컴파일러 과제-5

20192800 권대현

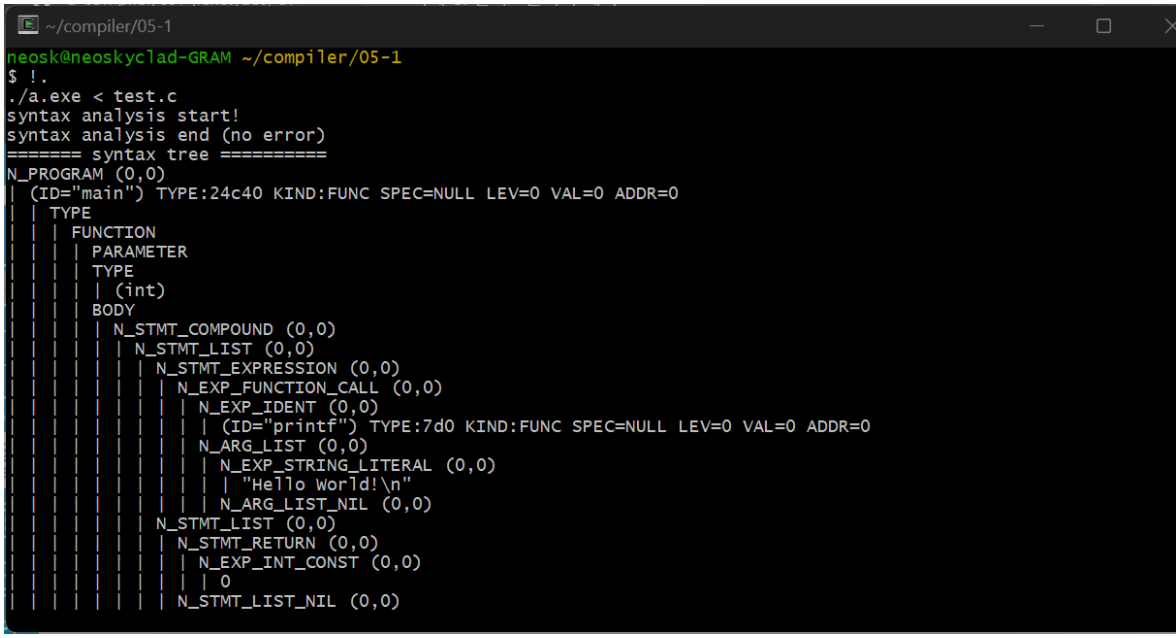
1. 과제 내용
   - 이번 컴파일러 과제는 2장의 문법(lex)과 3장에서 설명한 어휘 분석 프로그램(yacc), 4장, 5장에서 설명한 내용을 토대로 C언어의 신택스 분석기를 제작하여 C언어 코드의 문법을 검사하는 것이다.
   - 입력으로 선언문, 명령문 및 함수가 포함된 프로그램들이 주어진다.
   - 수식이 잘못된 경우 line 번호와 함께 syntax 오류가 어디서 일어났는지를 출력한다.
   - 수식이 올바를 경우 syntax tree를 출력하여 신택스 분석이 된 과정을 보여준다.
2. 문제 및 해결 방법
   - C언어 파서의 문법은 컴파일러-2장 강의노트.pdf 전체를, yacc 프로그램과 lex 문법은 컴파일러-3장 강의노트.pdf를 참고하여 작성했다.
   - Lex와 yacc 명령어를 터미널에서 사용하기 위해 bison과 flex 패키지를 설치했다.
   - 처음 yacc 프로그램을 돌렸을 때 중괄호{ }나 , 콤마를 잘못 사용하여 오류가 났었다. 이를 각각 LR RR, COMMA로 토큰명으로 바꿔줬다.
   - 2장의 강의노트 pdf를 통해 yacc 문법을 작성할 때 콜론 : 문양을 잘못 이해하여 에러가 많이 떴었다. 이를 COLON으로 토큰명으로 고쳐서 해결했다.
   - 이번 과제에서 type_identifier를 구분 짓는 함수를 제외해야 했기 때문에 따로 int, float, void, char 형의 토큰을 추가하였다.
   - 2개의 Shift/Reduce conflicts가 발생했다. 하나는 사전에 알고 있던 if – else ambiguity 문제이기에 넘어갔다.
   - 다른 하나는 unary_expression = assignment_expression (SHIFT)와 unary_expression -> constant_expression = initializer (REDUCE)에 대한 상호 충돌이었다. 이 문제는 콤마 기호의 혼란을 없애기 위해 constant_expression 쪽 문법과 initializer의 문법에서 expression을 assignment_expression으로 변경해서 생긴 문제이다. 해당 conflicts를 해결하여 shift/reduce 개수를 줄여도 parser에서 제대로 parsing을 하지 않는 문제가 발생하였기에 그대로 두었다.
   - 나머지는 reduce/reduce conflicts가 있다. Reduce/reduce의 경우 해결하기 위해 새로 예외 룰을 추가하거나 토큰 분류를 보다 상세히 해야 한다. 그러나 parser에서 문법의 오류를 검출하기엔 문제가 없음을 확인하여 그대로 두었다.
   - 3장의 강의노트 pdf를 통해 lex 파일을 만들던 중 /* */의 주석을 처리하는 정규식에 오류가 있음을 확인했다. 따라서 동일한 기능을 수행하는 정규식으로 코드를 수정했다.
   - 자르는 토큰에 대해서 yylval에 어떻게 값을 넘겨줄지 고민이 됐다. 이내 yytext를 넘겨주는 방식이 생각이 났고, 적절한 형변환을 통해 정수값, 문자값, 스트링 주소로

yytext를 넘겨줬다.

3. 테스트

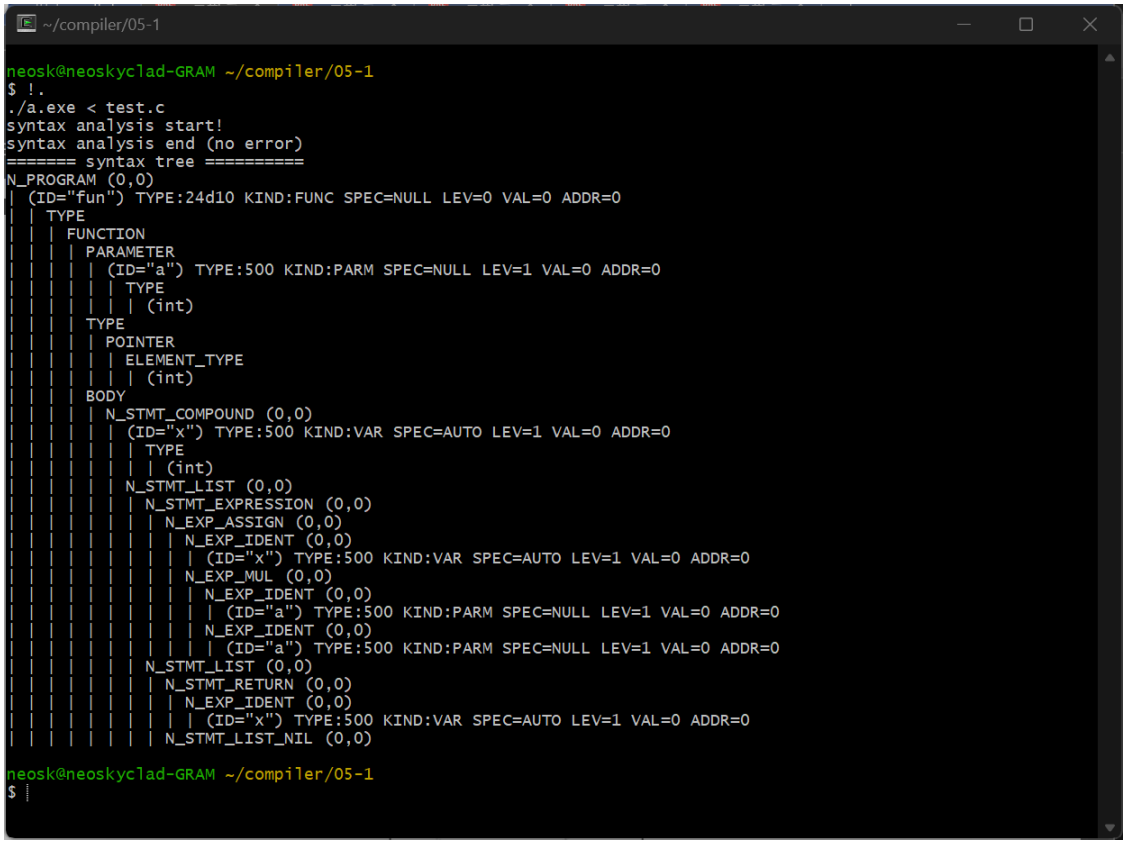
- 올바른 C언어 코드

■ Hello World 출력



◆ Main 함수에서 printf("Hello World!₩n")를 수행했다.

