[Compiler]

과제 6번 : 시멘틱분석기 제작

25-1 유재우 교수님

20201740 소프트웨어학부 오승민

# Introduction

본 과제의 목적은 **C 언어 소스 프로그램에 대한 시멘틱 분석기 (Semantic Analyzer)** 를 직접 구현하고, 이를 바탕으로 다양한 선언문과 명령문을 포함한 테스트 프로그램의 의미적 정당성을 검사하는데 있습니다.

교재 7장 (시멘틱 분석기)에서는 시멘틱 분석기의 주요 역할을 다음과 같이 제시합니다.

1. 모든 명칭 (Identifier) 의 의미 해석 및 타입 분석

2. 수식 (expression) 내 연산자의 타입 규칙 위반 여부 검사

3. 선언문의 크기 계산 및 변수 / 파라미터의 주소 설정

4. 함수의 반환 타입 및 파라미터 일치 여부 확인

5. 모든 명령문에 대해 정당성 여부를 검토하는 정적 의미 검사

이러한 기능들은 단순한 문법 검사 (구문 분석, syntax analysis) 이상의 역할을 수행하는 것으로, 프로그램의 논리적 오류까지도 사전에 탐지할 수 있습니다.

하여, 본 과제에서는 Yacc 기반의 신택스 트리(AST) 를 바탕으로 semantic\_analysis 함수를 통해 트리를 순회하며 의미 분석을 수행하고, 그 결과를 바탕으로 오류 메시지를 출력하거나 테이블과 AST 구조를 시각화합니다. 교수님께서 제공해주신 print\_sem\_ast() 함수는 최종적으로 구성된 시멘틱 분석 트리와 타입 / 심볼 테이블이 올바르게 생성되었는지를 검증하는 데 사용됩니다.

단순한 문법 분석을 넘어, 실제 C 언어 컴파일러에서 수행되는 정적 의미 분석 과정을 직접 구현하고 실험해보는 과정으로, 실질적인 컴파일러의 핵심 처리 흐름을 이해하고 응용해보는 데 큰 의의를 둘 것으로 보입니다.

# Main Code

**oh.l**

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"

extern YYSTYPE yylval;

extern int line\_no;

extern A\_ID \*current\_id;

char \*makeString();

int checkIdentifier();

%}

%%

{ws} { }

{line} { line\_no++; }

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(TYPE\_IDENTIFIER); }

union { return(UNION\_SYM); }

while { return(WHILE\_SYM); }

"++" { return(PLUSPLUS); }

"--" { return(MINUSMINUS); }

"->" { return(ARROW); }

"<" { return(LSS); }

">" { return(GTR); }

"<=" { return(LEQ); }

">=" { return(GEQ); }

"==" { return(EQL); }

"!=" { return(NEQ); }

"&&" { return(AMPAMP); }

"||" { return(BARBAR); }

"..." { return(DOTDOTDOT); }

"(" { return(LP); }

")" { return(RP); }

"[" { return(LB); }

"]" { return(RB); }

"{" { return(LR); }

"}" { return(RR); }

":" { return(COLON); }

"." { return(PERIOD); }

"," { return(COMMA); }

"!" { return(EXCL); }

"?" { return(QUESTION); }

"\*" { return(STAR); }

"/" { return(SLASH); }

"%" { return(PERCENT); }

"&" { return(AMP); }

";" { return(SEMICOLON); }

"+" { return(PLUS); }

"-" { return(MINUS); }

"=" { return(ASSIGN); }

{digit}+ { yylval=atoi(yytext); return(INTEGER\_CONSTANT); }

{digit}+\.{digit}+ { yylval=makeString(yytext); return(FLOAT\_CONSTANT); }

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

\"([^"\n]|\\["\n])\*\" { yylval=makeString(yytext); return(STRING\_LITERAL); }

\'([^'\n]|\'\')\' { yylval=\*(yytext+1); return(CHARACTER\_CONSTANT); }

%%

char \*makeString(char \*s) {

char \*t;

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

strcpy(t,s);

return(t);

}

int checkIdentifier(char \*s) {

A\_ID \*id;

char \*t;

id = current\_id;

while (id) {

if (strcmp(id->name, s) == 0) break;

id = id->prev;

}

if(id==0) {

yylval = makeString(s);

return(IDENTIFIER);

