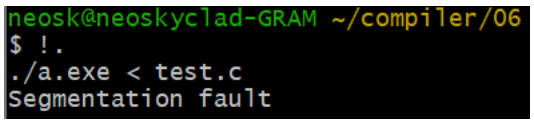
컴파일러 과제-6

20192800 권대현

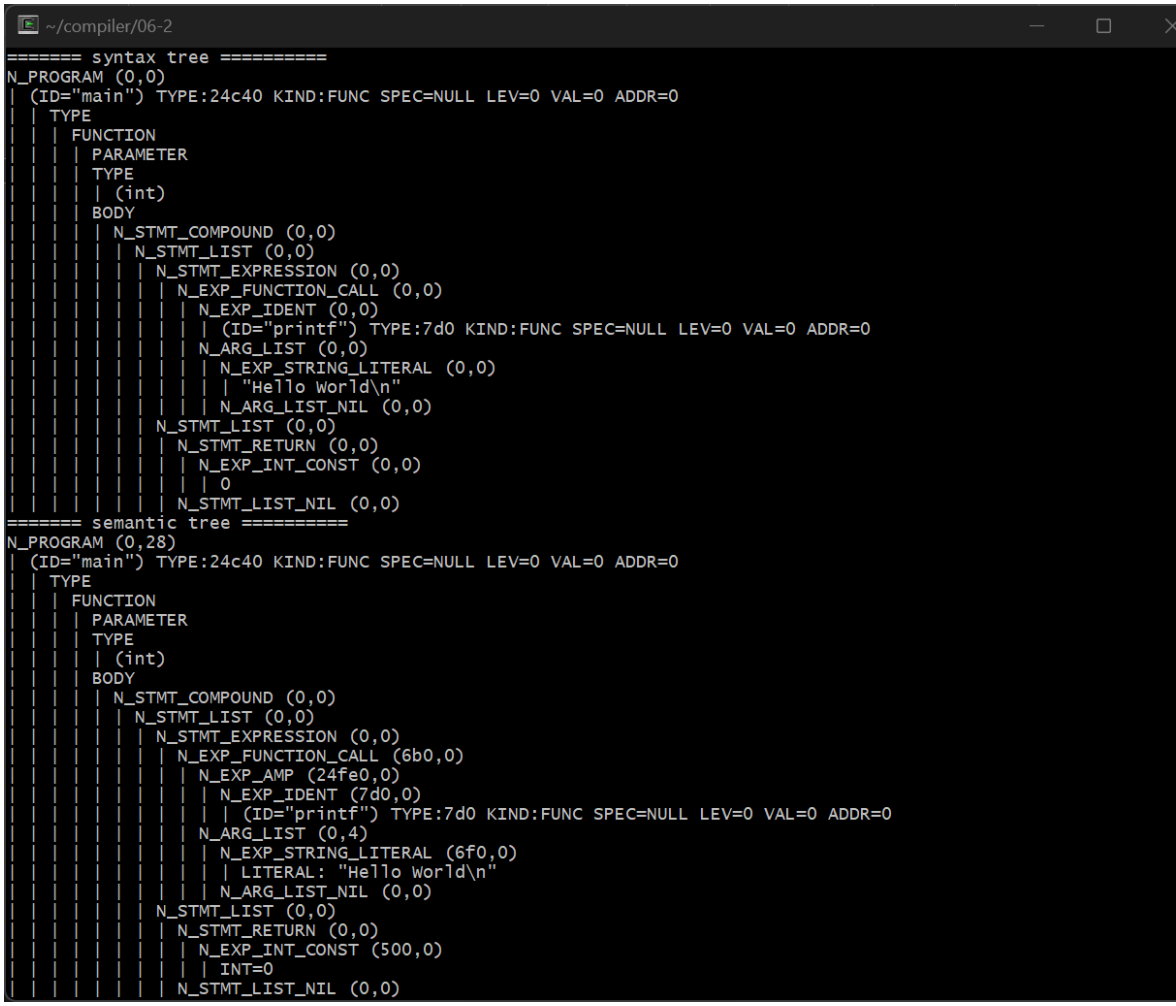
1. 과제 내용
   - 이번 컴파일러 과제는 과제-5에서 구현한 C언어 신택스 분석기에 더불어 7장에서 설명한 시멘틱 분석기를 완성하여 실험하는 것이다.
   - 입력으로 다양한 선언문과 명령문을 포함하는 프로그램들이 주어진다.
   - 수식이 잘못된 경우 line 번호와 함께 syntax 오류가 어디서 일어났는지를 출력한다.
   - 수식이 올바를 경우 syntax tree를 출력하여 신택스 분석이 된 과정을 보여준다. 그 다음으로 semantic tree를 출력하여 시멘틱 분석이 된 과정을 보여준다.
2. 문제 및 해결 방법
   - 시멘틱 분석기 구현 문제
     - 시멘틱 분석기를 위한 정보는 컴파일러-7장 강의노트.pdf를 참고하여 완성했다.
     - 시멘틱 분석기의 구현부 코드가 길기 때문에 따로 semantic.c로 파일을 만들어서 작성하였고, 빌드할 때 해당 C코드를 포함시켰다.
     - 1-10 페이지와 1-11 페이지부터 시멘틱 분석을 위한 함수 선언부가 나오지만 생략된 부분은 어떤 것을 채워야 할지 몰랐었다. 이후, IsIntType이나 IsVoidType이 사용되는 구현부를 보고 해당 함수들을 추가로 채워 넣었다.
   - 과제 구현 제외 사항
     - 수업 중에 상의한 바로는 구문에서 초기화, switch, case, goto가 있고, 구조체와 union 자료형은 구현하지 않기로 했었다. 이에 시멘틱 분석에서 해당 함수는 구현하지 않았다.
   - 실행 문제
     - 빌드 후 생성된 실행 파일 a.exe에 c코드를 테스트하기 위해 ./a.exe < test.c와 같이 Shell 명령어를 입력했으나, Segmentation fault 문제가 발생하였다.



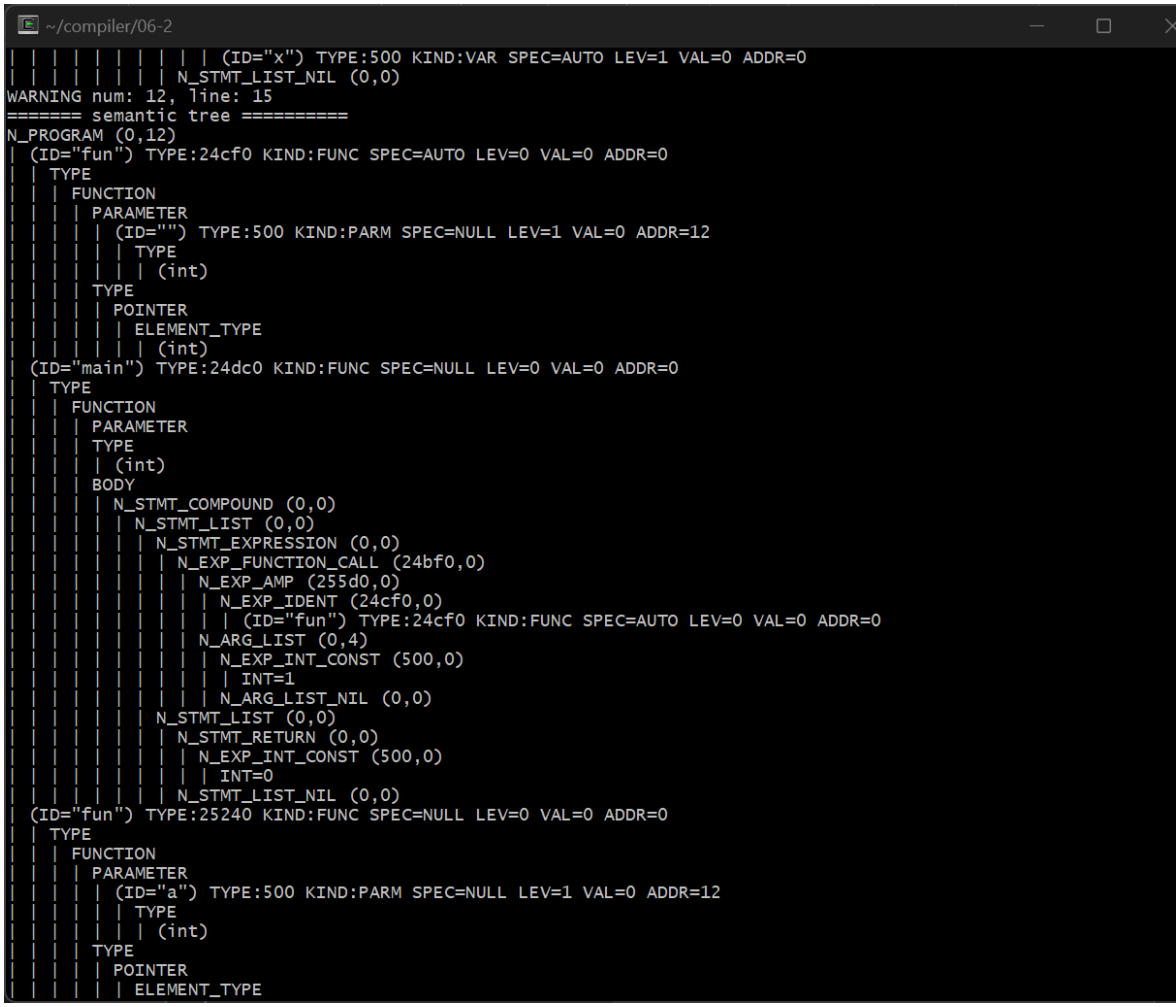
     - 실제 main 함수에서 어떤 부분에서 오류가 나는지 디버깅해본 결과, yyparse()에서 오류가 나는 것을 확인했다.
     - 따라서 기존에 정상적으로 작동했던 과제-5의 소스 코드를 백업하여 해당 코드로 덮어씌운 뒤, 빌드를 하였더니 segmentation fault가 발생하지 않았다.
     - 여기에 컴파일할 때 추가로 print_sem.c와 semantic.c를 같이 빌드하고 다시 명령어를 입력했을 때 segmentation fault가 발생하지 않고 정상 작동했다.

