**2023학년도 1학기 [프로그래밍언어론]**

**Lab02 과제 보고서**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 제출일 | 4월 12일 | 분반 | 002 | 학과 | 소프트웨어학부  컴퓨터과학전공 |
| 학번 | 2016133 | 이름 | 이유진 | | |

1. 구현한 프로그램에 대한 설명

어휘분석기 Lexer : 입력 스트링을 읽어 토큰 형태로 분리해 반환

파서 Parser : 입력 스트링을 재귀 하강 파싱 -> 해당 입력의 AST를 생성해 반환함

추상구문트리 AST : 입력 스트링의 구문 구조를 추상적으로 보여주는 트리

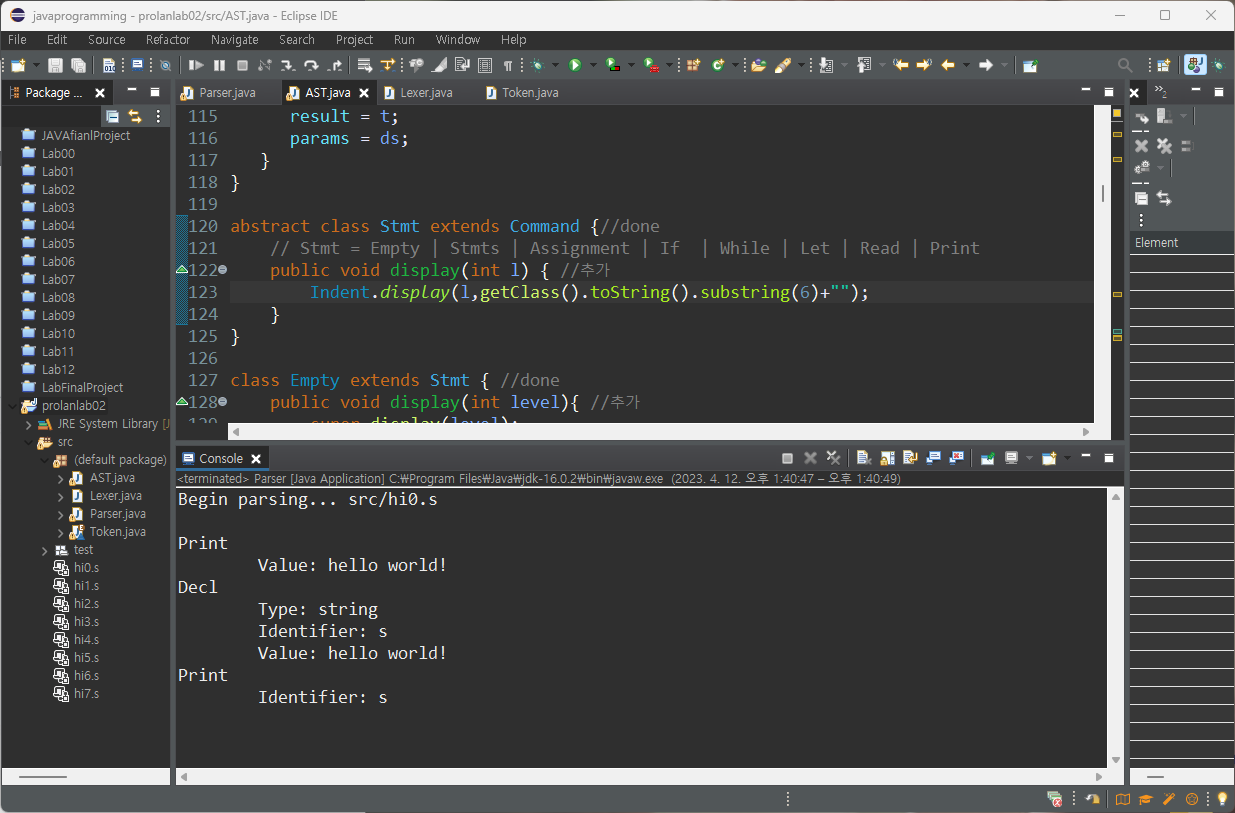
탭을 이용한 레벨 구분으로 구현한 트리를 출력함

display 메서드를 이용해 출력 -> 출력이 필요한 클래스에 display 메서드 추가

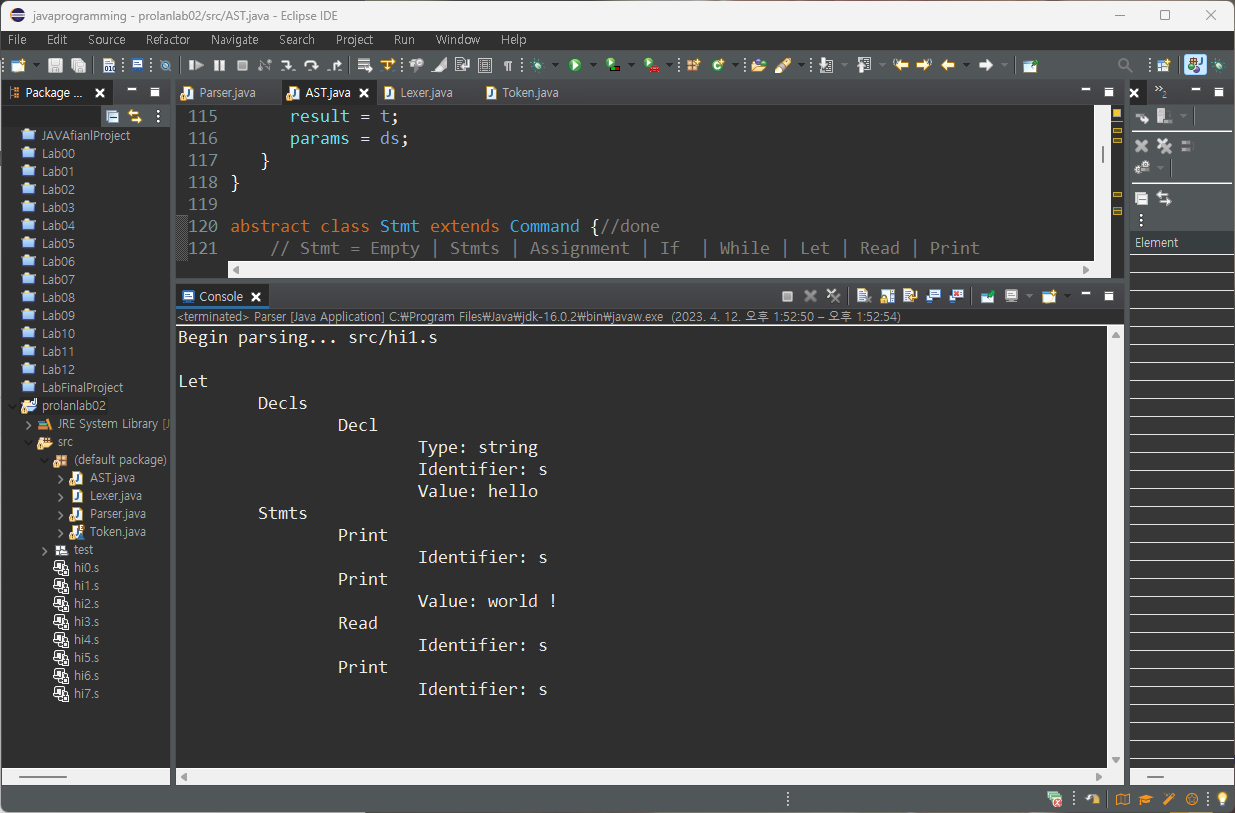
언어 S의 EBNF를 구현하는 것에 유의하여 파서 코드 작성함

2. 실행 결과 캡처 사진

1) hi0.s



2) hi1.s

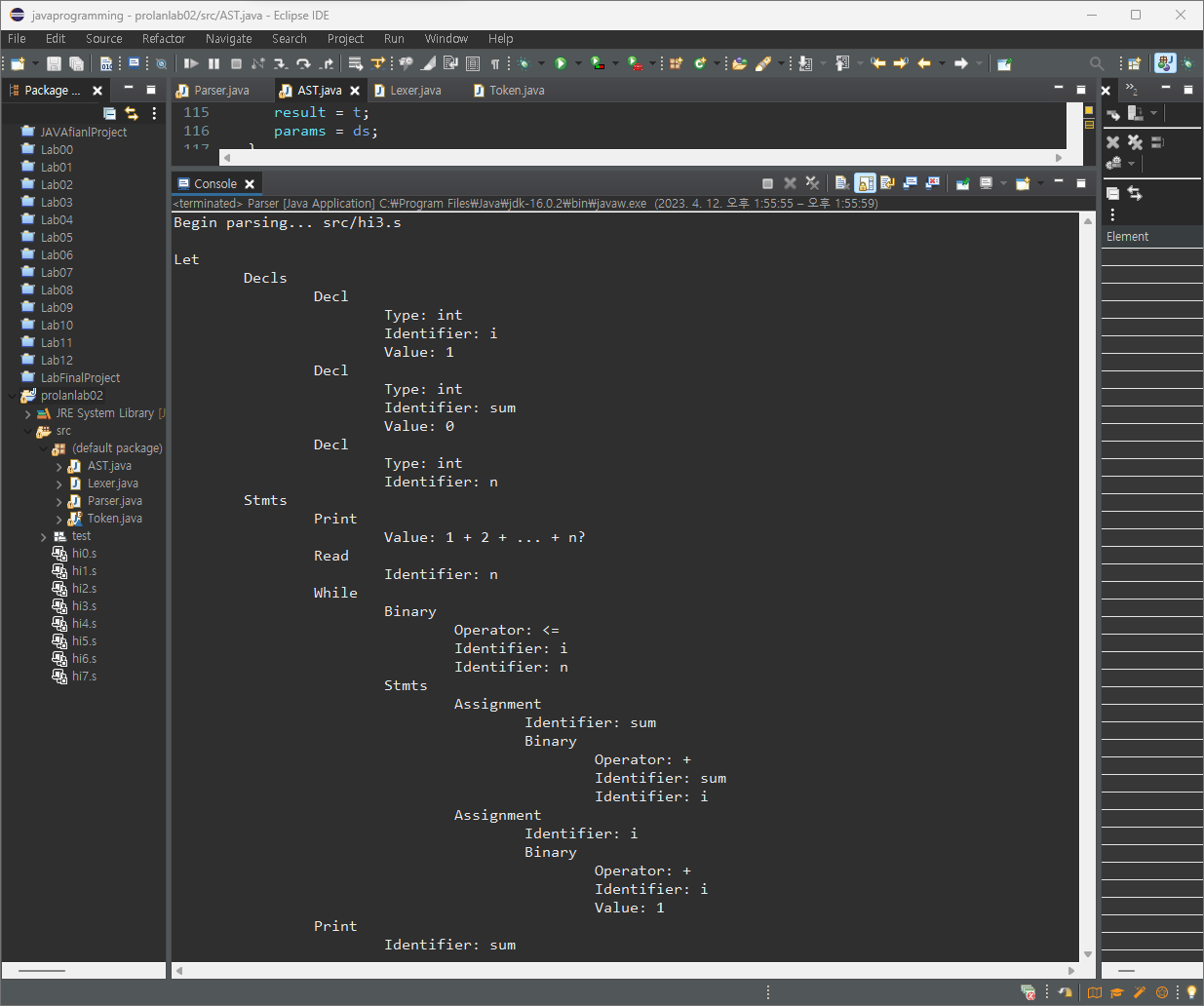


3) hi2.s