}

else if (id->kind == ID\_TYPE) {

yylval = id->type;

return(TYPE\_IDENTIFIER);

}

else {

yylval = id->name;

return(IDENTIFIER);

}

}

**oh.y**

%{

#define YYSTYPE\_IS\_DECLARED 1

typedef long YYSTYPE;

#include "type.h"

#include "oh.h"

#include <stdio.h>

#include <stdlib.h>

extern int line\_no, syntax\_err;

extern A\_NODE \*root;

extern A\_ID \*current\_id;

extern int current\_level;

extern A\_TYPE \*int\_type, \*float\_type, \*char\_type, \*void\_type, \*string\_type;

int yyerror(char \*s);

int yylex();

int yywrap();

%}

%start program

%token IDENTIFIER TYPE\_IDENTIFIER FLOAT\_CONSTANT INTEGER\_CONSTANT STRING\_LITERAL CHARACTER\_CONSTANT

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

PLUSPLUS MINUSMINUS ARROW LSS GTR LEQ GEQ EQL NEQ AMPAMP BARBAR DOTDOTDOT

LP RP LB RB LR RR COLON PERIOD COMMA EXCL STAR SLASH PERCENT AMP QUESTION SEMICOLON

PLUS MINUS ASSIGN

%%

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;}

;

declaration

: declaration\_specifiers init\_declarator\_list\_opt 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\_opt

: /\* empty \*/ {$$=makeDummyIdentifier();}

| init\_declarator\_list {$$=$1;}

;

init\_declarator\_list

: init\_declarator {$$=$1;}

| init\_declarator\_list COMMA init\_declarator

{$$=linkDeclaratorList($1,$3);}

;

init\_declarator

: declarator {$$=$1;}

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

;

initializer

: constant\_expression {$$=makeNode(N\_INIT\_LIST\_ONE,NIL,$1,NIL);}

| LR initializer\_list 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);}

;

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,$2);}

;

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 expression {$$=setDeclaratorInit($2,$4);}

;

declarator

: pointer direct\_declarator {$$=setDeclaratorElementType($2,$1);}

| direct\_declarator {$$=$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));}

;

constant\_expression\_opt

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

| constant\_expression {$$=$1;}

;

parameter\_type\_list\_opt

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

| parameter\_type\_list {$$=$1;}

;

parameter\_type\_list

: parameter\_list {$$=$1;}

| parameter\_list DOTDOTDOT

{$$=linkDeclaratorList($1,setDeclaratorKind(makeDummyIdentifier(),ID\_PARAM));}

;

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));}

;

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\_opt

{$$=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);}

;

relational\_expression

: additive\_expression {$$=$1;}

| relational\_expression LSS additive\_expression

{$$=makeNode(N\_EXP\_LSS,$1,NIL,$3);}

| relational\_expression GTR additive\_expression

{$$=makeNode(N\_EXP\_GTR,$1,NIL,$3);}

| relational\_expression LEQ additive\_expression

{$$=makeNode(N\_EXP\_LEQ,$1,NIL,$3);}

| relational\_expression GEQ additive\_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);}

;

logical\_and\_expression

: bitwise\_or\_expression {$$=$1;}

| logical\_and\_expression AMPAMP bitwise\_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);}

;

bitwise\_or\_expression

: bitwise\_xor\_expression {$$=$1;}

;

bitwise\_xor\_expression

: bitwise\_and\_expression {$$=$1;}

;

bitwise\_and\_expression

: equality\_expression {$$=$1;}

;

constant\_expression

: expression {$$=$1;}

;

expression

: assignment\_expression {$$=$1;}

;

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\_opt statement\_list\_opt RR

{checkForwardReference(); $$=makeNode(N\_STMT\_COMPOUND,$3,NIL,$4); current\_id=$2; current\_level--;}

;

declaration\_list\_opt

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

| declaration\_list {$$=$1;}

declaration\_list

: declaration {$$=$1;}

| declaration\_list declaration {$$=linkDeclaratorList($1,$2);}

;

statement\_list\_opt

: /\* empty \*/ {$$=makeNode(N\_STMT\_LIST\_NIL,NIL,NIL,NIL);}

| statement\_list {$$=$1;}

statement\_list

: statement {$$=makeNode(N\_STMT\_LIST,$1,NIL,

makeNode(N\_STMT\_LIST\_NIL,NIL,NIL,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 for\_expression RP statement