3. 테스트

- 올바른 C언어 코드
  - Hello World 출력

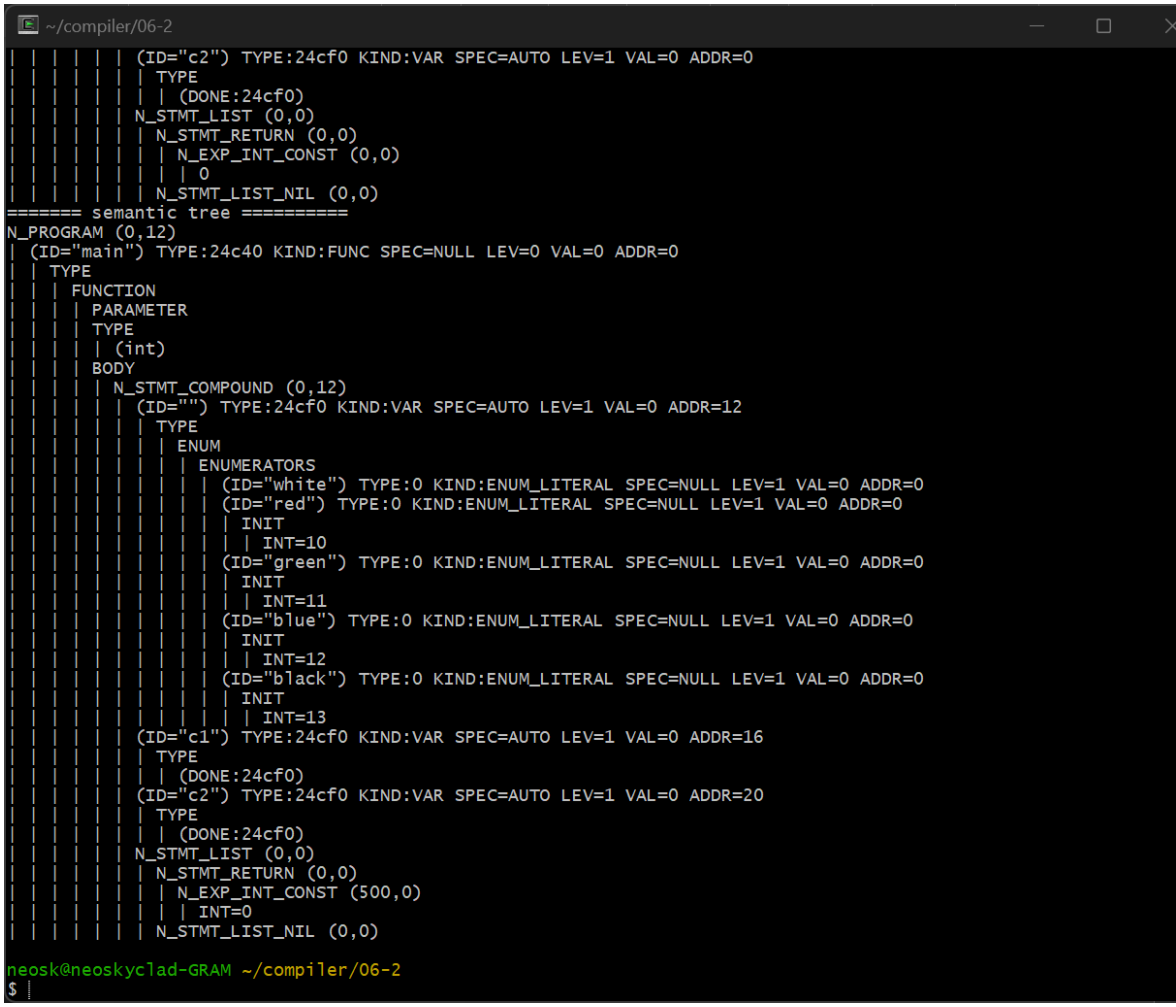


    - Main 함수에서 printf("Hello World!₩n")를 수행했다.
    - Syntax tree가 나오고 semantic tree가 이어서 출력된다.
  - int *fun() 함수



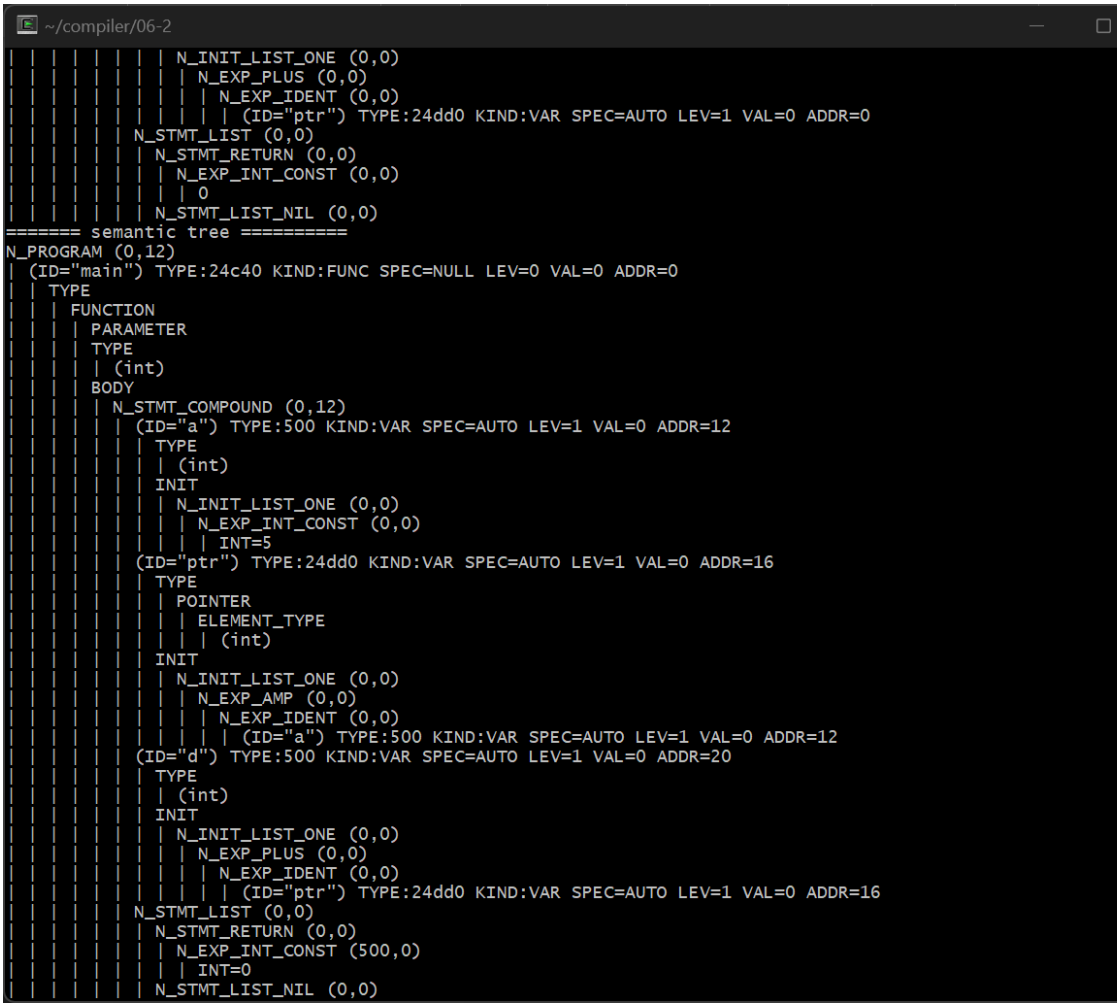
◆ 파라미터로 int a를 넘겨받고 지역 변수로 int x를 선언하는 int *fun()함수를 선언한다.