■ Int *fun()함수

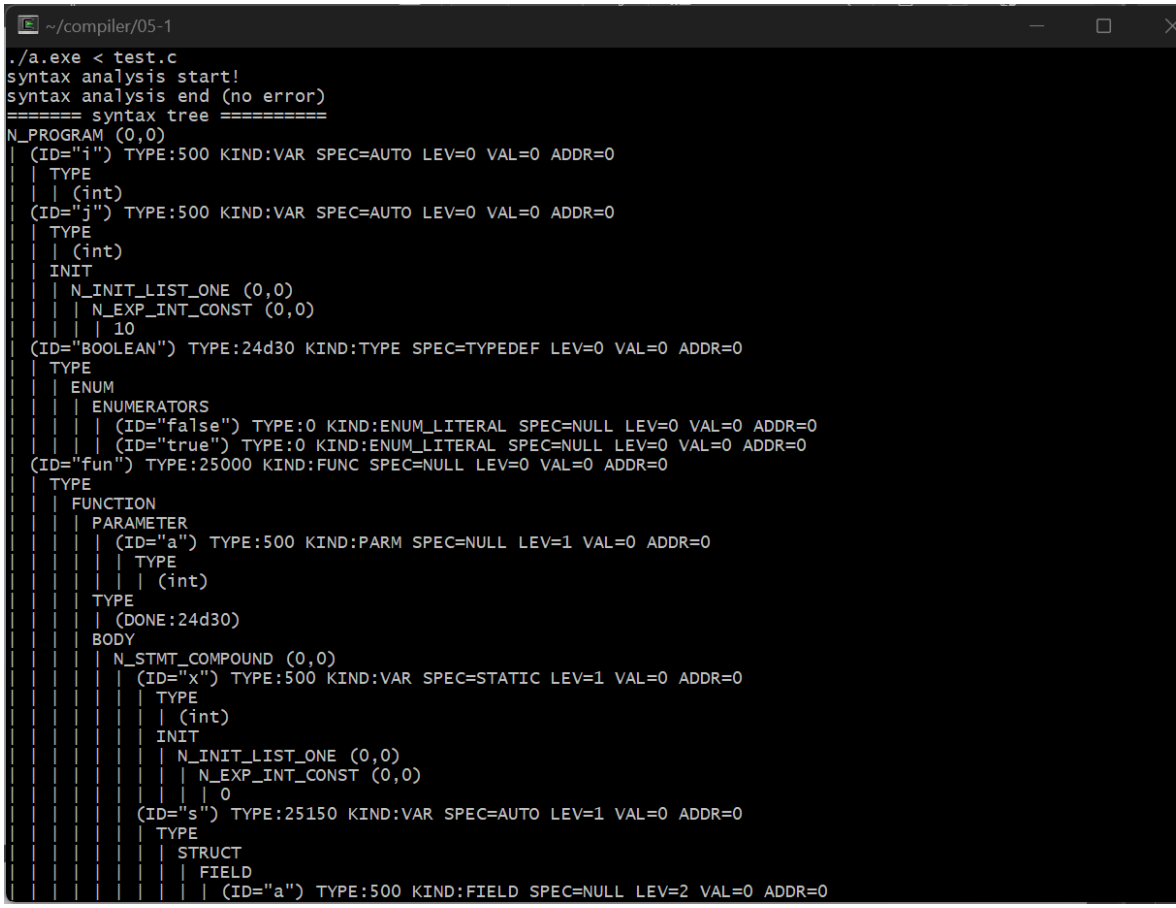


◆ 파라미터로 int a를 넘겨받고 지역변수로 int x를 선언하는 int *fun()함수를

선언한다.

- X = a*a;와 return x를 반환한다.

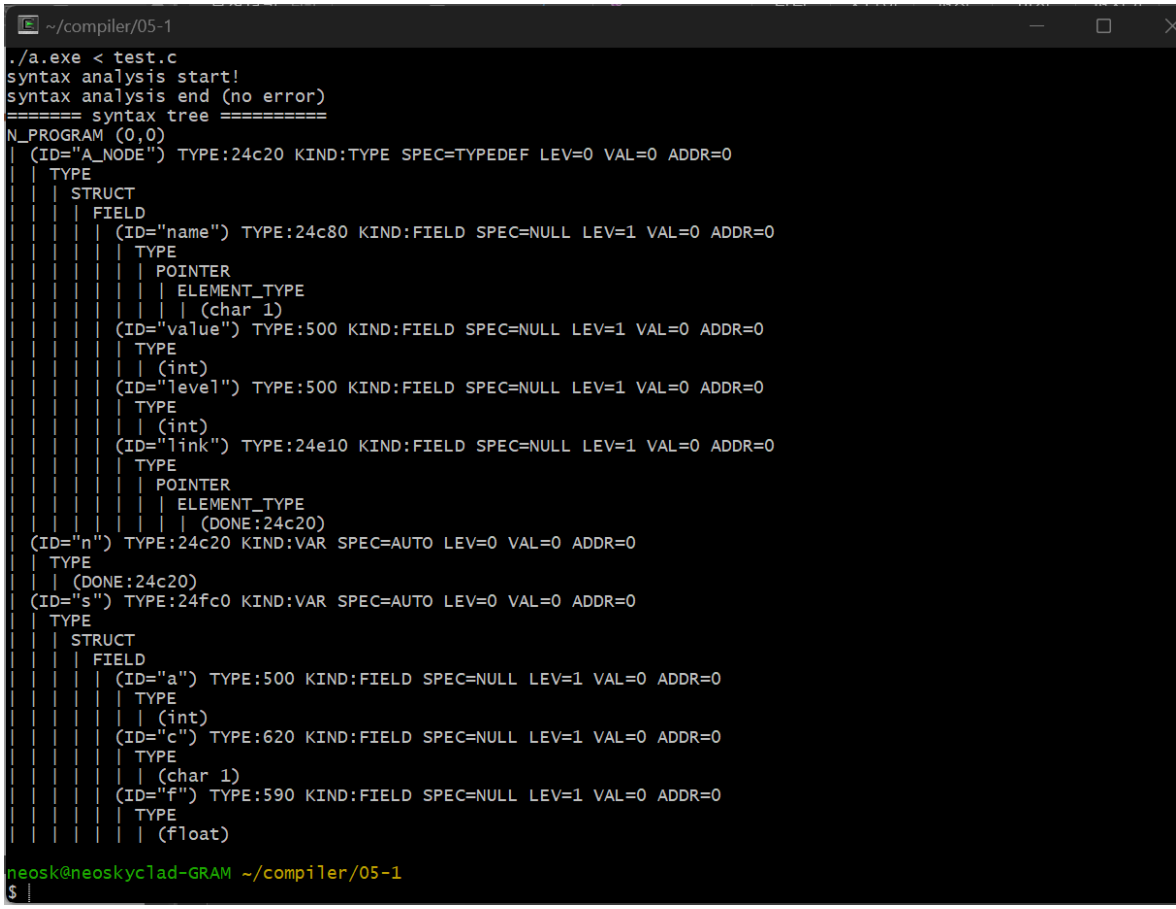
■ Enum typedef와 struct 지역변수 테스트



```
~/compiler/05-1
./a.exe < test.c
syntax analysis start!
syntax analysis end (no error)
======= syntax tree ==========
N_PROGRAM (0,0)
| (ID="i") TYPE:500 KIND:VAR SPEC=AUTO LEV=0 VAL=0 ADDR=0
| | TYPE
| | | (int)
| (ID="j") TYPE:500 KIND:VAR SPEC=AUTO LEV=0 VAL=0 ADDR=0
| | TYPE
| | | (int)
| | INIT
| | | N_INIT_LIST_ONE (0,0)
| | | | N_EXP_INT_CONST (0,0)
| | | | | 10
| (ID="BOOLEAN") TYPE:24d30 KIND:TYPE SPEC=TYPEDEF LEV=0 VAL=0 ADDR=0
| | TYPE
| | | ENUM
| | | | ENUMERATORS
| | | | | (ID="false") TYPE:0 KIND:ENUM_LITERAL SPEC=NULL LEV=0 VAL=0 ADDR=0
| | | | | (ID="true") TYPE:0 KIND:ENUM_LITERAL SPEC=NULL LEV=0 VAL=0 ADDR=0
| (ID="fun") TYPE:25000 KIND:FUNC SPEC=NULL LEV=0 VAL=0 ADDR=0
| | TYPE
| | | FUNCTION
| | | | PARAMETER
| | | | | (ID="a") TYPE:500 KIND:PARM SPEC=NULL LEV=1 VAL=0 ADDR=0
| | | | | | TYPE
| | | | | | | (int)
| | | | TYPE
| | | | | (DONE:24d30)
| | | | BODY
| | | | | N_STMT_COMPOUND (0,0)
| | | | | | (ID="x") TYPE:500 KIND:VAR SPEC=STATIC LEV=1 VAL=0 ADDR=0
| | | | | | | TYPE
| | | | | | | | (int)
| | | | | | | INIT
| | | | | | | | N_INIT_LIST_ONE (0,0)
| | | | | | | | | N_EXP_INT_CONST (0,0)
| | | | | | | | | | 0
| | | | | | (ID="s") TYPE:25150 KIND:VAR SPEC=AUTO LEV=1 VAL=0 ADDR=0
| | | | | | | TYPE
| | | | | | | | STRUCT
| | | | | | | | | FIELD
| | | | | | | | | | (ID="a") TYPE:500 KIND:FIELD SPEC=NULL LEV=2 VAL=0 ADDR=0
```