{$$=makeNode(N\_STMT\_FOR,$3,NIL,$5);}

;

for\_expression

: expression\_opt SEMICOLON expression\_opt SEMICOLON expression\_opt

{$$=makeNode(N\_FOR\_EXP,$1,$3,$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);}

;

assignment\_expression

: conditional\_expression {$$=$1;}

| unary\_expression ASSIGN expression {$$=makeNode(N\_EXP\_ASSIGN,$1,NIL,$3);}

;

conditional\_expression

: logical\_or\_expression {$$=$1;}

;

%%

extern char \*yytext;

int main(int argc, char \*\*argv) {

initialize();

yyparse();

if(syntax\_err) exit(1);

print\_ast(root);

printf("성공적으로 파싱을 마쳤습니다.\n프로그램을 종료합니다.\n");

}

int yyerror(char \*s) {

fprintf(stderr, "에러 발생 : %s , %s\n", s, yytext);

exit(1);

}

int yywrap() {

return (1);

}

**oh.h**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "y.tab.h"

extern *char* \*yytext;

A\_TYPE \*int\_type, \*char\_type, \*void\_type, \*float\_type, \*string\_type;

A\_NODE \*root;

A\_ID \*current\_id=NIL;

*int* syntax\_err=0;

*int* line\_no=1;

*int* current\_level=0;

A\_NODE \*makeNode(NODE\_NAME,A\_NODE \*,A\_NODE \*,A\_NODE \*);

A\_NODE \*makeNodeList(NODE\_NAME,A\_NODE \*,A\_NODE \*);

A\_ID \*makeIdentifier(*char* \*);

A\_ID \*makeDummyIdentifier();

A\_TYPE \*makeType(T\_KIND);

A\_SPECIFIER \*makeSpecifier(A\_TYPE \*,S\_KIND);

A\_ID \*searchIdentifier(*char* \*,A\_ID \*);

A\_ID \*searchIdentifierAtCurrentLevel(*char* \*,A\_ID \*);

A\_SPECIFIER \*updateSpecifier(A\_SPECIFIER \*,A\_TYPE \*,S\_KIND);

*void* checkForwardReference();

*void* setDefaultSpecifier(A\_SPECIFIER \*);

A\_ID \*linkDeclaratorList(A\_ID \*,A\_ID \*);

A\_ID \*getIdentifierDeclared(*char* \*);

A\_TYPE \*getTypeOfStructOrEnumRefIdentifier(T\_KIND,*char* \*,ID\_KIND);

A\_ID \*setDeclaratorInit(A\_ID \*,A\_NODE \*);

A\_ID \*setDeclaratorKind(A\_ID \*,ID\_KIND);

A\_ID \*setDeclaratorType(A\_ID \*,A\_TYPE \*);

A\_ID \*setDeclaratorElementType(A\_ID \*,A\_TYPE \*);

A\_ID \*setDeclaratorTypeAndKind(A\_ID \*,A\_TYPE \*,ID\_KIND);

A\_ID \*setDeclaratorListSpecifier(A\_ID \*,A\_SPECIFIER \*);

A\_ID \*setFunctionDeclaratorSpecifier(A\_ID \*,A\_SPECIFIER \*);

A\_ID \*setFunctionDeclaratorBody(A\_ID \*, A\_NODE \*);

A\_ID \*setParameterDeclaratorSpecifier(A\_ID \*,A\_SPECIFIER \*);

A\_ID \*setStructDeclaratorListSpecifier(A\_ID \*,A\_TYPE \*);

A\_TYPE \*setTypeNameSpecifier(A\_TYPE \*,A\_SPECIFIER \*);

A\_TYPE \*setTypeElementType(A\_TYPE \*,A\_TYPE \*);

A\_TYPE \*setTypeField(A\_TYPE \*,A\_ID \*);

A\_TYPE \*setTypeExpr(A\_TYPE \*,A\_NODE \*);

A\_TYPE \*setTypeAndKindOfDeclarator(A\_TYPE \*,ID\_KIND,A\_ID \*);

A\_TYPE \*setTypeStructOrEnumIdentifier(T\_KIND,*char* \*,ID\_KIND);

BOOLEAN isNotSameFormalParameters(A\_ID \*,A\_ID \*);