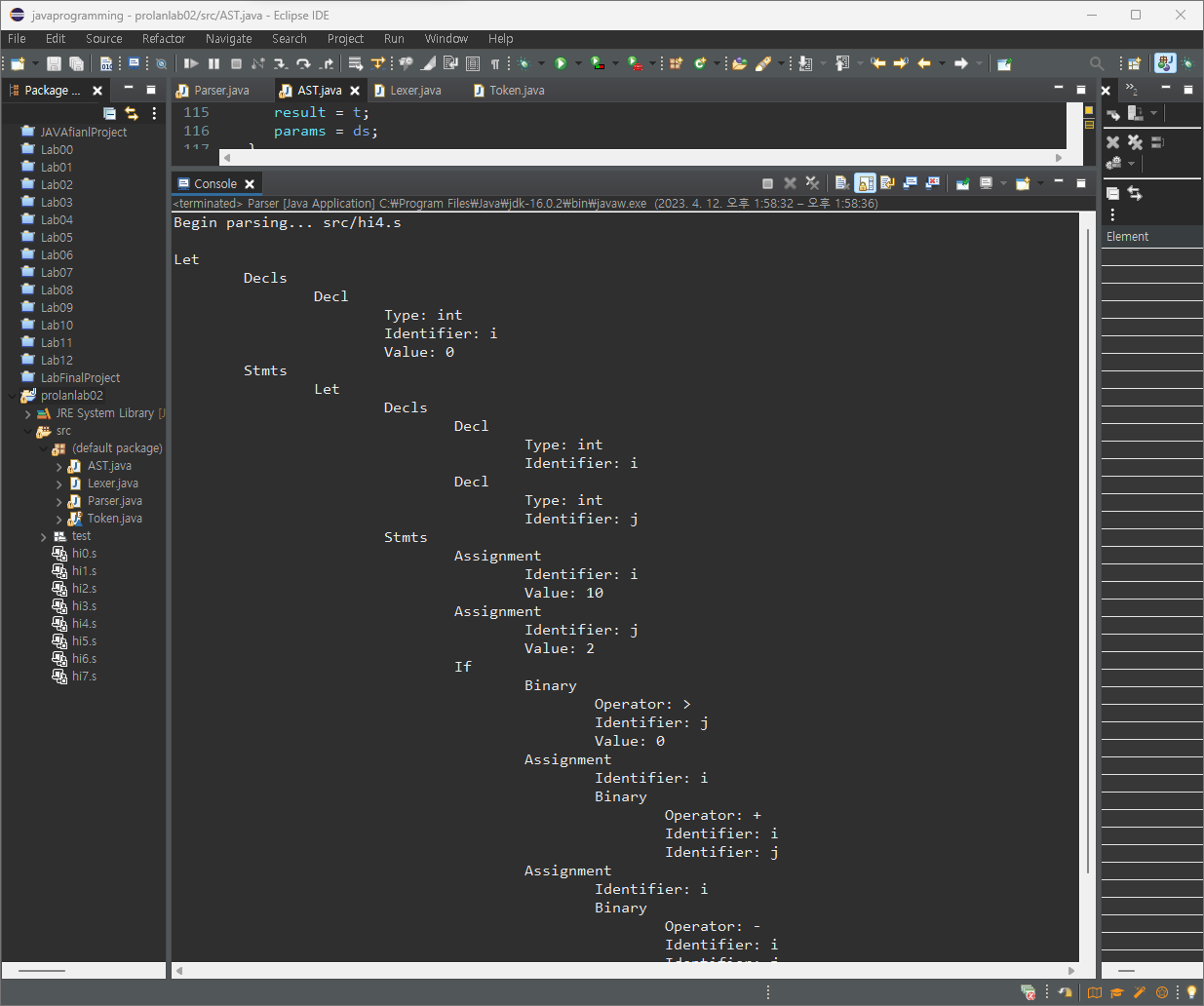
텍스트이(가) 표시된 사진

자동 생성된 설명

4) hi3.s



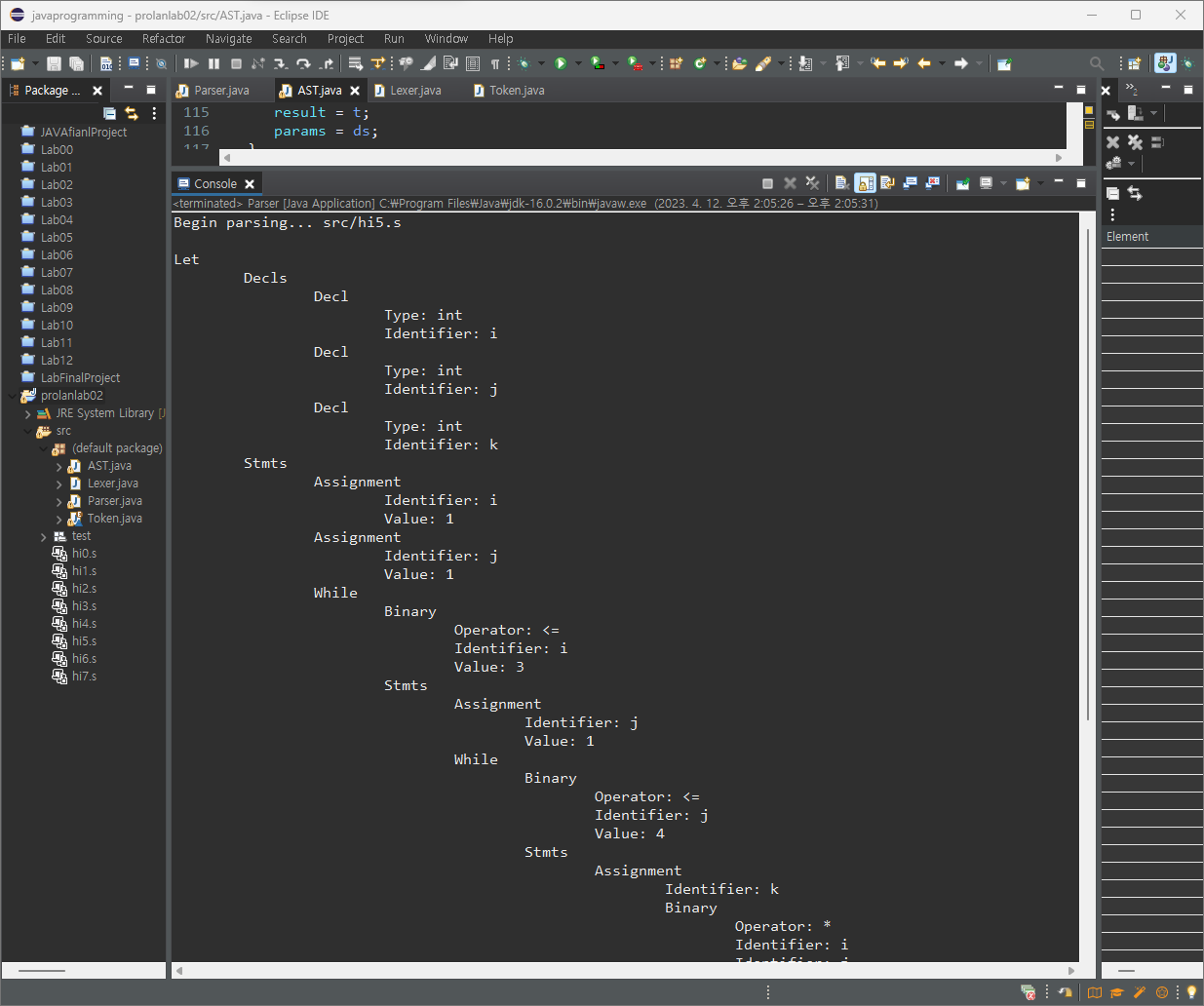
5) hi4.s

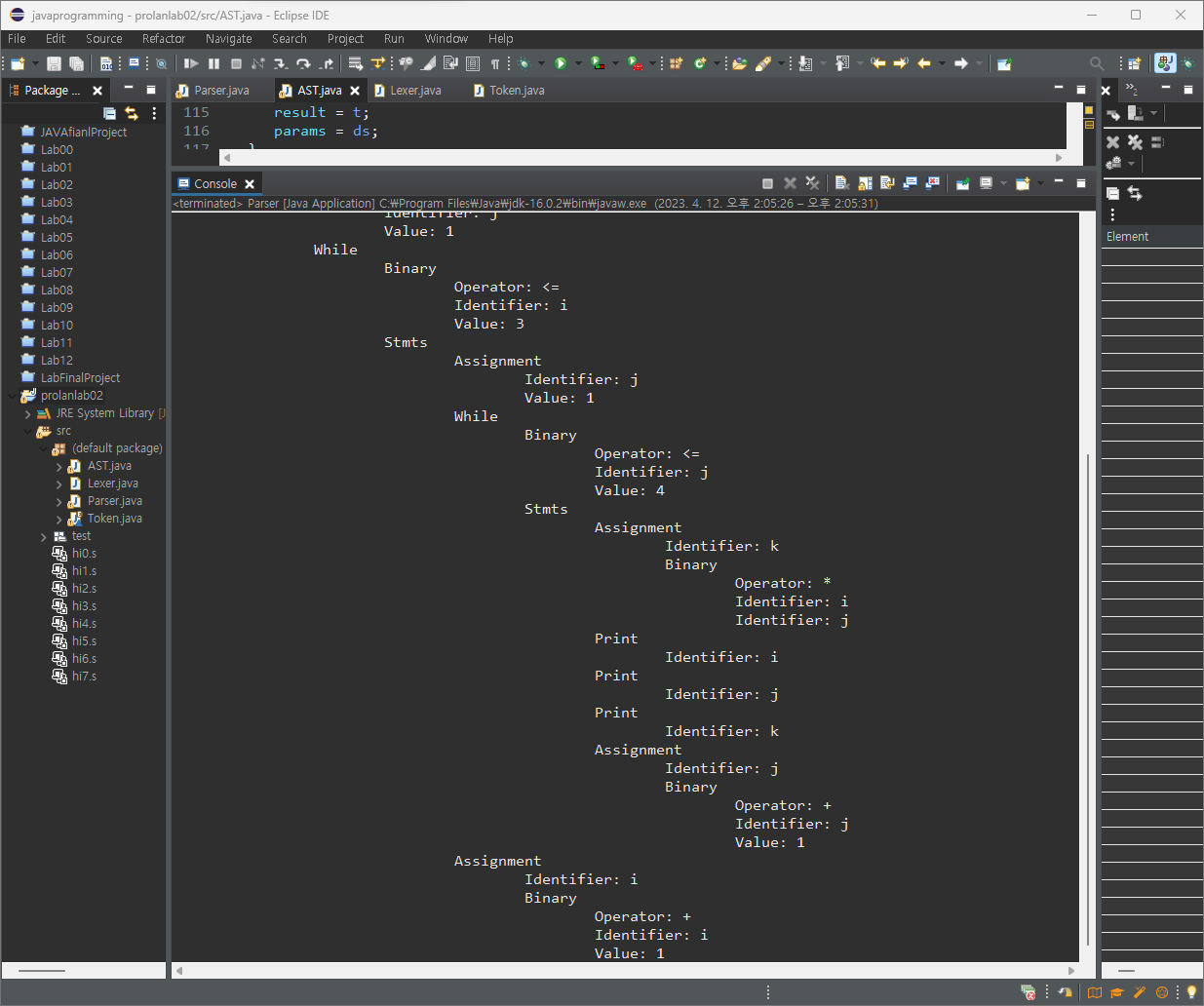


텍스트이(가) 표시된 사진

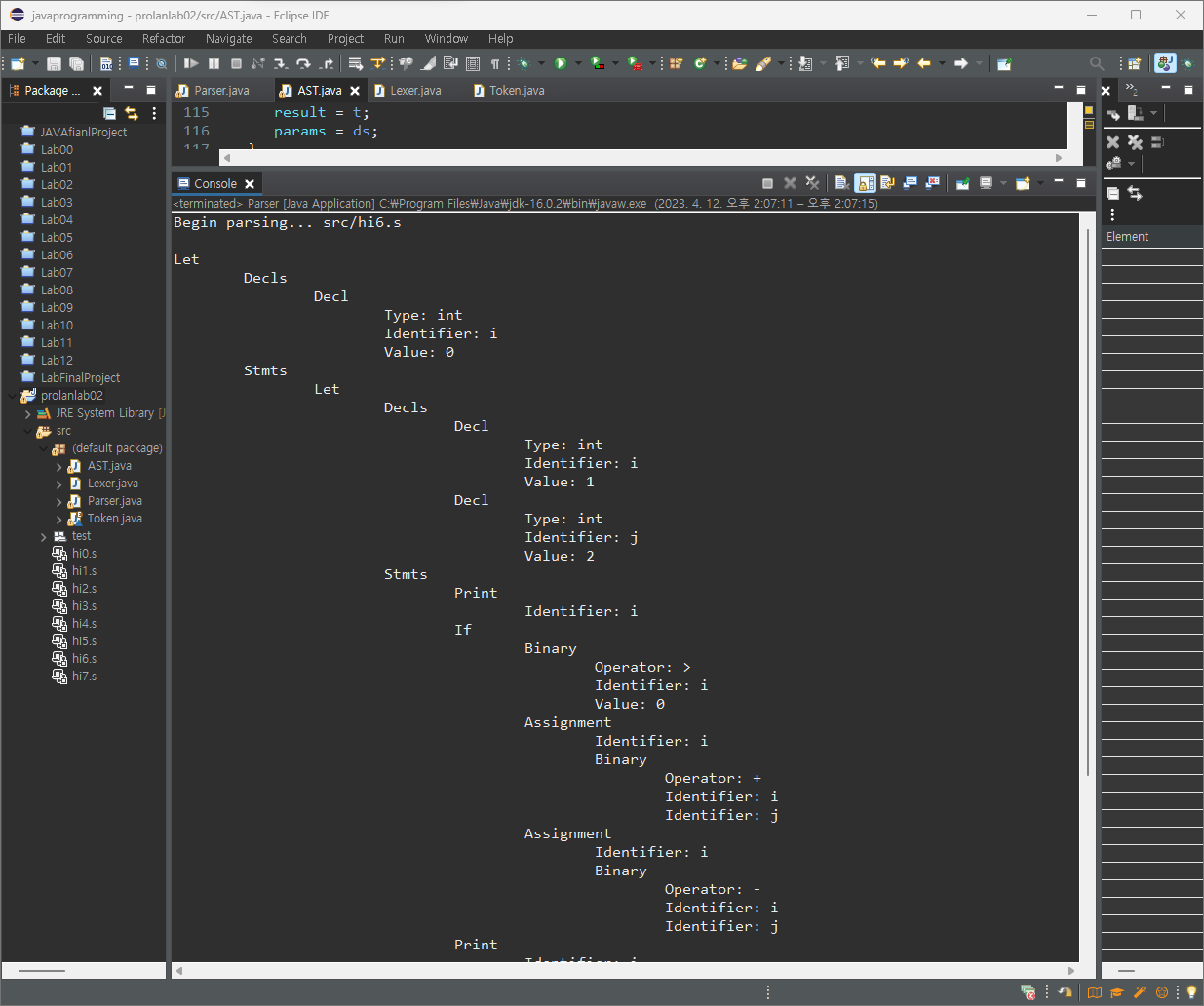
자동 생성된 설명

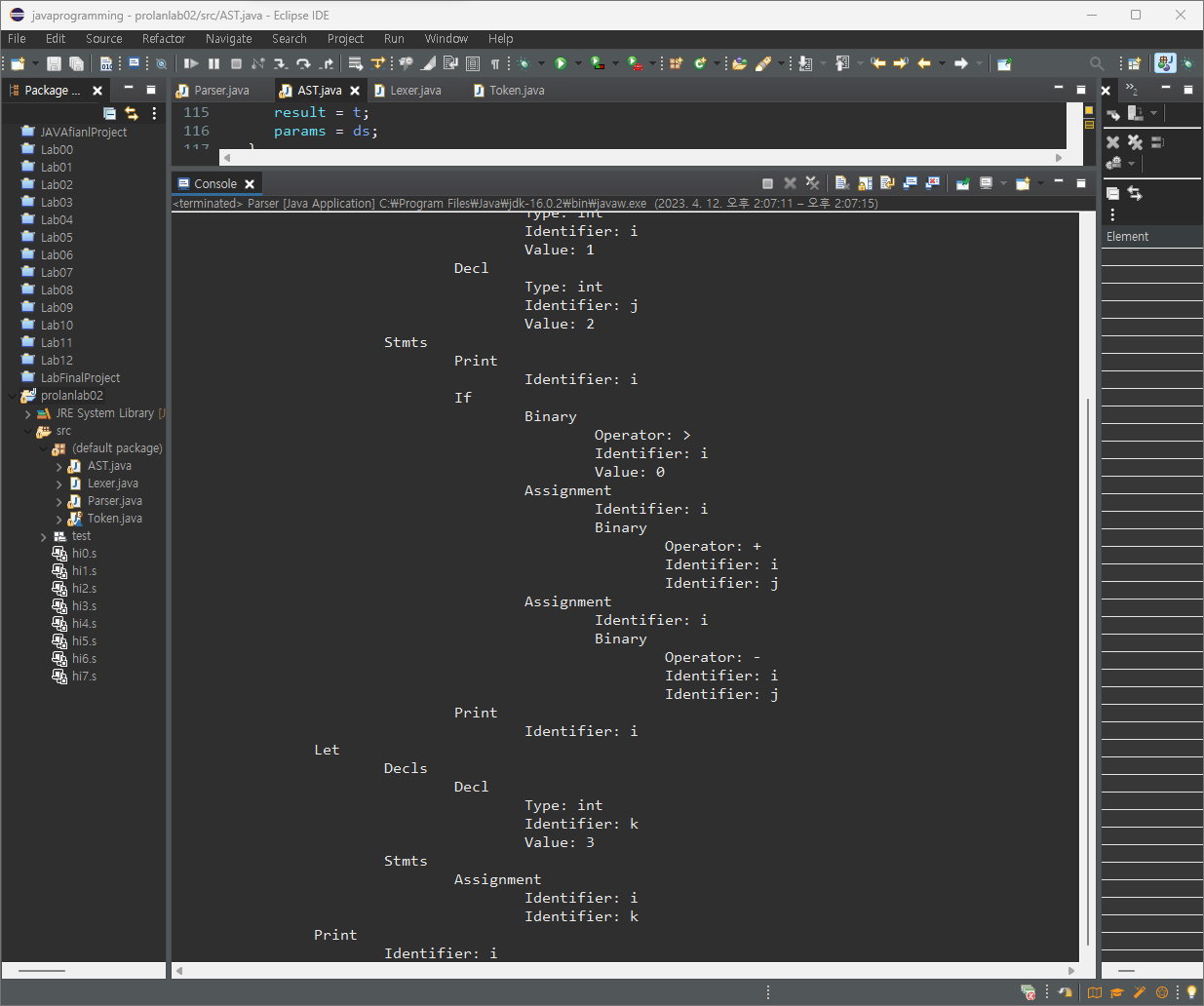
6) hi5.s



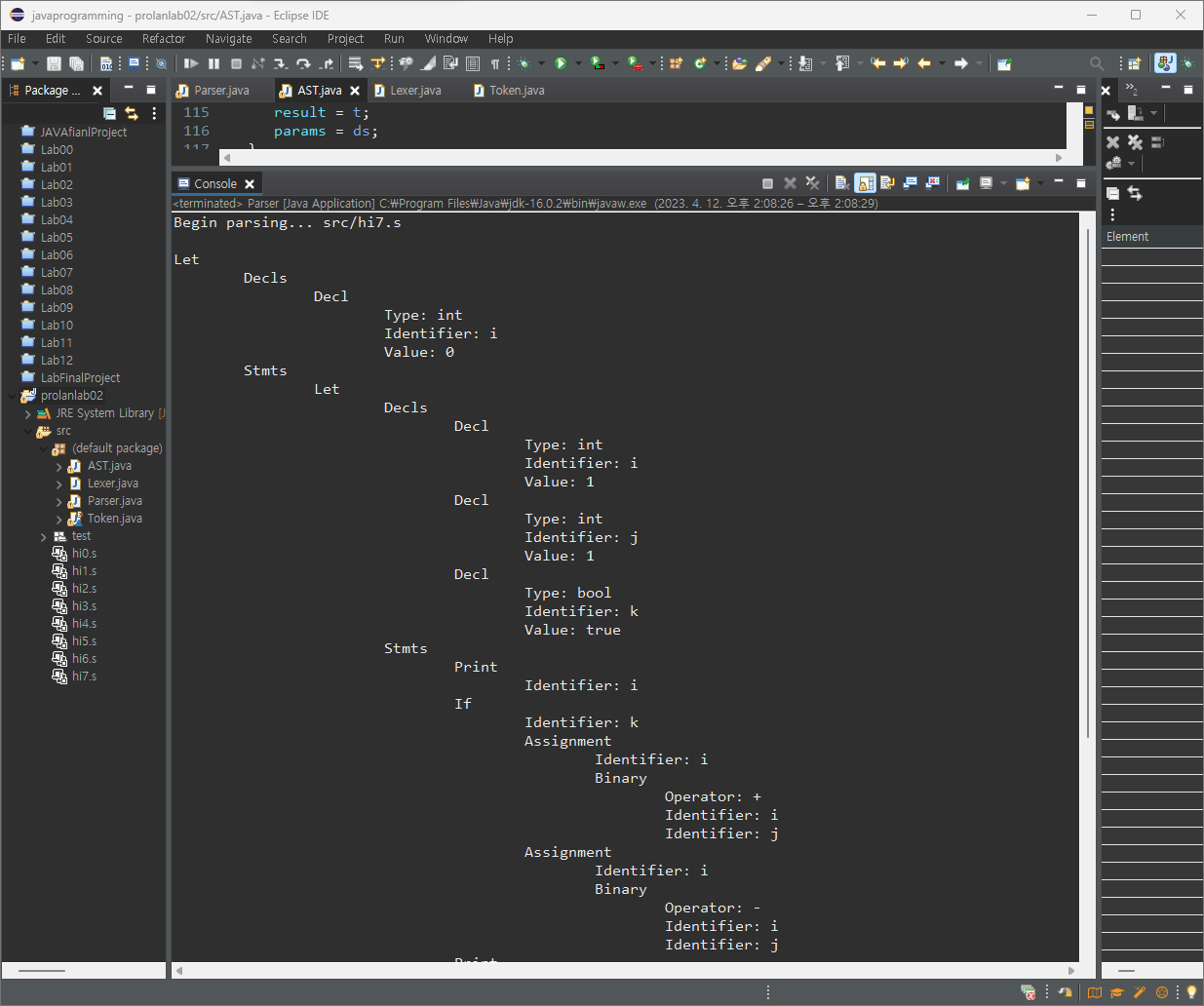


7) hi6.s





8) hi7.s



텍스트이(가) 표시된 사진

자동 생성된 설명

3. 프로그램 소스 코드 (복사-붙여넣기)

1) Parser.java

*// Parser.java*

*// Parser for language S*

*// 프로그래밍언어론 lab02 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;

**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** 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*

2) AST.java