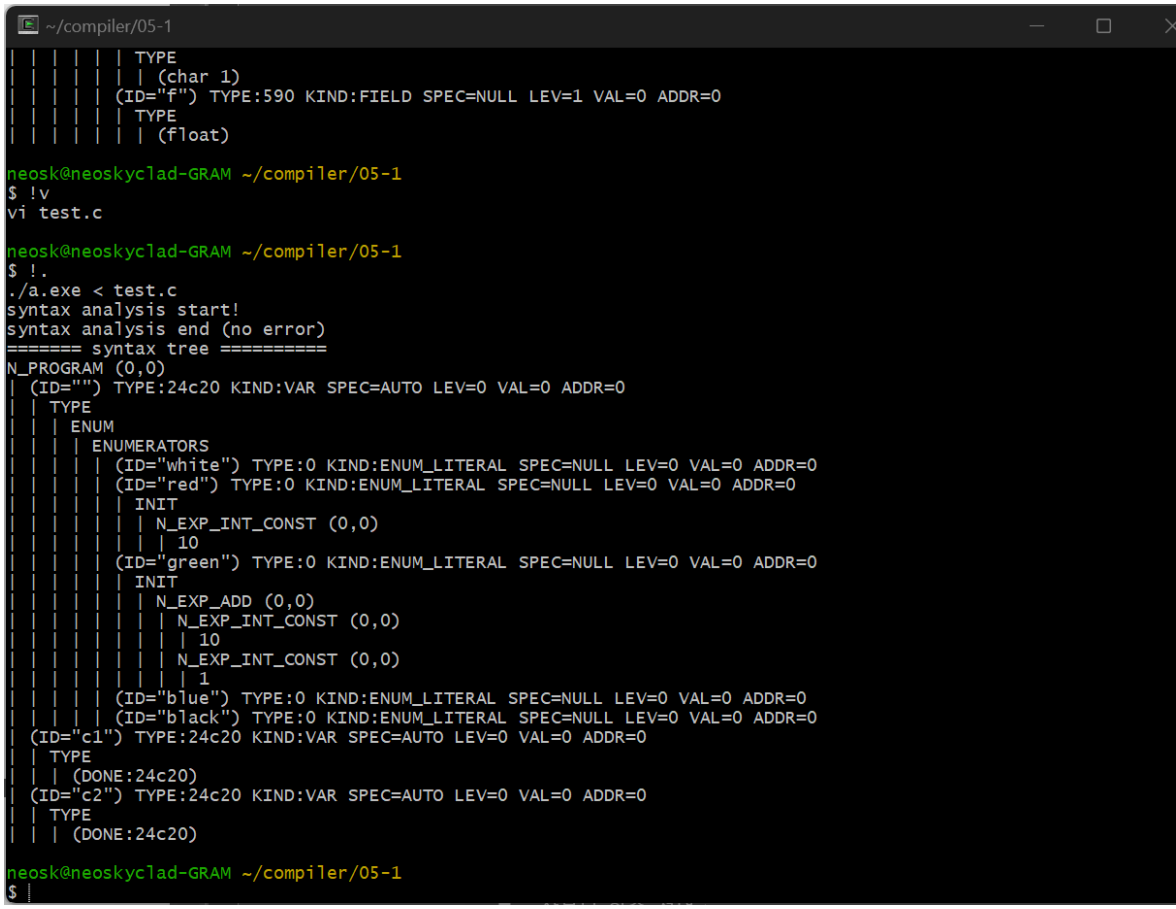
- Typedef enum {false, true} BOOLEAN으로 bool형 자료형을 정의한다.
- BOOLEAN형으로 fun() 함수를 선언했다.
  - Fun()함수 내부에는 static int x; auto struct { int a; char c; } s; float I, j, k가 존재한다.
  - 이것으로 int형, struct형, float형, char형의 선언이 동작하는 것을 확인할 수 있다.

■ Struct 구조체 선언 테스트



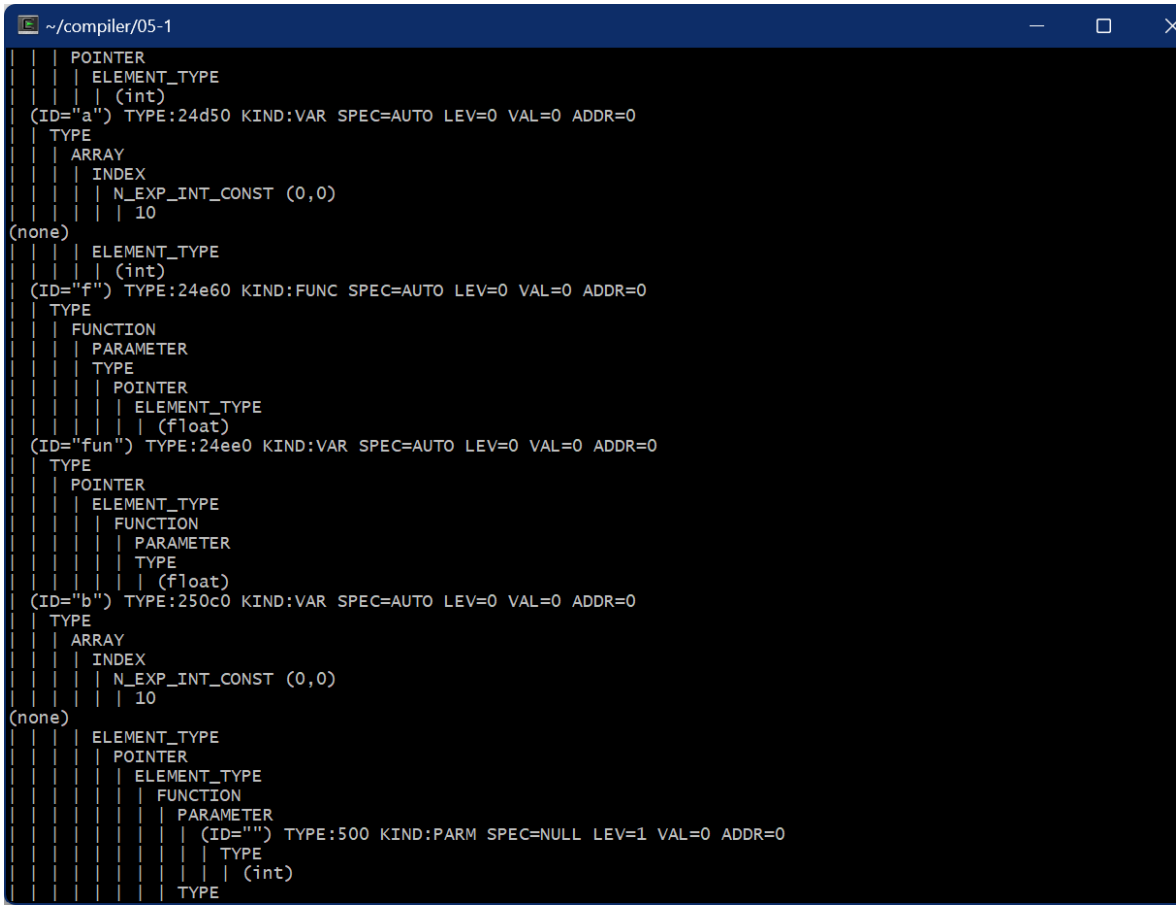
◆ Struct의 일반적인 선언, struct의 불완전선언을 테스트해봤다.