■ Enum 선언 테스트



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

- 잘못된 C언어 코드
  - ■ 13: arithmetic type expression required in unary operation
    - ◆ a.exe



- 오류가 검출되지 않고 시멘틱 검사가 진행됐다.

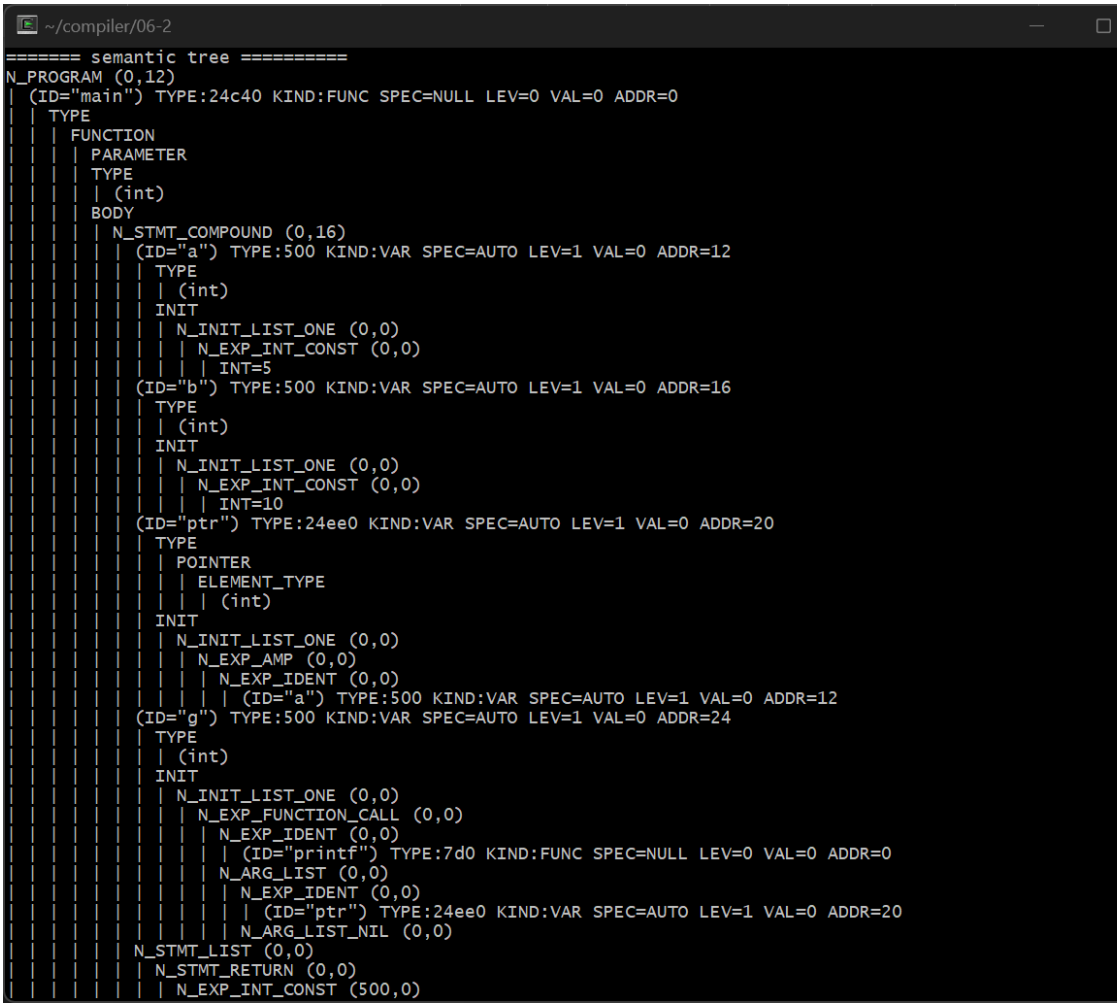
◆ gcc

```
neosk@neoskyclad-GRAM ~/compiler/06-2
$ gcc test.c
test.c: In function ‘main’:
test.c:9:13: error: wrong type argument to unary plus
    9 |     int d = +ptr;
      |             ^
```

- unary expression에 맞지 않는 type이라는 error 메시지를 출력했다.

■ 21: illegal type in function call expression

◆ a.exe



- function call이 검출됐지만, 에러 메시지를 보내지 않았다.

◆ gcc

```
test2.c:20:20: warning: passing argument 1 of ‘printf’ from incompatible pointer type [-Wincompatible-
types]
   20 |     int g = printf(ptr);
      |                    ^~~
      |                    |
      |                    int *
```

- printf()함수의 호환되지 않는 포인터 타입 에러 메시지를 출력했다.

■ 28: arithmetic type expression required in binary operation

◆ a.exe

```
~/compiler/06-2

neosk@neoskyclad-GRAM ~/compiler/06-2
$ !v
vi test.c

neosk@neoskyclad-GRAM ~/compiler/06-2
$ !.
./a.exe < test.c
syntax analysis start!
syntax analysis end (no error)
======= syntax tree ==========
N_PROGRAM (0,0)
| (ID="main") TYPE:24c40 KIND:FUNC SPEC=NULL LEV=0 VAL=0 ADDR=0
| | TYPE
| | | FUNCTION
| | | | PARAMETER
| | | | TYPE
| | | | | (int)
| | | | BODY
| | | | | N_STMT_COMPOUND (0,0)
| | | | | | (ID="a") TYPE:500 KIND:VAR SPEC=AUTO LEV=1 VAL=0 ADDR=0
| | | | | | | TYPE
| | | | | | | | (int)
| | | | | | | INIT
| | | | | | | | N_INIT_LIST_ONE (0,0)
| | | | | | | | | N_EXP_INT_CONST (0,0)
| | | | | | | | | | 5
| | | | | | (ID="j") TYPE:500 KIND:VAR SPEC=AUTO LEV=1 VAL=0 ADDR=0
| | | | | | | TYPE
| | | | | | | | (int)
| | | | | | N_STMT_LIST (0,0)
| | | | | | | N_STMT_EXPRESSION (0,0)
| | | | | | | | N_EXP_ASSIGN (0,0)
| | | | | | | | | N_EXP_IDENT (0,0)
| | | | | | | | | | (ID="j") TYPE:500 KIND:VAR SPEC=AUTO LEV=1 VAL=0 ADDR=0
| | | | | | | | | N_EXP_ADD (0,0)
| | | | | | | | | | N_EXP_IDENT (0,0)
| | | | | | | | | | | (ID="a") TYPE:500 KIND:VAR SPEC=AUTO LEV=1 VAL=0 ADDR=0
| | | | | | | | | | N_EXP_STRING_LITERAL (0,0)
| | | | | | | | | | | "hello"
| | | | | | | N_STMT_LIST (0,0)
| | | | | | | | N_STMT_RETURN (0,0)
| | | | | | | | | N_EXP_INT_CONST (0,0)
| | | | | | | | | | 0
| | | | | | | N_STMT_LIST_NIL (0,0)
ERROR num: 58, line: 28, identifier:

neosk@neoskyclad-GRAM ~/compiler/06-2
$
```

- syntax 분석은 통과했지만, semantic에서 error가 검출됐다.
- 28번 줄에서 에러 번호 58번의 에러가 발생했다.
  - Identifier에 해당하는 에러로 발생했다.