BOOLEAN isNotSameType(A\_TYPE \*,A\_TYPE \*);

BOOLEAN isPointerOrArrayType(A\_TYPE \*);

*void* syntax\_error();

*void* initialize();

// Make new node for syntax tree

A\_NODE \*makeNode (NODE\_NAME *n*, A\_NODE \**a*, A\_NODE \**b*, A\_NODE \**c*) {

A\_NODE \*m;

m = (A\_NODE\*)malloc(sizeof(A\_NODE));

m->name = n;

m->llink = a;

m->clink = b;

m->rlink = c;

m->type = NIL;

m->line = line\_no;

m->value = 0;

return (m);

}

A\_NODE \*makeNodeList (NODE\_NAME *n*, A\_NODE \**a*, A\_NODE \**b*) {

A\_NODE \*m,\*k;

k = a;

while (k->rlink) k = k->rlink;

m = (A\_NODE\*)malloc(sizeof(A\_NODE));

m->name = k->name;

m->llink = NIL;

m->clink = NIL;

m->rlink = NIL;

m->type = NIL;

m->line = line\_no;

m->value = 0;

k->name = n;

k->llink = b;

k->rlink = m;

return (a);

}

// Make a new declarator for identifier

A\_ID \*makeIdentifier(*char* \**s*) {

A\_ID \*id;

id = malloc(sizeof(A\_ID));

id->name = s;

id->kind = 0;

id->specifier = 0;

id->level = current\_level;

id->address = 0;

id->init = NIL;

id->type = NIL;

id->link = NIL;

id->line = line\_no;

id->value = 0;

id->prev = current\_id;

current\_id = id;

return(id);

}

// Make a new declarator for dummy identifier

A\_ID \*makeDummyIdentifier() {

A\_ID \*id;

id = malloc(sizeof(A\_ID));

id->name = "";

id->kind = 0;

id->specifier = 0;

id->level = current\_level;

id->address = 0;

id->init = NIL;

id->type = NIL;

id->link = NIL;

id->line = line\_no;

id->value = 0;

id->prev = 0;

return(id);

}

// Make a new type

A\_TYPE \*makeType(T\_KIND *k*) {

A\_TYPE \*t;

t = malloc(sizeof(A\_TYPE));

t->kind = k;

t->size = 0;

t->local\_var\_size = 0;

t->element\_type = NIL;

t->field = NIL;

t->expr = NIL;

t->check = FALSE;

t->prt = FALSE;

t->line = line\_no;

return(t);

}

// Make a new specifier

A\_SPECIFIER \*makeSpecifier(A\_TYPE \**t*, S\_KIND *s*) {

A\_SPECIFIER \*p;

p = malloc(sizeof(A\_SPECIFIER));

p->type = t;

p->stor = s;

p->line = line\_no;

return(p);

}

A\_ID \*searchIdentifier(*char* \**s*, A\_ID \**id*) {

while (id) {

if(strcmp(id->name,s) == 0) break;

id = id->prev;

}

return (id);

}

A\_ID \*searchIdentifierAtCurrentLevel(*char* \**s*, A\_ID \**id*) {

while (id) {

if (id->level < current\_level) return (NIL);

if (strcmp(id->name,s) == 0) break;

id = id->prev;

}

return (id);

}

*void* checkForwardReference() {

A\_ID \*id;

A\_TYPE \*t;

id = current\_id;

while (id) {

if (id->level < current\_level) break;

t = id->type;

if (id->kind == ID\_NULL) syntax\_error(31,id->name);

else if ((id->kind == ID\_STRUCT || id->kind == ID\_ENUM)

&& t->field == NIL)

syntax\_error(32,id->name);

id = id->prev;

}

}

// Set default specifier

*void* setDefaultSpecifier(A\_SPECIFIER \**p*) {

A\_TYPE \*t;

if (p->type == NIL) p->type = int\_type;

if (p->stor == S\_NULL) p->stor = S\_AUTO;

}

// Merge & Update specifier

A\_SPECIFIER \*updateSpecifier(A\_SPECIFIER \**p*, A\_TYPE \**t*, S\_KIND *s*) {

if (t) {

if (p->type) {

if (p->type == t) ;

else syntax\_error(24, NULL);

}

else p->type = t;

}

if (s) {

if (p->stor) {

if(s == p->stor) ;

else syntax\_error(24, NULL);