■ Enum 선언 테스트



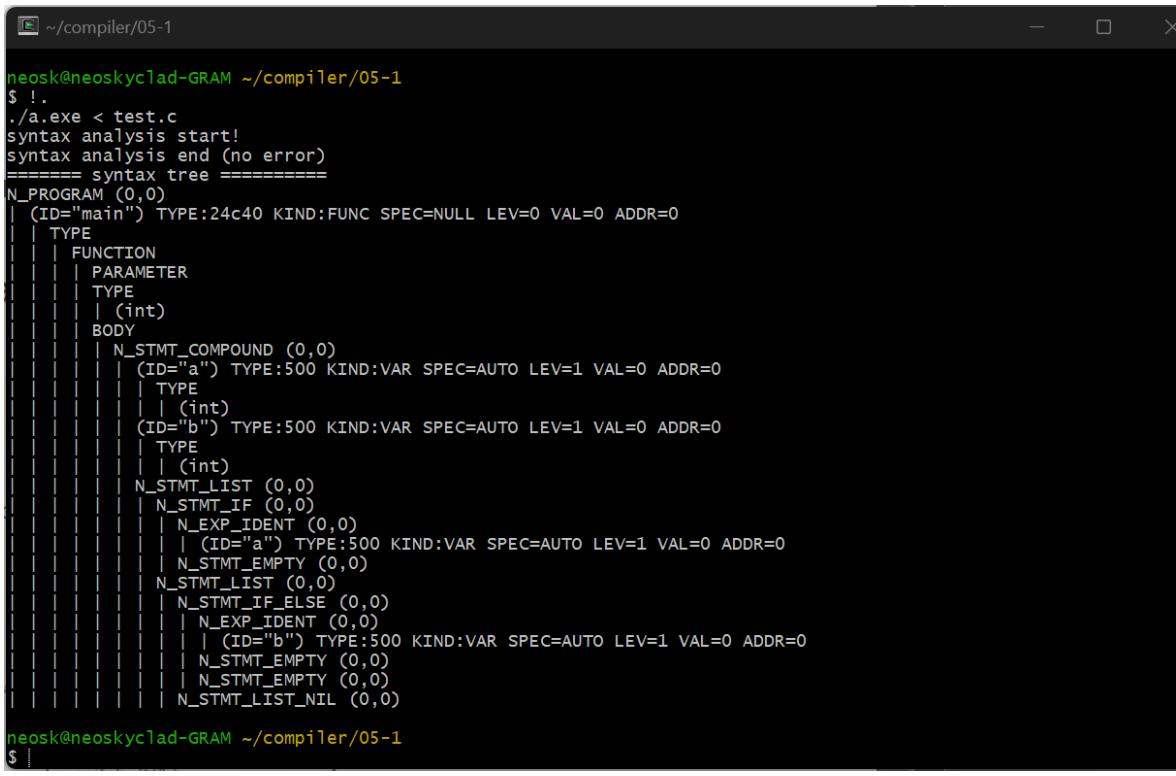
- ◆ Enum 형으로 color을 정의하고, color 내부 멤버를 초기화하는 코드를 선언했다.
- ◆ Enum color c1, c2; 를 통해 사전 정의된 color형을 사용하는 변수 선언을 테스트 했다.

■ 다양한 declarator 형태 테스트



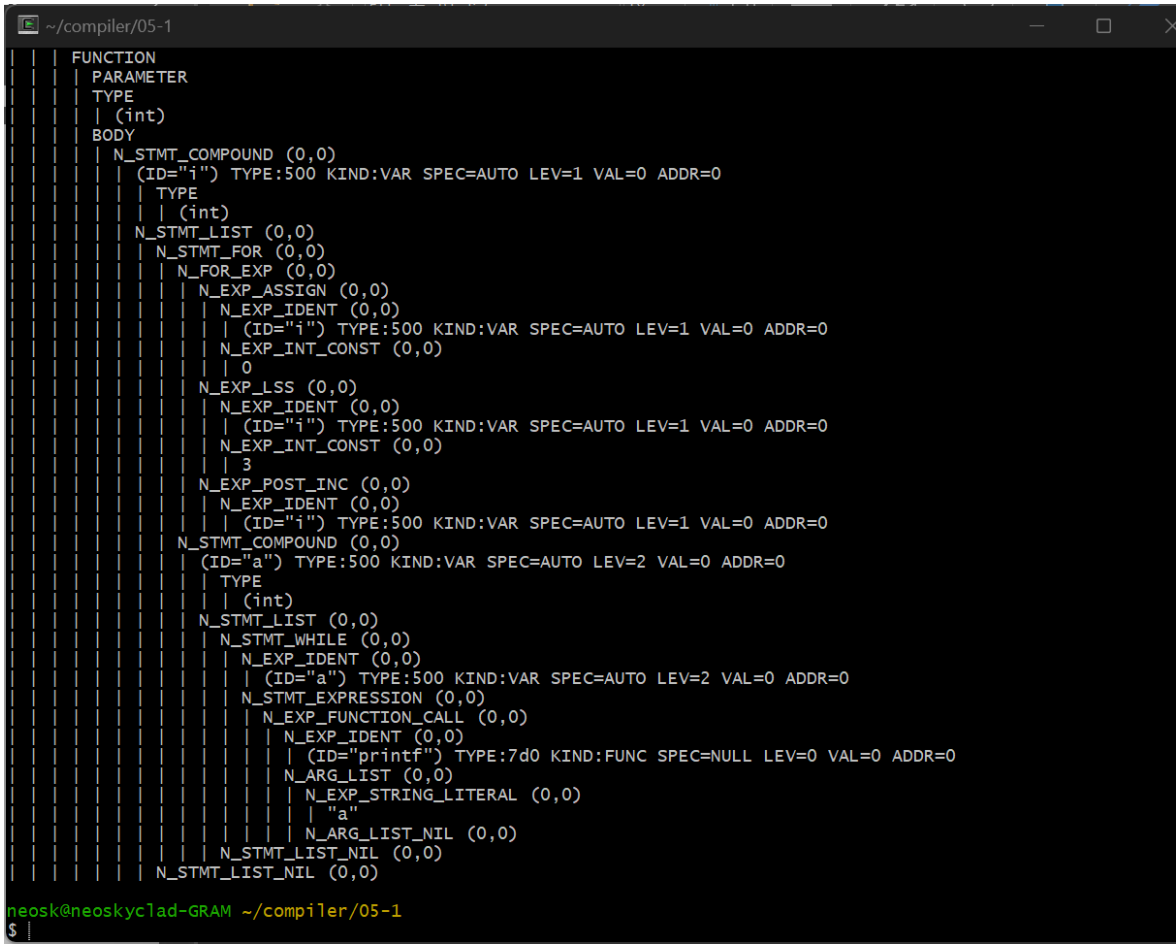
◆ Int *p, float *f(), float (*b[10])(int)와 같이 다양한 형태의 declarator를 테스트 했다.

