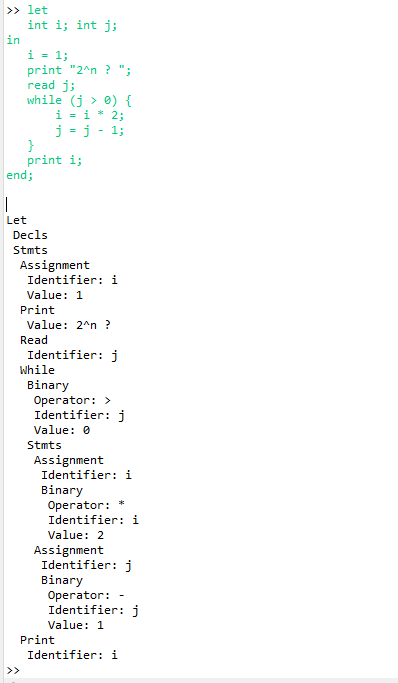
hi0.s



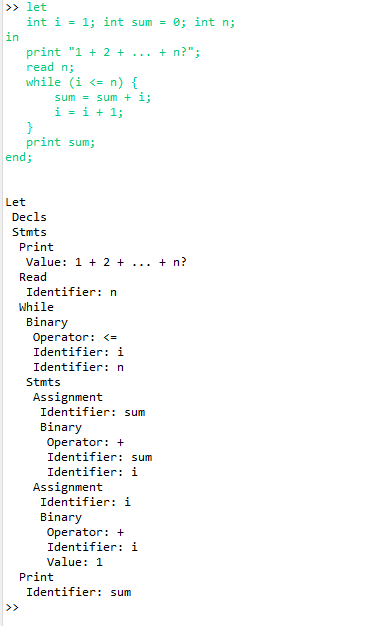
hi1.s



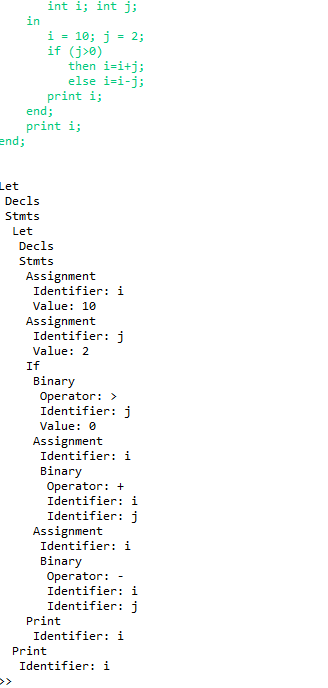
hi2.s



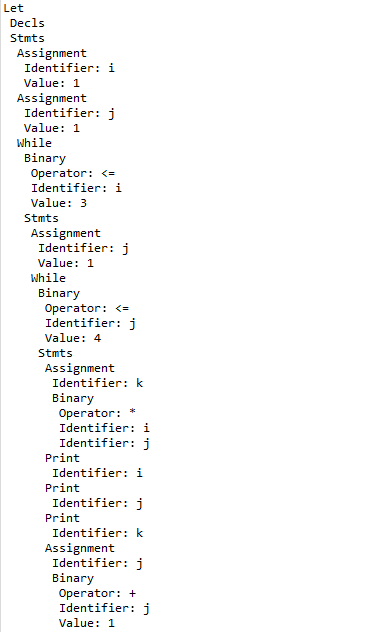
hi3.s

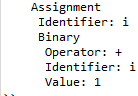


hi4.s



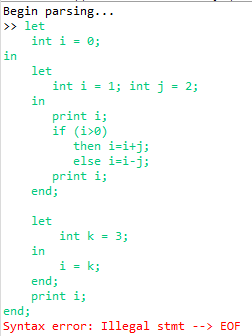
hi5.s





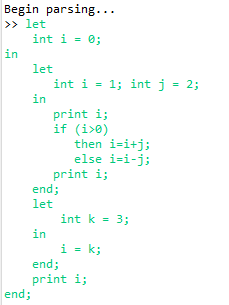
(캡처가 나뉘다보니 마지막 Assignment의 들여쓰기가 애매하게 보이는데 실제로는 안쪽 while과 들여쓰기가 동일함. 제대로 결과가 나옴.)

hi6.s

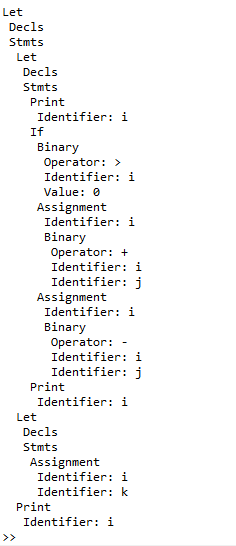


hi6.s 파일의 안쪽 let문 사이의 빈줄이 있는데 그 빈줄 때문에 오류가 발생한다.

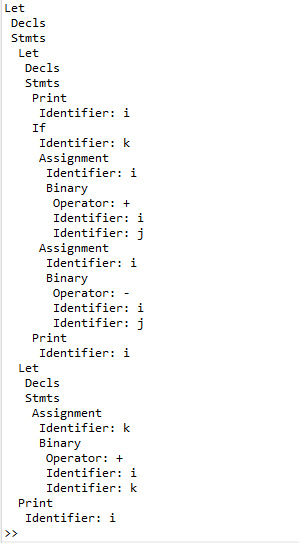
아래 캡처처럼 end;와 let 사이의 빈줄을 삭제하면 제대로 동작한다.



hi6.s 빈줄 삭제 버전.



hi7.s



3. 프로그램 소스 코드 (복사-붙여넣기)(코드 실행 시 엔터키를 2번 쳐야 수행됨.)

// 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 = decls.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;

} // declaration

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

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

} // 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());

}

}

// 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 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** 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;

**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>;

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

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

Expr e = expr();

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

**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 | <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;

**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