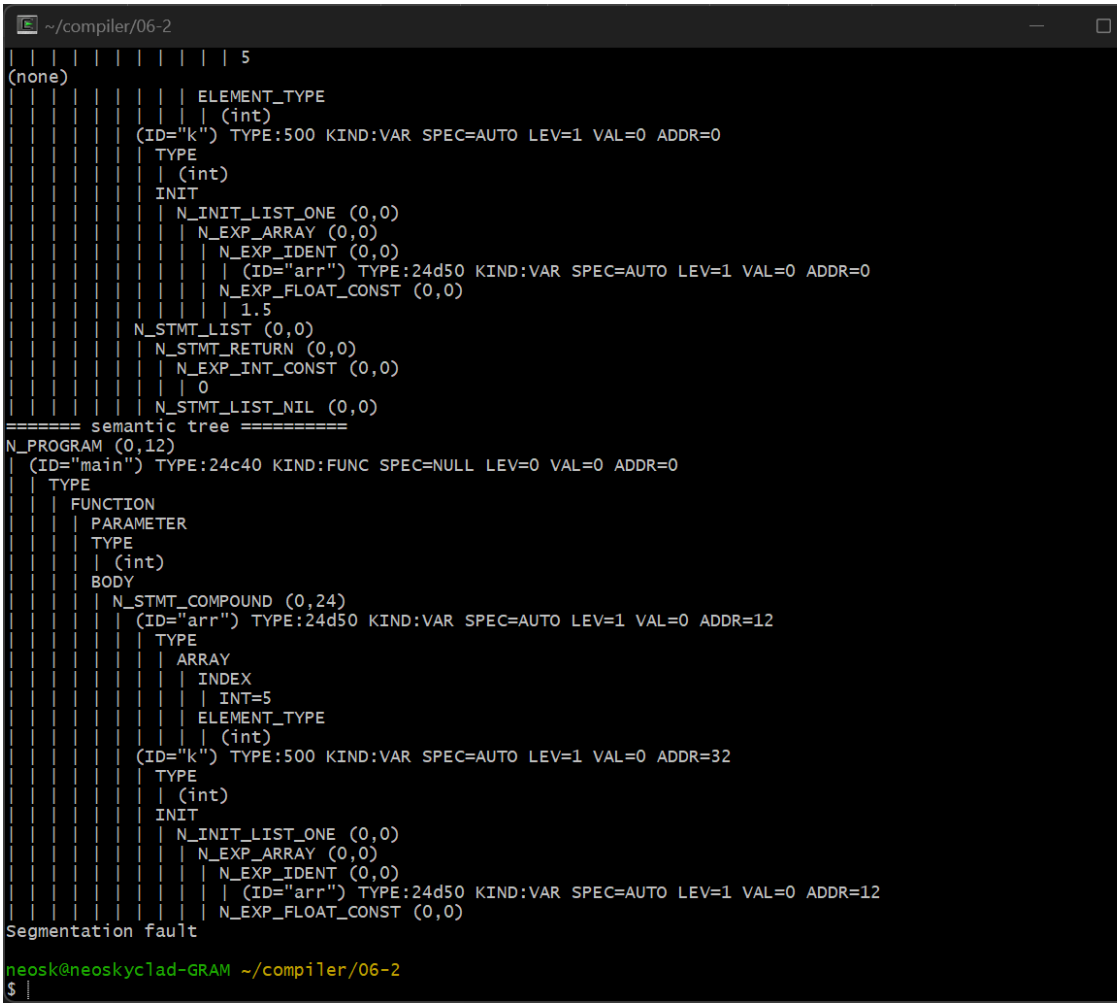
◆ gcc

```
test2.c:30:7: warning: assignment to ‘int’ from ‘char *’ makes integer from pointer without a cast [-Wi
ersion]
   30 |     j = a + "hello";
      |       ^
```

- 에러가 아닌 경고가 발생했다.

■ 29: integral type expression required in array subscript or binary operation

◆ a.exe



- arr변수까지 검출하다가 segmentation fault가 발생했다.

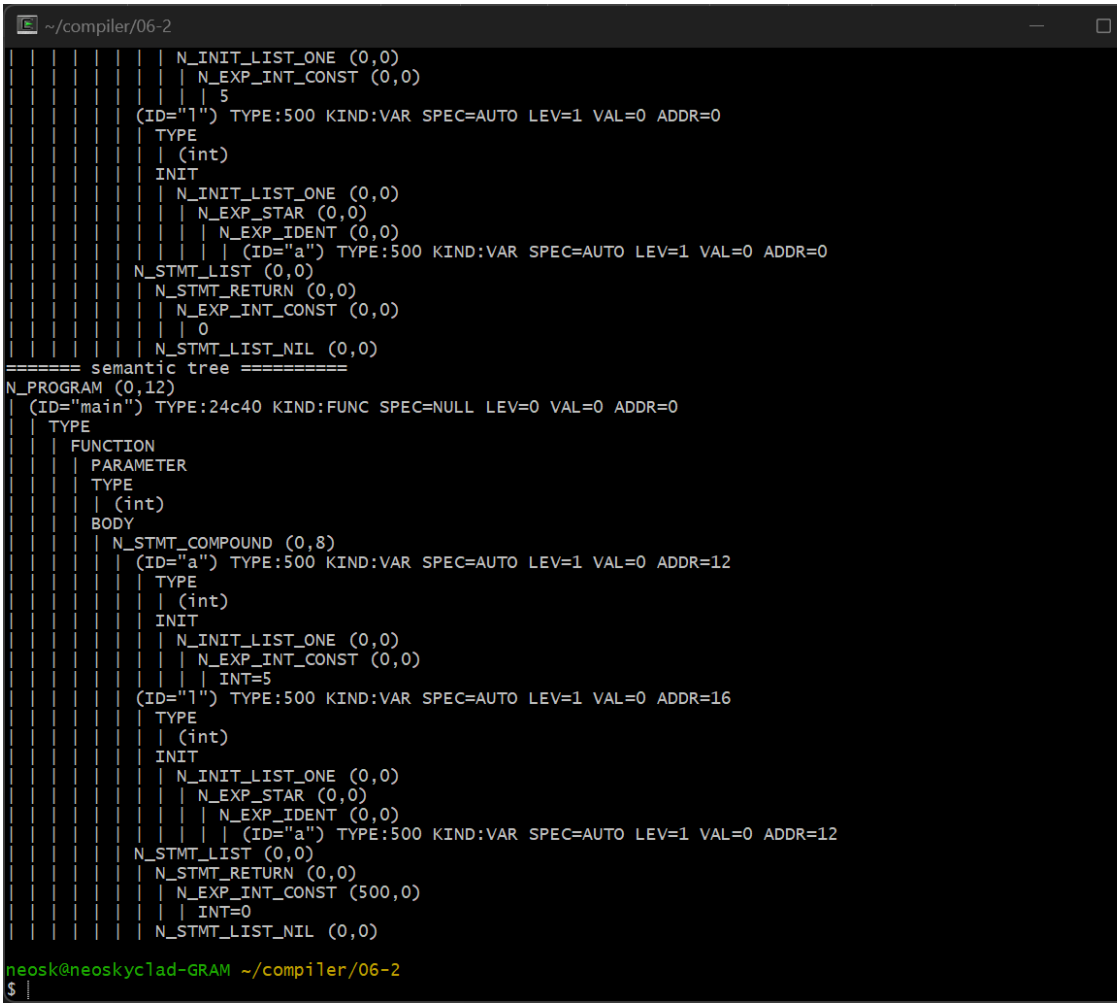
◆ gcc

```
test2.c:33:16: error: array subscript is not an integer
   33 |     int k = arr[1.5];
      |                ^
```

- 배열의 인덱스로 정수형만 올 수 있다는 에러 메시지를 출력했다.

■ 29: pointer type expression required in pointer operation

◆ a.exe



- gcc

```
test2.c:36:13: error: invalid type argument of unary '*' (have 'int')
   36 |     int l = *a;
      |             ^~
```

  - unary expression *에 대한 타입이 맞지 않다는 에러 메시지를 출력하고 있다.

- 90: fatal compiler error in parse result
  - a.exe

```
neosk@neoskyclad-GRAM ~/compiler/06-2
$ !.
./a.exe < test.c
syntax analysis start!
line 52: syntax error near return
```

    - syntax 분석에서 에러가 발생했다.
    - 52번째 줄에서 발생했음을 보이고 있다.
  - gcc

```
test2.c:52:5: error: expected ',' or ';' before 'int'
   52 |     int q = 20;
      |     ^~~
```

    - 똑같이 52번째 줄에서 에러가 발생함을 보이고 있다.

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 int semantic_err;
extern A_NODE *root;
```

```
void initialize();
void semantic_analysis();