■ If문 테스트



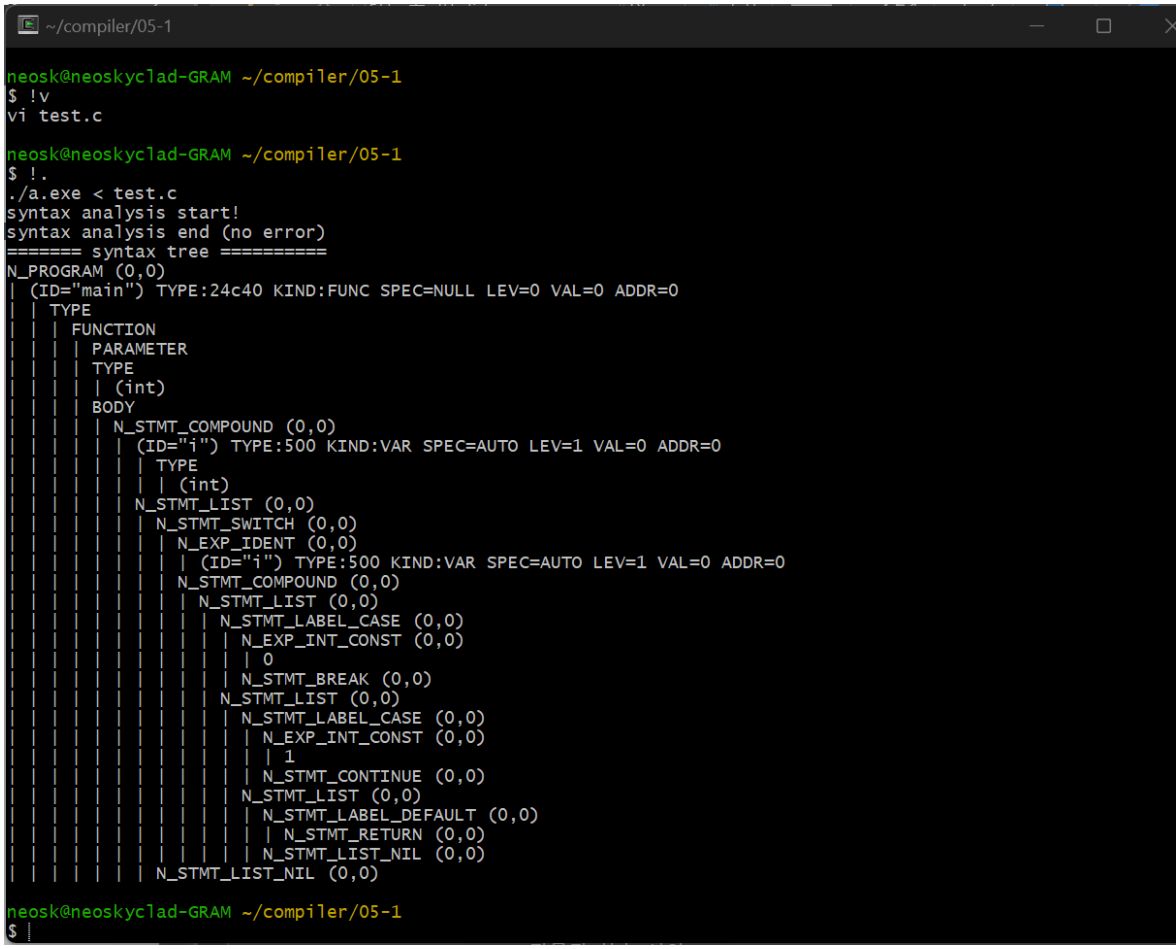
- ◆ If(expression) if(expression) else statement 형태의 if문을 테스트해봤다.
- ◆ If-else ambiguity가 있는 문장이지만, 신택스 분석에는 문제가 없다.

■ 반복문 테스트



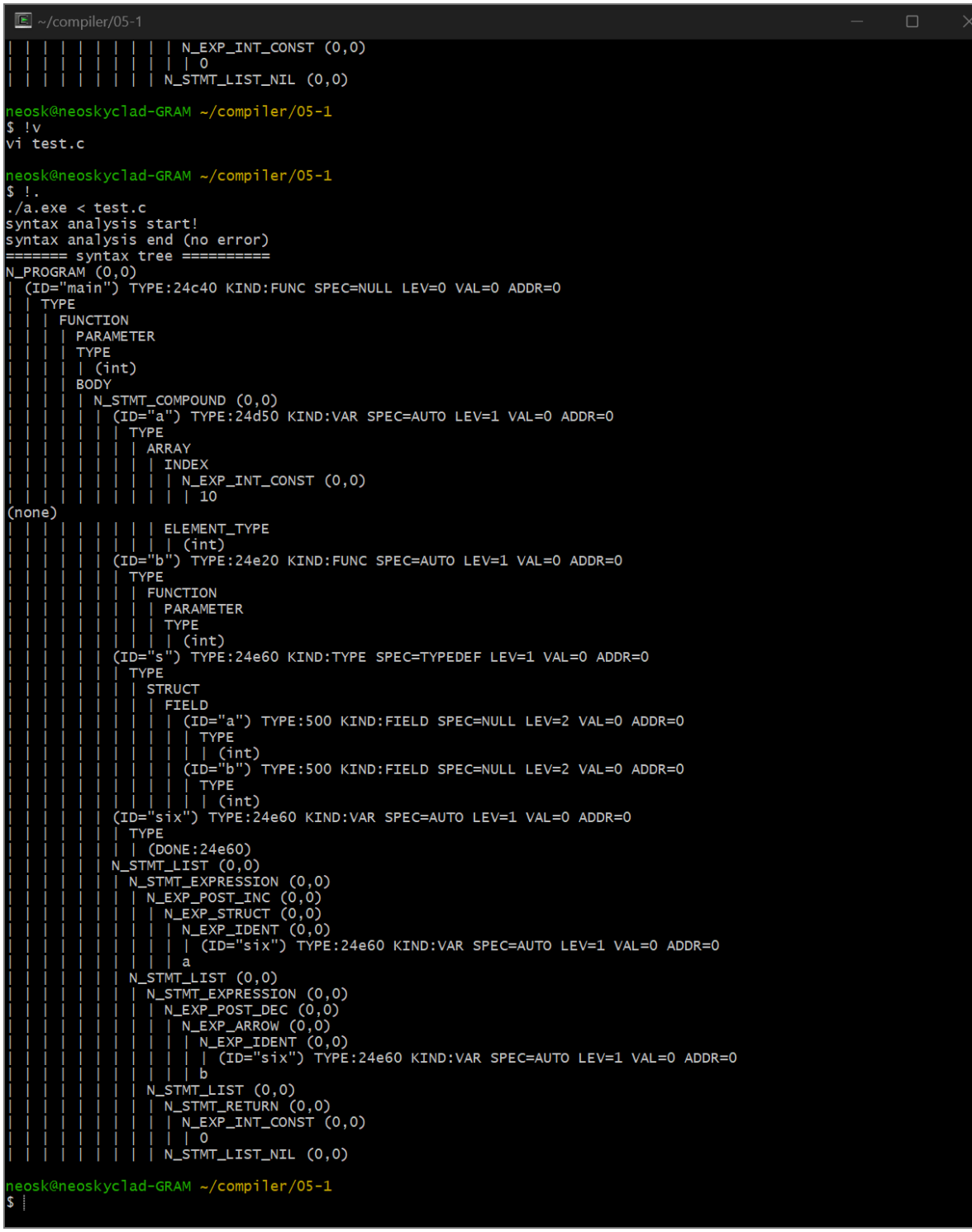
- ◆ while문과 for문을 테스트 해봤다.

■ Switch문과 jump문 테스트



- ◆ switch문은 사전에 정의된 i를 기준으로 검사된다.
  - Case 0: break;를 통해 case문과 break문의 동작을 테스트했다.
  - Case 1: continue;를 통해 continue문을 테스트했다.
  - Default: return;를 통해 default label과 return문을 테스트했다.

■ Postfix expression

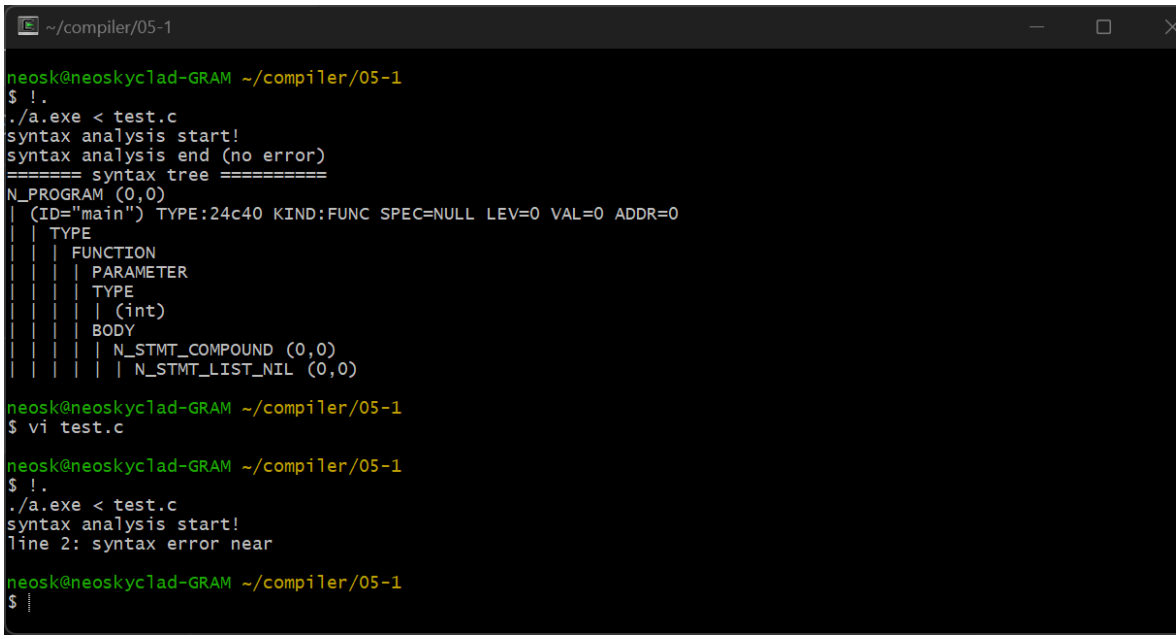


◆ 다양한 postfix 수식을 테스트했다.