*// AST.java*

*// AST for S*

*// 프로그래밍언어론 lab02 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)); *//추가*

}

}

**class** **Decls** **extends** ArrayList<Decl> { *//*

*// Decls = Decl\**

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

Decls() { **super**(); };

Decls(Decl d) {

**this**.add(d);

}

**public** void display(int level){ *//추가*

Indent.display(level,"Decls");

**for** (Decl decl:**this**){

decl.display(level+1);

}

}

}

**class** **Decl** **extends** Command { *//*

*// Decl = Type type; Identifier id*

Identifier id;

Expr expr = **null**;

int arraysize = 0;

Decl (String s, Type t) {

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

} *// declaration*

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

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

} *// array declaration*

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

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

} *// declaration*

**public** void display(int level){ *//추가*

**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){ *//추가*

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) { *//추가*

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

}

}

**class** **Empty** **extends** Stmt { *//done*

**public** void display(int level){ *//추가*

**super**.display(level);

}

}

**class** **Stmts** **extends** Stmt {*//*

*// Stmts = Stmt\**

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

Stmts() {

**super**();

}

Stmts(Stmt s) {

stmts.add(s);

}

**public** void display(int level){ *//추가*

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){*//추가*

**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){ *//추가*

**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){ *//추가*

**super**.display(level);

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){

**super**.display(level);

id.display(level+1); *//추가*

}

}

**class** **Print** **extends** Stmt { *//*

*// Print = Expr expr*

Expr expr;

Print (Expr e) {

expr = e;

}

**public** void display(int level){ *// 추가*

**super**.display(level);

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){ *//추가*

**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){ *//추가*

**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( ) {

*//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){ *//추가*

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 { *//*

*// 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){ *//추가*

**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){ *//추가*

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

}

}

3) Lexer.java

*// Lexer.java*

*// Lexical analyzer for S*

*// 프로그래밍언어론 lab02 소프트웨어학부 컴퓨터과학전공 2016133 이유진*

**import** **java.io.\***;

**public** **class** **Lexer** {

**private** char ch = ' ';

**private** BufferedReader input;

**private** **final** char eolnCh = '\n';

**private** **final** char eofCh = '\004';

**static** boolean interactive = **false**;

**public** Lexer (String fileName) { *// source filename*

**try** {

input = **new** BufferedReader (**new** FileReader(fileName));