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

- semantic.c

```
#include "type.h"


void semantic_analysis(A_NODE *);
void sem_program(A_NODE *);
int sem_declaration_list(A_ID *id, int addr);
int sem_declaration(A_ID *,int);
int sem_A_TYPE(A_TYPE *) ;
A_TYPE*sem_expression(A_NODE *);
int sem_statement(A_NODE *, int, A_TYPE *, BOOLEAN, BOOLEAN, BOOLEAN);
```

```
int sem_statement_list(A_NODE *, int, A_TYPE *, BOOLEAN, BOOLEAN, BOOLEAN);
void sem_for_expression(A_NODE *);
void sem_arg_expr_list(A_NODE *, A_ID *);
A_ID *getPointerFieldIdentifier(A_TYPE *, char *);
A_NODE *convertScalarToInteger(A_NODE *);
A_NODE *convertUsualAssignmentConversion(A_TYPE *, A_NODE *);
A_NODE *convertUsualUnaryConversion(A_NODE *);
A_TYPE *convertUsualBinaryConversion(A_NODE *);
A_NODE *convertCastingConversion(A_NODE *,A_TYPE *);
BOOLEAN isAllowableAssignmentConversion(A_TYPE *, A_TYPE *, A_NODE *);
BOOLEAN isAllowableCastingConversion(A_TYPE *, A_TYPE *);
BOOLEAN isModifiableLvalue(A_NODE *);
BOOLEAN isConstantZeroExp(A_NODE *);
BOOLEAN isSameParameterType(A_ID *, A_ID *);
BOOLEAN isNotSameType(A_TYPE *, A_TYPE *);
BOOLEAN isCompatibleType(A_TYPE *, A_TYPE *);
BOOLEAN isCompatiblePointerType(A_TYPE *, A_TYPE *);
BOOLEAN isIntType(A_TYPE *);
BOOLEAN isFloatType(A_TYPE *);
BOOLEAN isArithmeticType(A_TYPE *);
BOOLEAN isAnyIntegerType(A_TYPE *);
BOOLEAN isIntegralType(A_TYPE *);
BOOLEAN isFunctionType(A_TYPE *);
BOOLEAN isScalarType(A_TYPE *);
BOOLEAN isPointerType(A_TYPE *);
BOOLEAN isPointerOrArrayType_sem(A_TYPE *);
```

```
BOOLEAN isArrayType(A_TYPE *);
BOOLEAN isStringType(A_TYPE *);
BOOLEAN isVoidType(A_TYPE *);
A_LITERAL checkTypeAndConvertLiteral(A_LITERAL,A_TYPE*, int);
A_LITERAL getTypeAndValueOfExpression(A_NODE *);
A_TYPE *setTypeElementType(A_TYPE *, A_TYPE *);
A_TYPE *makeType(T_KIND);
void setTypeSize(A_TYPE *, int);
float atof();
void set_literal_address(A_NODE *);
int put_literal(A_LITERAL, int);
void semantic_warning(int, int);
void semantic_error();
A_NODE *makeNode(NODE_NAME, A_NODE *, A_NODE *, A_NODE*);
extern A_TYPE *int_type, *float_type, *char_type, *string_type, *void_type;

int global_address=12;
int semantic_err=0;
#define LIT_MAX 100
A_LITERAL literal_table[LIT_MAX];
int literal_no=0;
int literal_size=0;

void semantic_analysis(A_NODE *node) {
        sem_program(node);
        set_literal_address(node);
```

```
}

void set_literal_address(A_NODE *node) {
        int i;
        for (i=1;i<=literal_no; i++)
                literal_table[i].addr+=node->value;
        node->value+=literal_size;
}

void sem_program(A_NODE *node) {
        switch(node->name) {
                case N_PROGRAM :
                        sem_declaration_list(node->clink,12); // first parm addr = 12
                        node->value = global_address;
                        break;
        }
}

int put_literal(A_LITERAL lit, int ll) {
        float ff;
        if (literal_no >=LIT_MAX)
                semantic_error(93, ll);
        else
                literal_no++;
        literal_table[literal_no]=lit;
        literal_table[literal_no].addr=literal_size;
```

```
        if (lit.type->kind==T_ENUM)
                literal_size+=4;
        else if (isStringType(lit.type))
                literal_size+=strlen(lit.value.s)+1;
        if (literal_size%4)
                literal_size=literal_size/4*4+4;
        return(literal_no);
}

A_TYPE *sem_expression(A_NODE *node) {

        A_TYPE *result=NIL, *t,*t1, *t2;
        A_ID *id;
        A_LITERAL lit;
        int i;
        BOOLEAN lvalue=FALSE;
        switch(node->name) {
                case N_EXP_IDENT :
                        id=node->clink;
                        switch (id->kind) {
                                case ID_VAR:
                                case ID_PARM:
                                        result=id->type;
                                        if (!isArrayType(result))
                                                lvalue=TRUE;
                                        break;
```

```
                    case ID_FUNC:
                        result=id->type;
                        break;
                    case ID_ENUM_LITERAL:
                        result=int_type;
                        break;
                    default:
                        semantic_error(38, node->line, id->name);
                        break;
            }
            break;
    case N_EXP_INT_CONST :
            result=int_type;
            break;
    case N_EXP_FLOAT_CONST :
            lit.type=float_type;
            lit.value.f = atof(node->clink);
            node->clink=put_literal(lit,node->line);
            result = float_type;
            break;
    case N_EXP_CHAR_CONST :
            result=char_type;
            break;
    case N_EXP_STRING_LITERAL :
            lit.type=string_type;
            lit.value.s=node->clink;
```

```
            node->clink=put_literal(lit,node->line);
            result=string_type;
            break;
    case N_EXP_ARRAY :
            t1=sem_expression(node->llink);
            t2=sem_expression(node->rlink);
            t=convertUsualBinaryConversion(node);
            t1=node->llink->type;
            t2=node->rlink->type;
            if (isPointerOrArrayType_sem(t1))
                    result=t1->element_type;
            else
                    semantic_error(32,node->line);
            if (!isIntegralType(t2))
                    semantic_error(29,node->line);
            if (!isArrayType(result))
                    lvalue=TRUE;
            break;
    case N_EXP_ARROW:
            t=sem_expression(node->llink);
            id=getPointerFieldIdentifier(t,node->rlink);
            if (id) {
                    result=id->type;
                    if (!isArrayType(result))
                            lvalue=TRUE;
            }
```

```
                else
                        semantic_error(37,node->line);
                node->rlink=id;
                break;
        case N_EXP_FUNCTION_CALL :
                t=sem_expression(node->llink);
                node->llink=convertUsualUnaryConversion(node->llink);
                t=node->llink->type;
                if (isPointerType(t) && isFunctionType(t->element_type)) {
                        sem_arg_expr_list(node->rlink,t->element_type->field);
                        result=t->element_type->element_type;
                }
                else
                        semantic_error(21,node->line);
                break;
        case N_EXP_POST_INC :
        case N_EXP_POST_DEC :
                result=sem_expression(node->clink);
                // usual binary conversion between the expression and 1 if
(!isScalarType(result))
                if(!isScalarType(result))
                        semantic_error(27,node->line);
                // check if modifiable lvalue
                if (!isModifiableLvalue(node->clink))
                        semantic_error(60,node->line);//
                break;
```

```
		case N_EXP_CAST :
			result=node->llink;
			i=sem_A_TYPE(result);
			t=sem_expression(node->rlink);
			// check allowable casting conversion
			if (!isAllowableCastingConversion(result,t))
				semantic_error(58,node->line);
			break;
		case N_EXP_SIZE_TYPE :
			t=node->clink;
			i=sem_A_TYPE(t);
			// check if incomplete array, function, void
			if (isArrayType(t) && t->size==0 || isFunctionType(t) || isVoidType(t))
				semantic_error(39,node->line);
			else
				node->clink=i;
			result=int_type;
			break;
		case N_EXP_SIZE_EXP :
			t=sem_expression(node->clink);
			// check if incomplete array, function (for non parameter
			if ((node->clink->name!=N_EXP_IDENT || ₩
					((A_ID*)node->clink->clink)->kind!=ID_PARM)
&& ₩
				(isArrayType(t) && t->size==0 || isFunctionType(t)))
				semantic_error(39,node->line);
```

```
            else
                    node->clink=t->size;
            result=int_type;
            break;
    case N_EXP_PLUS :
    case N_EXP_MINUS :
            t=sem_expression(node->clink);
            if (isArithmeticType(t)) {
                    node->clink=convertUsualUnaryConversion(node->clink);
                    result=node->clink->type;
            }
            else
                    semantic_error(13,node->line);
            break;
    case N_EXP_NOT :
            t=sem_expression(node->clink);
            if (isScalarType(t)) {
                    node->clink=convertUsualUnaryConversion(node->clink);
                    result=node->clink->type;
            }
            else
                    semantic_error(27,node->line);
            break;
    case N_EXP_AMP :
            t=sem_expression(node->clink);
            if (node->clink->value==TRUE || isFunctionType(t)) {
```

```
                result=setTypeElementType(makeType(T_POINTER),t);
                result->size=4;
        }
        else
                semantic_error(60,node->line);
        break;
case N_EXP_STAR :
        t=sem_expression(node->clink);
        node->clink=convertUsualUnaryConversion(node->clink);
        if (isPointerType(t)) {
                result=t->element_type;
                // lvalue if points to an object
                if (isScalarType(result))
                        lvalue=TRUE;
        }
        else
                semantic_error(31,node->line);
        break;
case N_EXP_PRE_INC :
case N_EXP_PRE_DEC :
        result=sem_expression(node->clink);
        // usual binary conversion between the expression and 1
        if (!isScalarType(result))
                semantic_error(27,node->line);
        // check if modifiable lvalue
        if (!isModifiableLvalue(node->clink))
```

```
                semantic_error(60,node->line);
            break;
        case N_EXP_MUL :
        case N_EXP_DIV :
            t1=sem_expression(node->llink);
            t2=sem_expression(node->rlink);
            if (isArithmeticType(t1) && isArithmeticType(t2))
                result=convertUsualBinaryConversion(node);
            else
                semantic_error(28,node->line);
            break;
        case N_EXP_MOD :
            t1=sem_expression(node->llink);
            t2=sem_expression(node->rlink);
            if (isIntegralType(t1) && isIntegralType(t2))