- Int a[10];
- Int b();
- Six.a++;
- Six->b--;

- 잘못된 C언어 코드

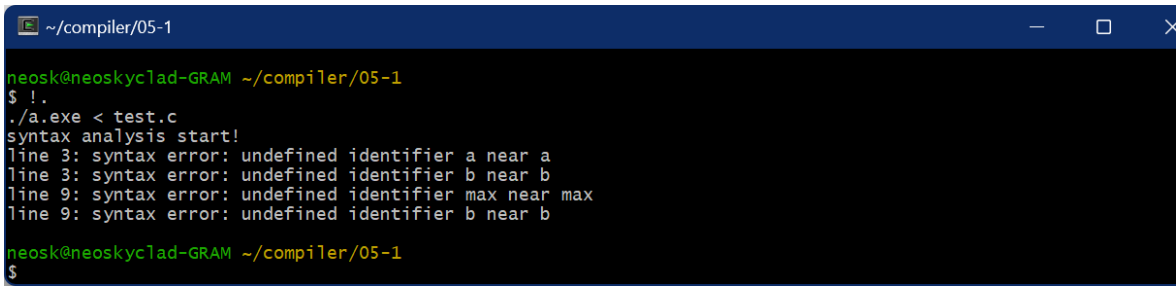
■ 잘못된 함수 선언



◆ Main 함수를 선언할 때 닫는 중괄호를 생략해봤다.

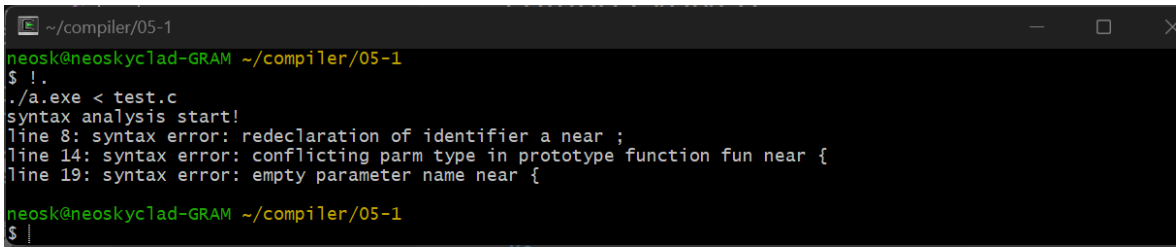
◆ Line 2에서 에러가 발생했다.

■ 선언되지 않은 변수 접근



◆ 선언되지 않은 a, b를 코드에서 사용해봤다.

■ 파라미터 관련 오류



◆ 먼저 a를 정의한 뒤, 뒤에서 한 번 더 정의하여 redeclaration 오류를 나타냈다.

◆ Int fun(int)라는 프로토타입을 선언하고 함수를 구현할 때 파라미터를 char 형으로 달리하여 conflicting parm type 오류를 나타냈다.

◆ Int fun2(int)라는 프로토타입을 선언하고 함수를 구현할 때 int fun2(int)로 파라미터의 이름을 넣지 않아서 empty parameter name 오류를 나타냈다.

4. 원시프로그램

- Lex

```
digit       [0-9]
letter      [a-zA-Z_]
delim       [ ₩t]
line        [₩n]
ws          {delim}+
%{
#define YYSTYPE_IS_DECLARED 1
typedef long YYSTYPE;
#include "y.tab.h"
#include "type.h"
#include <stdlib.h>
#include <string.h>
char *makeString(char *);
int checkIdentifier(char *);
%}

%%
{ws}        { }
{line}      { }
auto        { return (AUTO_SYM); }
break       { return (BREAK_SYM); }
case        { return (CASE_SYM); }
continue{ return (CONTINUE_SYM); }
default     { return (DEFAULT_SYM); }
```

```
do          { return (DO_SYM); }
else        { return (ELSE_SYM); }
enum        { return (ENUM_SYM); }
for         { return (FOR_SYM); }
if          { return (IF_SYM); }
return      { return (RETURN_SYM); }
sizeof      { return (SIZEOF_SYM); }
static      { return (STATIC_SYM); }
struct      { return (STRUCT_SYM); }
switch      { return (SWITCH_SYM); }
typedef     { return (TYPEDEF_SYM); }
union       { return (UNION_SYM); }
while       { return (WHILE_SYM); }
goto        { return (GOTO_SYM); }

"₩+₩+"      { return (PLUSPLUS); }
"₩-₩-"      { return (MINUSMINUS); }
"₩->"       { return (ARROW); }
"<"         { return (LSS); }
">"         { return (GTR); }
"<="        { return (LEQ); }
">="        { return (GEQ); }
"=="        { return (EQL); }
"!="        { return (NEQ); }
"&&"        { return (AMPAMP); }
"||"        { return (BARBAR); }
```

```
"<<"      { return (LSH); }

">>"      { return (RSH); }

"₩.₩.₩." { return (DOTDOTDOT); }

"₩("      { return (LP); }

"₩)"      { return (RP); }

"₩["      { return (LB); }

"₩]"      { return (RB); }

"₩{"      { return (LR); }

"₩}"      { return (RR); }

"₩:"      { return (COLON); }

"₩."      { return (PERIOD); }

"₩,"      { return (COMMA); }

"₩!"      { return (EXCL); }

"₩*"      { return (STAR); }

"₩/"      { return (SLASH); }

"₩%"      { return (PERCENT); }

"₩&"      { return (AMP); }

"₩;"      { return (SEMICOLON); }

"₩+"      { return (PLUS); }

"₩-"      { return (MINUS); }

"₩="      { return (ASSIGN); }

"₩~"      { return (NOT); }

"₩^"      { return (XOR); }

"₩|"      { return (BAR); }

"₩?"      { return (QUESTION); }
```

```
"const"   { return (CONST_SYM); }


{digit}+  { yylval = atoi(yytext); return (INTEGER_CONSTANT); }

{digit}+₩.{digit}+ { yylval = (long)makeString(yytext); return (FLOAT_CONSTANT); }

{letter}({letter}|{digit})* { return (checkIdentifier(yytext)); }

₩"([^"₩n]|₩₩["₩n])*₩" { yylval = (long)makeString(yytext); return (STRING_LITERAL); }

₩'([^'₩n]|₩'₩')₩' { yylval = *(yytext + 1); return (CHARACTER_CONSTANT); }

₩/₩*([^*]|₩*+[^*/])*₩*₩/ { }

"//"[^₩n]* { }

%%


char *makeString(char *s)

{

        char *tmp;

        tmp = malloc(strlen(s) + 1);

        strcpy(tmp, s);


        return (tmp);

}


int checkIdentifier(char *s)

{

        char *table[] = {"int", "float", "char", "void"};

        for (int i = 0; i < 4; i++)

        {

                if(strcmp(s, table[i]) == 0)
```

```
                {
                        yylval = makeString(s);
                        return (TYPE_IDENTIFIER);
                }
        }
        yylval = *s;
        return (IDENTIFIER);
}
```