}

else p->stor = s;

}

return (p);

}

// Link two declarator list id1 & id2

A\_ID \*linkDeclaratorList(A\_ID \**id1*, A\_ID \**id2*) {

A\_ID \*m = id1;

if (id1 == NIL) return (id2);

while (m->link) m = m->link;

m->link = id2;

return (id1);

}

// Check if identifier is already declared in primary expression

A\_ID \*getIdentifierDeclared(*char* \**s*) {

A\_ID \*id;

id = searchIdentifier(s, current\_id);

if (id == NIL) syntax\_error(13, s);

return (id);

}

// Get type of struct identifier

A\_TYPE \*getTypeOfStructOrEnumRefIdentifier(T\_KIND *k*, *char* \**s*, ID\_KIND *kk*) {

A\_TYPE \*t;

A\_ID \*id;

id = searchIdentifier(s, current\_id);

if (id) {

if (id->kind == kk && id->type->kind == k) return (id->type);

else syntax\_error(11, s);

}

// Make a new struct (or enum) identifier

t = makeType(k);

id = makeIdentifier(s);

id->kind = kk;

id->type = t;

return (t);

}

// Set declarator init (expression tree)

A\_ID \*setDeclaratorInit(A\_ID \**id*, A\_NODE \**n*) {

id->init = n;

return (id);

}

// Set declarator kind

A\_ID \*setDeclaratorKind(A\_ID \**id*, ID\_KIND *k*) {

A\_ID \*a;

a = searchIdentifierAtCurrentLevel(id->name, id->prev);

if (a) syntax\_error(12, id->name);

id->kind = k;

return (id);

}

// Set declarator type

A\_ID \*setDeclaratorType(A\_ID \**id*, A\_TYPE \**t*) {

id->type = t;

return (id);

}

// Set declarator type (or element type)

A\_ID \*setDeclaratorElementType(A\_ID \**id*, A\_TYPE \**t*) {

A\_TYPE \*tt;

if (id->type == NIL) id->type = t;

else {

tt = id->type;

while (tt->element\_type) tt = tt->element\_type;

tt->element\_type = t;

}

return (id);

}

// Set declarator element type and kind

A\_ID \*setDeclaratorTypeAndKind(A\_ID \**id*, A\_TYPE \**t*, ID\_KIND *k*) {

id = setDeclaratorElementType(id, t);

id = setDeclaratorKind(id, k);

return (id);

}

// Check function declarator and return type

A\_ID \*setFunctionDeclaratorSpecifier(A\_ID \**id*, A\_SPECIFIER \**p*) {

A\_ID \*a;

// Check storage class

if (p->stor) syntax\_error(25, NULL);

setDefaultSpecifier(p);

// Check function identifier immediately befor '('

if (id->type == 0 || id->type->kind != T\_FUNC) {

syntax\_error(21, NULL);

return (id);

}

else {

id = setDeclaratorElementType(id, p->type);

id->kind = ID\_FUNC;

}

// Check redeclaration

a = searchIdentifierAtCurrentLevel(id->name, id->prev);

if (a)

if (a->kind != ID\_FUNC || a->type->expr) syntax\_error(12, id->name);

else {

// Check prototype: parameters and return type

if (isNotSameFormalParameters(a->type->field, id->type->field))

syntax\_error(22, id->name);

if (isNotSameType(a->type->element\_type, id->type->element\_type))

syntax\_error(26, a->name);

}

// Change parameter scope and check empty name

a = id->type->field;

while (a) {

if (strlen(a->name)) current\_id = a;

else if (a->type) syntax\_error(23, NULL);

a = a->link;

}

return (id);

}

A\_ID \*setFunctionDeclaratorBody(A\_ID \**id*, A\_NODE \**n*) {

id->type->expr = n;

return (id);

}

// Set declarator\_list type and kind based on storage class

A\_ID \*setDeclaratorListSpecifier(A\_ID \**id*, A\_SPECIFIER \**p*) {

A\_ID \*a;

setDefaultSpecifier(p);

a = id;

while (a) {

if (strlen(a->name) && searchIdentifierAtCurrentLevel(a->name, a->prev))

syntax\_error(12, a->name);

a = setDeclaratorElementType(a, p->type);

if (p->stor == S\_TYPEDEF) a->kind = ID\_TYPE;

else if (a->type->kind == T\_FUNC) a->kind = ID\_FUNC;

else a->kind = ID\_VAR;

a->specifier = p->stor;

if (a->specifier == S\_NULL) a->specifier = S\_AUTO;

a = a->link;