                result=convertUsualBinaryConversion(node);
            else
                semantic_error(29,node->line);
            result=int_type;
            break;
        case N_EXP_ADD :
            t1=sem_expression(node->llink);
            t2=sem_expression(node->rlink);
            if (isArithmeticType(t1) && isArithmeticType(t2))
                result=convertUsualBinaryConversion(node);
            else if (isPointerType(t1) && isIntegralType(t2))
```

```
                result=t1;
        else if (isIntegralType(t1) && isPointerType(t2))
                result=t2;
        else
                semantic_error(24,node->line);
        break;
case N_EXP_SUB :
        t1=sem_expression(node->llink);
        t2=sem_expression(node->rlink);
        if (isArithmeticType(t1) && isArithmeticType(t2))
                result=convertUsualBinaryConversion(node);
        else if (isPointerType(t1) && isIntegralType(t2))
                result=t1;
        else if (isCompatiblePointerType(t1, t2))
                result=t1;
        else
                semantic_error(24,node->line);
        break;
case N_EXP_LSS :
case N_EXP_GTR :
case N_EXP_LEQ :
case N_EXP_GEQ :
        t1=sem_expression(node->llink);
        t2=sem_expression(node->rlink);
        if (isArithmeticType(t1) && isArithmeticType(t2))
                result=convertUsualBinaryConversion(node);
```

```
                else if (!isCompatiblePointerType(t1,t2))
                        semantic_error(40, node->line);
                result = int_type;
                break;
        case N_EXP_NEQ :
        case N_EXP_EQL :
                t1=sem_expression(node->llink);
                t2=sem_expression(node->rlink);
                if (isArithmeticType(t1) && isArithmeticType(t2))
                        result=convertUsualBinaryConversion(node);
                else if (!isCompatiblePointerType(t1,t2) &&
                                (!isPointerType(t1) || isConstantZeroExp(node->rlink))
&&
                                (!isPointerType(t2) || isConstantZeroExp(node->rlink)))
                        semantic_error(40, node->line);
                result = int_type;
                break;
        case N_EXP_AND :
        case N_EXP_OR :
                t=sem_expression(node->llink);
                if(!isScalarType(t))
                        node->llink = convertUsualUnaryConversion(node->llink);
                else
                        semantic_error(27, node->line);
                t = sem_expression(node->rlink);
                if(!isScalarType(t))
```

```
                        node->rlink = convertUsualUnaryConversion(node->rlink);
                    else
                        semantic_error(27, node->line);
                    result = int_type;
                    break;
                case N_EXP_ASSIGN :
                    result = sem_expression(node->llink);
                    // check modifiable lvalue
                    if(!isModifiableLvalue(node->llink))
                        semantic_error(60, node->line);
                    t = sem_expression(node->rlink);
                    // check modifiable lvalue
                    if(isAllowableAssignmentConversion(result, t, node->rlink)) {
                        if(isArithmeticType(result) && isArithmeticType(t))
                            node->rlink =
convertUsualAssignmentConversion(result, node->rlink);
                    }
                    else
                        semantic_error(58, node->line);
                    break;
                default :
                    semantic_error(90,node->line);
                    break;
            }
            node->type = result;
            node->value = lvalue;
```

```
	return (result);
}

// check argument-expression-list in function call expression
void sem_arg_expr_list(A_NODE *node, A_ID *id)
{
	A_TYPE *t;
	A_ID *a;
	int arg_size=0;
	switch(node->name) {
		case N_ARG_LIST :
			if (id==0)
				semantic_error(34,node->line);
			else {
				if (id->type) {
					t=sem_expression(node->llink);
					node->llink=convertUsualUnaryConversion(node->llink);
					if(isAllowableCastingConversion(id->type,node->llink->type))
						node->llink=convertCastingConversion(node->llink,id->type);
					else
						semantic_error(59,node->line);
					sem_arg_expr_list(node->rlink,id->link);
				}
```

```
                                else {     // DOTDOT parameter : no conversion
                                        t=sem_expression(node->llink);
                                        sem_arg_expr_list(node->rlink,id);
                                }
                                arg_size=node->llink->type->size+node->rlink->value;
                        }
                        break;
                case N_ARG_LIST_NIL :
                        if (id && id->type) // check if '...' argument
                                semantic_error(35,node->line);
                        break;
                default :
                        semantic_error(90,node->line);
                        break;
        }
        if (arg_size%4)
                arg_size=arg_size/4*4+4;
        node->value=arg_size;
}

BOOLEAN isModifiableLvalue(A_NODE *node)
{
        if (node->value==FALSE || isFunctionType(node->type))
                return FALSE;
        else
                return TRUE;
```

```
}

// check statement and return local variable size
int sem_statement(A_NODE *node, int addr, A_TYPE *ret, BOOLEAN sw, BOOLEAN brk, BOOLEAN cnt)
{
	int local_size=0,i;
	A_LITERAL lit;
	A_TYPE *t;
	switch(node->name) {
		case N_STMT_LABEL_CASE :
			if (sw==FALSE) // case statement is not in 'switch'
				semantic_error(71,node->line);
			lit=getTypeAndValueOfExpression(node->llink);
			if (isIntegralType(lit.type))
				node->llink=lit.value.i;
			else
				semantic_error(51,node->line);
			local_size=sem_statement(node->rlink,addr,ret,sw,brk,cnt);
			break;
		case N_STMT_LABEL_DEFAULT :
			if (sw==FALSE)
				semantic_error(72,node->line);
			local_size=sem_statement(node->clink,addr,ret,sw,brk,cnt);
			break;
		case N_STMT_COMPOUND:
```

```
                if(node->llink)
                        local_size=sem_declaration_list(node->llink,addr);
                local_size+=sem_statement_list(node->rlink,local_size+addr,ret,sw,brk,cnt);
                break;
        case N_STMT_EMPTY:
                break;
        case N_STMT_EXPRESSION:
                t=sem_expression(node->clink);
                break;
        case N_STMT_IF:
                t=sem_expression(node->llink);
                if (isScalarType(t))
                        node->llink=convertScalarToInteger(node->llink);
                else
                        semantic_error(50,node->line);
                local_size=sem_statement(node->rlink,addr,ret,FALSE,brk,cnt);
                break;
        case N_STMT_IF_ELSE:
                t=sem_expression(node->llink);
                if (isScalarType(t))
                        node->llink=convertScalarToInteger(node->llink);
                else
                        semantic_error(50,node->line);
                local_size=sem_statement(node->clink,addr,ret,FALSE,brk,cnt);
                i=sem_statement(node->rlink,addr,ret,FALSE,brk,cnt);
```

```
            if (local_size<i)
                    local_size=i;
            break;
    case N_STMT_WHILE:
            t=sem_expression(node->llink);
            if (isScalarType(t))
                    node->llink=convertScalarToInteger(node->llink);
            else
                    semantic_error(50,node->line);
            local_size=sem_statement(node->rlink,addr,ret,FALSE,TRUE,TRUE);
            break;
    case N_STMT_DO:
            local_size=sem_statement(node->llink,addr,ret,FALSE,TRUE,TRUE);
            t=sem_expression(node->rlink);
            if (isScalarType(t))
                    node->rlink=convertScalarToInteger(node->rlink);
            else
                    semantic_error(50,node->line);
            break;
    case N_STMT_FOR:
            sem_for_expression(node->llink);
            local_size=sem_statement(node->rlink,addr,ret,FALSE,TRUE,TRUE);
            break;
    case N_STMT_CONTINUE:
            if (cnt==FALSE)
                    semantic_error(74,node->line);
```

```
                        break;
                case N_STMT_BREAK:
                        if (brk==FALSE)
                                semantic_error(73,node->line);
                        break;
                case N_STMT_RETURN:
                        if(node->clink){
                                t=sem_expression(node->clink);
                                if (isAllowableCastingConversion(ret,t))
                                        node->clink=convertCastingConversion(node-
>clink,ret);
                                else
                                        semantic_error(57,node->line);
                        }
                        break;
                default:
                        semantic_error(90,node->line);
                        break;
        }
        node->value=local_size;
        return(local_size);
}

void sem_for_expression(A_NODE *node) {
        A_TYPE *t;
        switch (node->name) {
```

```
                case N_FOR_EXP :
                        if(node->llink)
                                t=sem_expression(node->llink);
                        if(node->clink) {
                                t=sem_expression(node->clink);
                                if (isScalarType(t))
                                        node->clink=convertScalarToInteger(node->clink);
                                else
                                        semantic_error(49,node->line);
                        }
                        if(node->rlink)
                                t=sem_expression(node->rlink);
                        break;
                default :
                        semantic_error(90,node->line);
                        break;
        }
}

// check statement-list and return local variable size
int sem_statement_list(A_NODE *node, int addr, A_TYPE *ret, BOOLEAN sw, BOOLEAN brk,
BOOLEAN cnt)
{
        int size,i;
        switch(node->name) {
                case N_STMT_LIST:
```

```
                        size=sem_statement(node->llink, addr,ret,sw,brk,cnt);
                        i=sem_statement_list(node->rlink, addr,ret,sw,brk,cnt);
                        if(size<i)
                                size=i;
                        break;
                case N_STMT_LIST_NIL:
                        size=0;
                        break;
                default :
                        semantic_error(90,node->line);
                        break;
        }
        node->value=size;
        return(size);
}

// check type and return its size (size of incomplete type is 0)
int sem_A_TYPE(A_TYPE *t)
{
        A_ID *id;
        A_TYPE *tt;
        A_LITERAL lit;
        int result=0,i;

        if (t->check)
                return(t->size);
```

```
t->check=1;


switch (t->kind) {
        case T_NULL:
                semantic_error(80,t->line);
                break;
        case T_ENUM:
                i=0;
                id=t->field;
                while (id) { // enumerators
                        if (id->init){
                                lit=getTypeAndValueOfExpression(id->init);
                                if (!isIntType(lit.type))
                                        semantic_error(81,id->line);
                                i=lit.value.i;
                        }
                        id->init=i++;
                        id=id->link;
                }
                result=4;
                break;
        case T_ARRAY:
                if (t->expr){
                        lit=getTypeAndValueOfExpression(t->expr);
                        if (!isIntType(lit.type) || lit.value.i<=0) {
                                semantic_error(82,t->line);
```

```
                        t->expr=0;
                    } else
                        t->expr=lit.value.i;
                }
                i=sem_A_TYPE(t->element_type)*(int)t->expr;
                if (isVoidType(t->element_type) || isFunctionType(t->element_type))
                    semantic_error(83,t->line);
                else
                    result=i;
                break;
            case T_FUNC:
                tt=t->element_type;
                i=sem_A_TYPE(tt);
                if (isArrayType(tt) || isFunctionType(tt)) // check return type
                    semantic_error(85,t->line);
                i=sem_declaration_list(t->field,12)+12; // parameter type & size
                if (t->expr) {
                    i=i+sem_statement(t->expr,i,t->element_type,FALSE,FALSE,FALSE);
                    t->local_var_size=i;
                    break;
                }
                t->local_var_size=i;
                break;
            case T_POINTER:
                i=sem_A_TYPE(t->element_type);
```

```
                result=4;
                break;
        case T_VOID:
                break;
        default:
                semantic_error(90,t->line);
                break;
    }
    t->size=result;
    return(result);
}

// set variable address in declaration-list, and return its total variable size
int sem_declaration_list(A_ID *id, int addr)
{
    int i=addr;
    while (id) {
        addr+=sem_declaration(id,addr);
        id=id->link;
    }
    return(addr-i);
}

// check declaration (identifier), set address, and return its size
int sem_declaration(A_ID *id,int addr)
{
```

```
A_TYPE *t;
int size=0,i;
A_LITERAL lit;

switch (id->kind) {
        case ID_VAR:
                i=sem_A_TYPE(id->type);

                // check empty array
                if (isArrayType(id->type) && id->type->expr==NIL)
                        semantic_error(86,id->line);
                if (i%4)
                        i=i/4*4+4;
                if (id->specifier==S_STATIC)
                        id->level=0;
                if (id->level==0) // if global scope
                {
                        id->address=global_address;
                        global_address+=i;
                }
                else {
                        id->address=addr;
                        size=i;
                }
                break;
        case ID_FIELD:
```

```
            i=sem_A_TYPE(id->type);
            if (isFunctionType(id->type) || isVoidType(id->type))
                        semantic_error(84,id->line);
            if (i%4)
                        i=i/4*4+4;
            id->address=addr;
            size=i;
            break;
case ID_FUNC:
            i=sem_A_TYPE(id->type);
            break;
case ID_PARM:
            if (id->type)
            {
                        size=sem_A_TYPE(id->type);
                        // usual unary conversion of parm type
                        if (id->type==char_type)
                                    id->type=int_type;
                        else if (isArrayType(id->type)){
                                    id->type->kind=T_POINTER;
                                    id->type->size=4;
                        }
                        else if (isFunctionType(id->type)) {
                                    t=makeType(T_POINTER);
                                    t->element_type=id->type;
                                    t->size=4;
```

```
                                        id->type=t;
                                }
                                size=id->type->size;
                                if (size%4)
                                        size=size/4*4+4;
                                id->address=addr;
                        }
                        break;
                case ID_TYPE:
                        i=sem_A_TYPE(id->type);
                        break;
                default:
                        semantic_error(89,id->line,id->name);
                        break;
        }
        return (size);
}
A_ID *getPointerFieldIdentifier(A_TYPE *t, char *s) {
        A_ID *id=NIL;
        if (t && t->kind==T_POINTER) {
                t=t->element_type;
        }
}
BOOLEAN isSameParameterType(A_ID *a, A_ID *b) {
        while (a) {
                if (b==NIL || isNotSameType(a->type,b->type))
```

```
                return (FALSE);
            a=a->link;
            b=b->link;
        }
        if (b)
            return (FALSE);
        else
            return (TRUE);
}

BOOLEAN isCompatibleType(A_TYPE *t1, A_TYPE *t2) {
        if (isArrayType(t1) && isArrayType(t2))
            if (t1->size==0 || t2->size==0 || t1->size==t2->size)
                return(isCompatibleType(t1->element_type,t2->element_type));
            else
                return(FALSE);
        else if (isFunctionType(t1) && isFunctionType(t2))
            if (isSameParameterType(t1->field,t2->field))
                return(isCompatibleType(t1->element_type,t2->element_type));
            else
                return (FALSE);
        else if (isPointerType(t1) && isPointerType(t2))
            return(isCompatibleType(t1->element_type,t2->element_type));
        else
            return(t1==t2);
}
```

```
BOOLEAN isConstantZeroExp(A_NODE *node) {
	if (node->name==N_EXP_INT_CONST && node->clink==0)
		return (TRUE);
	else
		return (FALSE);
}
BOOLEAN isCompatiblePointerType(A_TYPE *t1, A_TYPE *t2) {
	if (isPointerType(t1) && isPointerType(t2))
		return(isCompatibleType(t1->element_type,t2->element_type));
	else
		return(FALSE);
}
A_NODE *convertScalarToInteger(A_NODE *node) {
	if (isFloatType(node->type)) {
		semantic_warning(16,node->line);
		node=makeNode(N_EXP_CAST,int_type,NIL,node);
	}
	node->type=int_type;
	return(node);
}
A_NODE *convertUsualAssignmentConversion(A_TYPE *t1, A_NODE *node)
{
	A_TYPE *t2;
	t2=node->type;
	if (!isCompatibleType(t1,t2)) {
		semantic_warning(11,node->line);
```

```
                node=makeNode(N_EXP_CAST,t1,NIL,node);
                node->type=t1;
        }
        return (node);
}
A_NODE *convertUsualUnaryConversion(A_NODE *node) {
        A_TYPE *t;
        t=node->type;
        if (t==char_type) {
                t=int_type;
                node=makeNode(N_EXP_CAST,t,NIL,node);
                node->type=t;
        }
        else if (isArrayType(t)){
                t=setTypeElementType(makeType(T_POINTER),t->element_type);
                t->size=4;
                node=makeNode(N_EXP_CAST,t,NIL,node);
                node->type=t;
        }
        else if (isFunctionType(t)){
                t=setTypeElementType(makeType(T_POINTER),t);
                t->size=4;
                node=makeNode(N_EXP_AMP,NIL,node,NIL);
                node->type=t;
        }
        return (node);
```

```
}
A_TYPE *convertUsualBinaryConversion(A_NODE *node) {
        A_TYPE *t1, *t2, *result=NIL;
        t1=node->llink->type;
        t2=node->rlink->type;
        if(isFloatType(t1) && !isFloatType(t2)) {
                semantic_warning(14,node->line);
                node->rlink=makeNode(N_EXP_CAST,t1,NIL,node->rlink);
                node->rlink->type=t1;
                result=t1;
        }
        else if(!isFloatType(t1) && isFloatType(t2)) {
                semantic_warning(14,node->line);
                node->llink=makeNode(N_EXP_CAST,t2,NIL,node->llink);
                node->llink->type=t2;
                result=t2;
        }
        else if (t1==t2)
                result=t1;
        else
                result=int_type;
        return (result);
}
A_NODE *convertCastingConversion(A_NODE *node,A_TYPE *t1) {
        A_TYPE *t2;
        t2=node->type;
```

```
	if (!isCompatibleType(t1,t2)) {
		semantic_warning(12,node->line);
		node=makeNode(N_EXP_CAST,t1,NIL,node);
		node->type=t1;
	}
	return (node);
}
BOOLEAN isAllowableAssignmentConversion(A_TYPE *t1, A_TYPE *t2, A_NODE *node) // t1 <--- t2
{
	if (isArithmeticType(t1) && isArithmeticType(t2))
		return (TRUE);
	else if (isCompatibleType(t1,t2))
		return (TRUE);
	else if (isPointerType(t1) && (isConstantZeroExp(node) || isCompatiblePointerType(t1,t2)))
		return (TRUE);
	else
		return (FALSE);
}
BOOLEAN isAllowableCastingConversion(A_TYPE *t1, A_TYPE *t2)
{
	// t1 <---
	if (isAnyIntegerType(t1) && (isAnyIntegerType(t2) ||
					isFloatType(t2) ||
					isPointerType(t2)))
		return (TRUE);
	else if (isFloatType(t1) && isArithmeticType(t2))
```

```
                return (TRUE);