- Yacc

```
%{
#define YYSTYPE_IS_DECLARED 1
typedef long YYSTYPE;

#include "type.h"
#include "func.h"
#include <stdio.h>
#include <stdlib.h>

int yyerror(char*);
int yylex();
%}

%token IDENTIFIER TYPE_IDENTIFIER AUTO_SYM BREAK_SYM CASE_SYM CONTINUE_SYM
DEFAULT_SYM DO_SYM ELSE_SYM ENUM_SYM FOR_SYM IF_SYM RETURN_SYM SIZEOF_SYM
STATIC_SYM STRUCT_SYM SWITCH_SYM TYPEDEF_SYM UNION_SYM WHILE_SYM GOTO_SYM
PLUSPLUS MINUSMINUS ARROW LSS GTR LEQ GEQ EQL NEQ AMPAMP BARBAR LSH RSH
```

```
DOTDOTDOT LP RP LB RB LR RR COLON PERIOD COMMA EXCL STAR SLASH PERCENT AMP
SEMICOLON PLUS MINUS ASSIGN NOT XOR BAR QUESTION INTEGER_CONSTANT
FLOAT_CONSTANT STRING_LITERAL CHARACTER_CONSTANT CONST_SYM

%start program


%%

program : translation_unit { root=makeNode(N_PROGRAM, NIL, $1, NIL);
checkForwardReference(); }

        ;

translation_unit : external_declaration { $$=$1; }

                 | translation_unit external_declaration { $$=linkDeclaratorList($1, $2); }

;

external_declaration : function_definition { $$=$1; }

                     | declaration { $$=$1; }

;


function_definition : declaration_specifiers declarator { $$=setFunctionDeclaratorSpecifier($2, $1); }
compound_statement { $$=setFunctionDeclaratorBody($3, $4); current_id=$2; }

                    | declarator { $$=setFunctionDeclaratorSpecifier($1, makeSpecifier(int_type,
0)); } compound_statement { $$=setFunctionDeclaratorBody($2, $3); current_id=$1; }

;

declaration : declaration_specifiers init_declarator_list SEMICOLON
{ $$=setDeclaratorListSpecifier($2, $1); }

            ;

declaration_specifiers : type_specifier { $$=makeSpecifier($1, 0); }

                       | storage_class_specifier { $$=makeSpecifier(0, $1); }

| type_specifier declaration_specifiers { $$=updateSpecifier($2, $1, 0); }
```

```
| storage_class_specifier declaration_specifiers {$$=updateSpecifier($2, 0, $1); }

;

storage_class_specifier : AUTO_SYM { $$=S_AUTO; } | STATIC_SYM { $$=S_STATIC; } | TYPEDEF_SYM
{$$=S_TYPEDEF; }

                                ;

init_declarator_list : init_declarator { $$=$1; }

                        | init_declarator_list COMMA init_declarator { $$=linkDeclaratorList($1, $3); }

;

init_declarator : declarator {$$=$1;}

                | declarator ASSIGN initializer {$$=setDeclaratorInit((A_ID*)$1, (A_NODE*)$3); }

;

type_specifier : struct_specifier {$$=$1;}

                | enum_specifier {$$=$1;}

| TYPE_IDENTIFIER {$$=$1;}

;


struct_specifier : struct_or_union IDENTIFIER {$$=setTypeStructOrEnumIdentifier($1, $2,
ID_STRUCT); } LR { $$=current_id; current_level++; } struct_declaration_list RR
{checkForwardReference(); $$=setTypeField($3, $6); current_level--; current_id=$5; }

                | struct_or_union {$$=makeType($1); } LR {$$=current_id; current_level++; }
struct_declaration_list RR {checkForwardReference(); $$=setTypeField($2, $5); current_level--;
current_id=$4; }

| struct_or_union IDENTIFIER {$$=getTypeOfStructOrEnumRefIdentifier($1, $2, ID_STRUCT); }

;

struct_or_union : STRUCT_SYM {$$=T_STRUCT; }

                | UNION_SYM {$$=T_UNION; }

;
```

```
struct_declaration_list : struct_declaration {$$=$1;}
                        | struct_declaration_list struct_declaration {$$=linkDeclaratorList($1, $2); }
;
struct_declaration : type_specifier struct_declarator_list SEMICOLON
{$$=setStructDeclaratorListSpecifier($2, $1); }
                   ;
struct_declarator_list : struct_declarator {$$=$1;}
                       | struct_declarator_list COMMA struct_declarator
{$$=linkDeclaratorList($1, $3);}
;
struct_declarator : declarator {$$=$1;}
;

enum_specifier : ENUM_SYM IDENTIFIER {$$=setTypeStructOrEnumIdentifier(T_ENUM, $2,
ID_ENUM); } LR enumerator_list RR {$$=setTypeField($3, $5); }
               | ENUM_SYM {$$=makeType(T_ENUM);} LR enumerator_list RR
{$$=setTypeField($2, $4);}
| ENUM_SYM IDENTIFIER {$$=getTypeOfStructOrEnumRefIdentifier(T_ENUM, $2, ID_ENUM); }
;
enumerator_list : enumerator {$$=$1;}
                | enumerator_list COMMA enumerator {$$=linkDeclaratorList($1, $3);}
;
enumerator : IDENTIFIER {$$=setDeclaratorKind(makeIdentifier($1), ID_ENUM_LITERAL);}
           | IDENTIFIER {$$=setDeclaratorKind(makeIdentifier($1), ID_ENUM_LITERAL);} ASSIGN
constant_expression {$$=setDeclaratorInit($2, $4);}
;
```

```
declarator : pointer direct_declarator {$$=setDeclaratorElementType($2, $1);}
           | direct_declarator {$$=$1;}
;

constant_expression_opt : /* empty */ {$$=NIL;}
                        | constant_expression {$$=$1;}
;
parameter_type_list_opt : /* empty */ {$$=NIL;}
                        | parameter_type_list {$$=$1;}
;

pointer : STAR {$$=makeType(T_POINTER);}
        | STAR pointer {$$=setTypeElementType($2, makeType(T_POINTER));}
;

direct_declarator : IDENTIFIER {$$=makeIdentifier($1);}
                  | LP declarator RP {$$=$2;}
| direct_declarator LB constant_expression_opt RB {$$=setDeclaratorElementType($1,
setTypeExpr(makeType(T_ARRAY), $3));}
| direct_declarator LP {$$=current_id; current_level++;} parameter_type_list_opt RP
{checkForwardReference(); current_id=$3; current_level--; $$=setDeclaratorElementType($1,
setTypeField(makeType(T_FUNC), $4));}
;

parameter_type_list : parameter_list {$$=$1;}
                    | parameter_list COMMA DOTDOTDOT {$$=linkDeclaratorList($1,
setDeclaratorKind(makeDummyIdentifier(), ID_PARM));}
```

```
;

parameter_list : parameter_declaration {$$=$1;}

                | parameter_list COMMA parameter_declaration {$$=linkDeclaratorList($1, $3);}

;

parameter_declaration : declaration_specifiers declarator
{$$=setParameterDeclaratorSpecifier($2,$1);}

                      | declaration_specifiers abstract_declarator_opt
{$$=setParameterDeclaratorSpecifier(setDeclaratorType(makeDummyIdentifier(), $2), $1);}

;

abstract_declarator_opt : /* empty */ {$$=NIL;}

                        | abstract_declarator {$$=$1;}

;

abstract_declarator : pointer {$$=makeType(T_POINTER);}

                    | direct_abstract_declarator {$$=$1;}

| pointer direct_abstract_declarator {$$=setTypeElementType($2, makeType(T_POINTER));}

;

direct_abstract_declarator : LP abstract_declarator RP {$$=$2;}

                           | LB constant_expression_opt RB
{$$=setTypeExpr(makeType(T_ARRAY), $2);}

| LP parameter_type_list_opt RP {$$=setTypeExpr(makeType(T_FUNC), $2);}

| direct_abstract_declarator LB constant_expression_opt RB {$$=setTypeElementType($1,
setTypeExpr(makeType(T_ARRAY), $3));}

| direct_abstract_declarator LP parameter_type_list_opt RP {$$=setTypeElementType($1,
setTypeExpr(makeType(T_FUNC),$3));}


initializer : constant_expression {$$=(A_NODE*)makeNode(N_INIT_LIST_ONE, NIL, $1, NIL);}

            | LR initializer_list RR {$$=$2;}
```

```
| LR initializer_list COMMA RR {$$=$2;}
;
initializer_list : initializer {$$=makeNode(N_INIT_LIST, $1, NIL, makeNode(N_INIT_LIST_NIL, NIL, NIL, NIL));}
                 | initializer_list COMMA initializer {$$=makeNodeList(N_INIT_LIST,$1,$3);}
;