}

**catch** (FileNotFoundException e) {

System.out.println("File not found: " + fileName);

System.exit(1);

}

}

**public** Lexer ( ) { *// from standard input*

input = **new** BufferedReader (**new** InputStreamReader(System.in));

}

**private** char getchar() { *// Return next char*

int c = 0;

**try** {

c = input.read();

**if** (c == -1)

c = eofCh;

} **catch**(IOException e) { System.err.println(e);}

**return** (char) c;

}

**public** Token getToken( ) { *// Return next token*

**do** {

**if** (Character.isLetter(ch)) { *// ident or keyword*

String s = "";

**do** {

s += ch;

ch = getchar();

} **while** (Character.isLetter(ch) || Character.isDigit(ch));

**return** Token.idORkeyword(s);

}

**if** (Character.isDigit(ch)) { *// number*

String s = "";

**do** {

s += ch;

ch = getchar();

} **while** (Character.isDigit(ch));

**return** Token.NUMBER.setValue(s);

}

**switch** (ch) {

**case** ' ': **case** '\t': **case** '\r':

ch = getchar();

**break**;

**case** eolnCh:

ch = getchar();

**if** (ch == '\r') *// for Windows*

ch = getchar(); *// for Windows*

**if** (ch == eolnCh && interactive)

**return** Token.EOF;

**break**;

**case** '/': *// divide*

ch = getchar();

**if** (ch != '/') **return** Token.DIVIDE;

**do** {

ch = getchar();

} **while** (ch != eolnCh);

ch = getchar();

**break**;

**case** '\"': *// string literal*

String s ="";

**while** ((ch = getchar()) != '\"')

s += ch;

ch = getchar();

**return** Token.STRLITERAL.setValue(s);

**case** eofCh:

**return** Token.EOF;

**case** '+': ch = getchar();

**return** Token.PLUS;

**case** '-': ch = getchar();

**return** Token.MINUS;

**case** '\*': ch = getchar();

**return** Token.MULTIPLY;

**case** '(': ch = getchar();

**return** Token.LPAREN;

**case** ')': ch = getchar();

**return** Token.RPAREN;

**case** '{': ch = getchar();

**return** Token.LBRACE;

**case** '}': ch = getchar();

**return** Token.RBRACE;

**case** '[': ch = getchar();

**return** Token.LBRACKET;

**case** ']': ch = getchar();

**return** Token.RBRACKET;

**case** ';': ch = getchar();

**return** Token.SEMICOLON;

**case** ',': ch = getchar();

**return** Token.COMMA;

**case** '&': ch = getchar();

**return** Token.AND;

**case** '|': ch = getchar();

**return** Token.OR;

**case** '=': ch = getchar();

**if** (ch != '=') **return** Token.ASSIGN;

**else** {ch = getchar(); **return** Token.EQUAL;}

**case** '<': ch = getchar();

**if** (ch != '=') **return** Token.LT;

**else** {ch = getchar(); **return** Token.LTEQ;}

**case** '>': ch = getchar();

**if** (ch != '=') **return** Token.GT;

**else** {ch = getchar(); **return** Token.GTEQ;}

**case** '!': ch = getchar();

**if** (ch != '=') **return** Token.NOT;

**else** {ch = getchar(); **return** Token.NOTEQ;}

} *// switch*

} **while** (**true**);

} *// next*

**public** void error (String msg) {

System.err.println("Error: " + msg);

*// System.exit(1);*

}

**static** **public** void main (String[] args) {

Lexer lexer;

**if** (args.length == 0)

lexer = **new** Lexer( );

**else**

lexer = **new** Lexer(args[0]);

Token tok = lexer.getToken( );

**while** (tok != Token.EOF) {

System.out.println(tok.toString());

tok = lexer.getToken( );

}

} *// main*

}

4) Token.java

*// Token.java*

*// 프로그래밍언어론 lab02 소프트웨어학부 컴퓨터과학전공 2016133 이유진*

**import** **java.lang.Enum**;

**enum** Token {

BOOL("bool"),CHAR("char"), ELSE("else"), FALSE("false"), FLOAT("float"),

STRING("string"), IF("if"), INT("int"), TRUE("true"), WHILE("while"),

RETURN("return"), VOID("void"), FUN("fun"), THEN("then"), LET("let"),

IN("in"), END("end"), READ("read"), PRINT("print"), DO("do"), FOR("for"),

EOF("<<EOF>>"),

LBRACE("{"), RBRACE("}"), LBRACKET("["), RBRACKET("]"),

LPAREN("("), RPAREN(")"), SEMICOLON(";"), COMMA(","),

ASSIGN("="), EQUAL("=="), LT("<"), LTEQ("<="), GT(">"),

GTEQ(">="), NOT("!"), NOTEQ("!="), PLUS("+"), MINUS("-"),

MULTIPLY("\*"), DIVIDE("/"), AND("&"), OR("|"), ID(""),

NUMBER(""), STRLITERAL("");

**private** String value;

**private** Token (String v) {

value = v;

}

**public** String value( ) { **return** value; }

**public** Token setValue(String v) {

**this**.value = v;

**return** **this**;

}

**public** **static** Token idORkeyword (String name) {

**for** (Token token : Token.values()) {

**if** (token.value().equals(name))

**return** token;

**if** (token == Token.EOF)

**break**;

}

**return** ID.setValue(name);

} *// keyword or ID*

}