        else if (isPointerType(t1) && (isAnyIntegerType(t2) || isPointerType(t2)))
                return (TRUE);
        else if (isVoidType(t1))
                return (TRUE);
        else
                return (FALSE);
}
BOOLEAN isFloatType(A_TYPE *t) {
        if (t ==float_type)
                return(TRUE);
        else
                return(FALSE);
}
BOOLEAN isArithmeticType(A_TYPE *t) {
        if (t && t->kind==T_ENUM)
                return(TRUE);
        else
                return(FALSE);
}
BOOLEAN isScalarType(A_TYPE *t) {
        if (t && ((t->kind==T_ENUM) || (t->kind==T_POINTER)))
                return(TRUE);
        else
                return(FALSE);
}
```

```
BOOLEAN isAnyIntegerType(A_TYPE *t) {
        if ( t && (t==int_type || t==char_type))
                return(TRUE);
        else
                return(FALSE);
}
BOOLEAN isIntegralType(A_TYPE *t) {
        if ( t && t->kind==T_ENUM && t!=float_type)
                return(TRUE);
        else
                return(FALSE);
}
BOOLEAN isFunctionType(A_TYPE *t) {
        if (t && t->kind==T_FUNC)
                return(TRUE);
        else
                return(FALSE);
}
BOOLEAN isPointerType(A_TYPE *t) {
        if (t && t->kind==T_POINTER)
                return(TRUE);
        else
                return(FALSE);
}
BOOLEAN isPointerOrArrayType_sem(A_TYPE *t) {
        if (t && ( t->kind==T_POINTER || t->kind == T_ARRAY))
```

```
                return(TRUE);
        else
                return(FALSE);
}
BOOLEAN isIntType(A_TYPE *t) {
        if (t && t==int_type)
                return(TRUE);
        else
                return(FALSE);
}
BOOLEAN isVoidType(A_TYPE *t) {
        if (t && t==void_type)
                return(TRUE);
        else
                return(FALSE);
}
BOOLEAN isArrayType(A_TYPE *t) {
        if (t && t->kind==T_ARRAY)
                return(TRUE);
        else
                return(FALSE);
}
BOOLEAN isStringType(A_TYPE *t) {
        if (t && (t->kind==T_POINTER||t->kind==T_ARRAY) && t->element_type==char_type)
                return(TRUE);
        else
```

```
                return(FALSE);
}
// convert literal type
A_LITERAL checkTypeAndConvertLiteral(A_LITERAL result,A_TYPE *t, int ll) {
        if (result.type==int_type && t==int_type ||
                        result.type==char_type && t==char_type ||
                        result.type==float_type && t==float_type ) ;
        else if (result.type==int_type && t==float_type){
                result.type=float_type;
                result.value.f=result.value.i;
        }
        else if (result.type==int_type && t==char_type){
                result.type=char_type;
                result.value.c=result.value.i;
        }
        else if (result.type==float_type && t==int_type){
                result.type=int_type;
                result.value.i=result.value.f;
        }
        else if (result.type==char_type && t==int_type){
                result.type=int_type;
                result.value.i=result.value.c;
        }
        else
                semantic_error(41,ll); return (result);
}
```

```
A_LITERAL getTypeAndValueOfExpression(A_NODE *node) {
        A_TYPE *t;
        A_ID *id;
        A_LITERAL result,r;
        result.type=NIL;
        switch(node->name) {
                case N_EXP_IDENT :
                        id=node->clink;
                        if (id->kind!=ID_ENUM_LITERAL)
                                semantic_error(19,node->line,id->name);
                        else {
                                result.type=int_type;
                                result.value.i=id->init;
                        }
                        break;
                case N_EXP_INT_CONST :
                        result.type=int_type;
                        result.value.i=(int)node->clink;
                        break;
                case N_EXP_CHAR_CONST :
                        result.type=char_type;
                        result.value.c=(char)node->clink;
                        break;
                case N_EXP_FLOAT_CONST :
                        result.type=float_type;
                        result.value.f=atof(node->clink);
```

```
            break;
case N_EXP_STRING_LITERAL :
case N_EXP_ARRAY :
case N_EXP_FUNCTION_CALL :
case N_EXP_ARROW :
case N_EXP_POST_INC :
case N_EXP_PRE_INC :
case N_EXP_POST_DEC :
case N_EXP_PRE_DEC :
case N_EXP_AMP :
case N_EXP_STAR :
case N_EXP_NOT :
            semantic_error(18,node->line);
            break;
case N_EXP_MINUS :
            result=getTypeAndValueOfExpression(node->clink);
            if (result.type==int_type)
                        result.value.i=-result.value.i;
            else if (result.type==float_type)
                        result.value.f=-result.value.f;
            else
                        semantic_error(18,node->line);
            break;
case N_EXP_SIZE_EXP :
            t=sem_expression(node->clink);
            result.type=int_type;
```

```
                result.value.i=t->size;
                break;
        case N_EXP_SIZE_TYPE :
                result.type=int_type;
                result.value.i=sem_A_TYPE(node->clink);
                break;
        case N_EXP_CAST :
                result=getTypeAndValueOfExpression(node->rlink);
                result=checkTypeAndConvertLiteral(result,(A_TYPE*)node->llink,node->line);
                break;
        case N_EXP_MUL :
                result=getTypeAndValueOfExpression(node->llink);
                r=getTypeAndValueOfExpression(node->rlink);
                if (result.type==int_type && r.type==int_type){
                        result.type=int_type;
                        result.value.i=result.value.i*r.value.i;
                }
                else if (result.type==int_type && r.type==float_type){
                        result.type=float_type;
                        result.value.f=result.value.i*r.value.f;
                }
                else if (result.type==float_type && r.type==int_type){
                        result.type=float_type;
                        result.value.f=result.value.f*r.value.i;
                }
```

```
            else if (result.type==float_type && r.type==float_type){
                    result.type=float_type;
                    result.value.f=result.value.f*r.value.f;
            }
            else
                    semantic_error(18,node->line);
            break;
    case N_EXP_DIV :
            result=getTypeAndValueOfExpression(node->llink);
            r=getTypeAndValueOfExpression(node->rlink);
            if (result.type==int_type && r.type==int_type){
                    result.type=int_type;
                    result.value.i=result.value.i/r.value.i;
            }
            else if (result.type==int_type && r.type==float_type){
                    result.type=float_type;
                    result.value.f=result.value.i/r.value.f;
            }
            else if (result.type==float_type && r.type==int_type){
                    result.type=float_type;
                    result.value.f=result.value.f/r.value.i;
            }
            else if (result.type==float_type && r.type==float_type){
                    result.type=float_type;
                    result.value.f=result.value.f/r.value.f;
            }
```

```
        else
                semantic_error(18,node->line);
        break;
case N_EXP_MOD :
        result=getTypeAndValueOfExpression(node->llink);
        r=getTypeAndValueOfExpression(node->rlink);
        if (result.type==int_type && r.type==int_type)
                result.value.i=result.value.i%r.value.i;
        else
                semantic_error(18,node->line); break;
case N_EXP_ADD :
        result=getTypeAndValueOfExpression(node->llink);
        r=getTypeAndValueOfExpression(node->rlink);
        if (result.type==int_type && r.type==int_type){
                result.type=int_type;
                result.value.i=result.value.i+r.value.i;}
        else if (result.type==int_type && r.type==float_type){
                result.type=float_type;
                result.value.f=result.value.i+r.value.f;
        }
        else if (result.type==float_type && r.type==int_type){
                result.type=float_type;
                result.value.f=result.value.f+r.value.i;
        }
        else if (result.type==float_type && r.type==float_type){
                result.type=float_type; result.value.f=result.value.f+r.value.f;
```

```
        }
        else
                semantic_error(18,node->line);
        break;
case N_EXP_SUB :
        result=getTypeAndValueOfExpression(node->llink);
        r=getTypeAndValueOfExpression(node->rlink);
        if (result.type==int_type && r.type==int_type){
                result.type=int_type;
                result.value.i=result.value.i - r.value.i;
        }
        else if (result.type==int_type && r.type==float_type){
                result.type=float_type;
                result.value.f=result.value.i - r.value.f;}
        else if (result.type==float_type && r.type==int_type){
                result.type=float_type;
                result.value.f=result.value.f-r.value.i;
        }
        else if (result.type==float_type && r.type==float_type){
                result.type=float_type;
                result.value.f=result.value.f-r.value.f;
        }
        else
                semantic_error(18,node->line);
        break;
case N_EXP_LSS :
```

```
                case N_EXP_GTR :
                case N_EXP_LEQ :
                case N_EXP_GEQ :
                case N_EXP_NEQ :
                case N_EXP_EQL :
                case N_EXP_AND :
                case N_EXP_OR :
                case N_EXP_ASSIGN :
                        semantic_error(18,node->line);
                        break;
                default :
                        semantic_error(90,node->line);
                        break;
        }// close switch statement
        return (result);
}

// simplified error procedure.
void semantic_error(int i, int ll, char *s) {
        semantic_err++;
        printf("ERROR num: %d, line: %d, identifier: %s₩n",i, ll, s);
}

void semantic_warning(int i, int ll)
{
        printf("WARNING num: %d, line: %d₩n",i, ll);
```

}