statement : labeled_statement {$$=$1;}
          | compound_statement {$$=$1;}
| expression_statement {$$=$1;}
| selection_statement {$$=$1;}
| iteration_statement {$$=$1;}
| jump_statement {$$=$1;}
;
labeled_statement : CASE_SYM constant_expression COLON statement {$$=makeNode(N_STMT_LABEL_CASE, $2, NIL, $4);}
                  | DEFAULT_SYM COLON statement {$$=makeNode(N_STMT_LABEL_DEFAULT, NIL, $3, NIL);}
;

compound_statement : LR {$$=current_id; current_level++; } declaration_list statement_list RR {checkForwardReference(); $$=makeNode(N_STMT_COMPOUND, $3, NIL, $4); current_id=$2; current_level--;}
                   ;
declaration_list : /* empty */ {$$=NIL;}
                 | declaration_list declaration {$$=linkDeclaratorList($1, $2);}
;
```

```
statement_list : /* empty */ {$$=NIL;}
                | statement_list statement {$$=makeNodeList(N_STMT_LIST, $1, $2);}
;

expression_statement : SEMICOLON {$$=makeNode(N_STMT_EMPTY, NIL, NIL, NIL);}
                     | expression SEMICOLON {$$=makeNode(N_STMT_EXPRESSION, NIL, $1, NIL);}

selection_statement : IF_SYM LP expression RP statement {$$=makeNode(N_STMT_IF, $3, NIL, $5);}
                    | IF_SYM LP expression RP statement ELSE_SYM statement {$$=makeNode(N_STMT_IF_ELSE, $3, $5, $7);}
| SWITCH_SYM LP expression RP statement {$$=makeNode(N_STMT_SWITCH, $3, NIL, $5);}
;

iteration_statement : WHILE_SYM LP expression RP statement {$$=makeNode(N_STMT_WHILE, $3, NIL, $5);}
                    | DO_SYM statement WHILE_SYM LP expression RP SEMICOLON {$$=makeNode(N_STMT_DO, $2, NIL, $5);}
| FOR_SYM LP expression_opt SEMICOLON expression_opt SEMICOLON expression_opt RP statement {$$=makeNode(N_STMT_FOR, $3, NIL, $5);}
;

expression_opt : /* empty */ {$$=NIL;}
                | expression {$$=$1;}
;

jump_statement : RETURN_SYM expression_opt SEMICOLON {$$=makeNode(N_STMT_RETURN, NIL, $2, NIL);}
                | CONTINUE_SYM SEMICOLON {$$=makeNode(N_STMT_CONTINUE, NIL, NIL, NIL);}
| BREAK_SYM SEMICOLON {$$=makeNode(N_STMT_BREAK, NIL, NIL, NIL);}
;
```

```
primary_expression : IDENTIFIER {$$=makeNode(N_EXP_IDENT, NIL, getIdentifierDeclared($1), NIL);}
                   | INTEGER_CONSTANT {$$=makeNode(N_EXP_INT_CONST, NIL, $1, NIL);}
| FLOAT_CONSTANT {$$=makeNode(N_EXP_FLOAT_CONST, NIL, $1, NIL);}
| CHARACTER_CONSTANT {$$=makeNode(N_EXP_CHAR_CONST, NIL, $1, NIL);}
| STRING_LITERAL {$$=makeNode(N_EXP_STRING_LITERAL, NIL, $1, NIL);}
| LP expression RP {$$=$2;}
;
postfix_expression : primary_expression {$$=$1;}
                   | postfix_expression LB expression RB {$$=makeNode(N_EXP_ARRAY, $1, NIL,
$3);}
| postfix_expression LP arg_expression_list_opt RP {$$=makeNode(N_EXP_FUNCTION_CALL, $1, NIL,
$3);}
| postfix_expression PERIOD IDENTIFIER {$$=makeNode(N_EXP_STRUCT, $1, NIL, $3);}
| postfix_expression ARROW IDENTIFIER {$$=makeNode(N_EXP_ARROW, $1, NIL, $3);}
| postfix_expression PLUSPLUS {$$=makeNode(N_EXP_POST_INC, NIL, $1, NIL);}
| postfix_expression MINUSMINUS {$$=makeNode(N_EXP_POST_DEC, NIL, $1, NIL);}
;
arg_expression_list_opt : /* empty */ {$$=makeNode(N_ARG_LIST_NIL, NIL, NIL, NIL);}
                        | arg_expression_list {$$=$1;}
;
arg_expression_list : assignment_expression {$$=makeNode(N_ARG_LIST, $1, NIL,
makeNode(N_ARG_LIST_NIL, NIL, NIL, NIL));}
                    | arg_expression_list COMMA assignment_expression
{$$=makeNodeList(N_ARG_LIST, $1, $3);}
;
```

```
unary_expression : postfix_expression {$$=$1;}
                 | PLUSPLUS unary_expression {$$=makeNode(N_EXP_PRE_INC,NIL,$2,NIL);}
| MINUSMINUS unary_expression {$$=makeNode(N_EXP_PRE_DEC,NIL,$2,NIL);}
| AMP cast_expression {$$=makeNode(N_EXP_AMP,NIL,$2,NIL);}
| STAR cast_expression {$$=makeNode(N_EXP_STAR,NIL,$2,NIL);}
| EXCL cast_expression {$$=makeNode(N_EXP_NOT,NIL,$2,NIL);}
| MINUS cast_expression {$$=makeNode(N_EXP_MINUS,NIL,$2,NIL);}
| PLUS cast_expression {$$=makeNode(N_EXP_PLUS,NIL,$2,NIL);}
| SIZEOF_SYM unary_expression {$$=makeNode(N_EXP_SIZE_EXP,NIL,$2,NIL);}
| SIZEOF_SYM LP type_name RP {$$=makeNode(N_EXP_SIZE_TYPE,NIL,$3,NIL);}
;
cast_expression : unary_expression {$$=$1;}
                | LP type_name RP cast_expression {$$=makeNode(N_EXP_CAST,$2,NIL, $4);}
;
type_name : declaration_specifiers abstract_declarator {$$=setTypeNameSpecifier($2,$1);}
;

multiplicative_expression : cast_expression {$$=$1;}
                          | multiplicative_expression STAR cast_expression
{$$=makeNode(N_EXP_MUL, $1, NIL, $3);}
| multiplicative_expression SLASH cast_expression {$$=makeNode(N_EXP_DIV, $1, NIL, $3);}
| multiplicative_expression PERCENT cast_expression {$$=makeNode(N_EXP_MOD, $1, NIL, $3);}
;
additive_expression : multiplicative_expression {$$=$1;}
                    | additive_expression PLUS multiplicative_expression
{$$=makeNode(N_EXP_ADD,$1,NIL,$3);}
```

```
| additive_expression MINUS multiplicative_expression {$$=makeNode(N_EXP_SUB,$1,NIL,$3);}
;
shift_expression : additive_expression {$$=$1;}
;

relational_expression : shift_expression {$$=$1;}
                      | relational_expression LSS shift_expression {$$=makeNode(N_EXP_LSS, $1,
NIL, $3);}
| relational_expression GTR shift_expression {$$=makeNode(N_EXP_GTR, $1, NIL, $3);}
| relational_expression LEQ shift_expression {$$=makeNode(N_EXP_LEQ, $1, NIL, $3);}
| relational_expression GEQ shift_expression {$$=makeNode(N_EXP_GEQ, $1, NIL, $3);}
;
equality_expression : relational_expression {$$=$1;}
                    | equality_expression EQL relational_expression
{$$=makeNode(N_EXP_EQL,$1,NIL,$3);}
| equality_expression NEQ relational_expression {$$=makeNode(N_EXP_NEQ,$1,NIL,$3);}
;

AND_expression : equality_expression {$$=$1;}
;
exclusive_OR_expression : AND_expression {$$=$1;}
;
inclusive_OR_expression : exclusive_OR_expression {$$=$1;}
                        | inclusive_OR_expression BAR exclusive_OR_expression
{$$=makeNode(N_EXP_OR, $1, NIL, $3);}
;
```

```
logical_AND_expression : inclusive_OR_expression {$$=$1;}
                       | logical_AND_expression AMPAMP inclusive_OR_expression
{$$=makeNode(N_EXP_AND,$1,NIL, $3);}
;
logical_OR_expression : logical_AND_expression {$$=$1;}
                      | logical_OR_expression BARBAR logical_AND_expression
{$$=makeNode(N_EXP_OR, $1, NIL, $3);}
;
conditional_expression : logical_OR_expression {$$=$1;}
;

assignment_expression : conditional_expression {$$=$1;}
                      | unary_expression ASSIGN assignment_expression
{$$=makeNode(N_EXP_ASSIGN, $1, NIL, $3);}
;
comma_expression : assignment_expression {$$=$1;}
                 ;
expression : comma_expression {$$=$1;}
           ;
constant_expression : assignment_expression {$$=$1;}
                    ;
%%
extern int syntax_err;
extern A_NODE *root;

void main() {
```

```
        initialize();
        yyparse();
        if (syntax_err) exit(1);
        print_ast(root);
        printf("success!₩n");
        exit(0);
}
extern char *yytext;
int yyerror(char *s) { printf("%s near %s₩n", s, yytext); exit(1); }
int yywrap() { return (1); }
```