}

return (id);

}

// Set declarator\_list type and kind

A\_ID \*setParameterDeclaratorSpecifier(A\_ID \**id*, A\_SPECIFIER \**p*) {

// Check redeclaration

if (searchIdentifierAtCurrentLevel(id->name, id->prev))

syntax\_error(12, id->name);

// Check parameter storage class && void type

if (p->stor || p->type == void\_type)

syntax\_error(14, NULL);

setDefaultSpecifier(p);

id = setDeclaratorElementType(id, p->type);

id->kind = ID\_PARAM;

return (id);

}

A\_ID \*setStructDeclaratorListSpecifier(A\_ID \**id*, A\_TYPE \**t*) {

A\_ID \*a;

a = id;

while (a) {

if (searchIdentifierAtCurrentLevel(a->name, a->prev))

syntax\_error(12, a->name);

a = setDeclaratorElementType(a, t);

a->kind = ID\_FIELD;

a = a->link;

}

return (id);

}

// Set type name specifier

A\_TYPE \*setTypeNameSpecifier(A\_TYPE \**t*, A\_SPECIFIER \**p*) {

// Check storage class in type name

if (p->stor) syntax\_error(20, NULL);

setDefaultSpecifier(p);

t = setTypeElementType(t, p->type);

return (t);

}

// Set type element type

A\_TYPE \*setTypeElementType(A\_TYPE \**t*, A\_TYPE \**s*) {

A\_TYPE \*q;

if (t == NIL) return (s);

q = t;

while (q->element\_type) q = q->element\_type;

q->element\_type = s;

return (t);

}

// Set type field

A\_TYPE \*setTypeField(A\_TYPE \**t*, A\_ID \**n*) {

t->field = n;

return (t);

}

// Set type initial value (expression tree)

A\_TYPE \*setTypeExpr(A\_TYPE \**t*, A\_NODE \**n*) {

t->expr = n;

return (t);

}

// Set type of struct identifier

A\_TYPE \*setTypeStructOrEnumIdentifier(T\_KIND *k*, *char* \**s*, ID\_KIND *kk*) {

A\_TYPE \*t;

A\_ID \*id, \*a;

// Check redeclaration or forward declaration

a = searchIdentifierAtCurrentLevel(s, current\_id);

if (a)

if (a->kind == kk && a->type->kind == k)

if (a->type->field) syntax\_error(12, s);

else return (a->type);

else syntax\_error(12, s);

// Make a new struct (or enum) identifier

id = makeIdentifier(s);

t = makeType(k);

id->type = t;

id->kind = kk;

return (t);

}

// Set type and kind of identifier

A\_TYPE \*setTypeAndKindOfDeclarator(A\_TYPE \**t*, ID\_KIND *k*, A\_ID \**id*) {

if (searchIdentifierAtCurrentLevel(id->name, id->prev))

syntax\_error(12, id->name);

id->type = t;

id->kind = k;

return (t);

}

// Check function parameters with prototype

BOOLEAN isNotSameFormalParameters(A\_ID \**a*, A\_ID \**b*) {

if (a == NIL) // no parameters in prototype..

return (FALSE);

while (a) {

if (b == NIL || isNotSameType(a->type, b->type)) return (TRUE);

a = a->link;

b = b->link;

}

if (b) return (TRUE);

else return (FALSE);

}

BOOLEAN isNotSameType(A\_TYPE \**t1*, A\_TYPE \**t2*) {

if (isPointerOrArrayType(t1) || isPointerOrArrayType(t2))

return (isNotSameType(t1->element\_type, t2->element\_type));

else return (t1 != t2);

}

BOOLEAN isPointerOrArrayType(A\_TYPE \**t*) {

if (t && (t->kind == T\_POINTER || t->kind == T\_ARRAY))

return(TRUE);

else

return(FALSE);

}

*void* initialize() {

// Primitive data types

int\_type = setTypeAndKindOfDeclarator(

makeType(T\_ENUM), ID\_TYPE, makeIdentifier("int"));

float\_type = setTypeAndKindOfDeclarator(

makeType(T\_ENUM), ID\_TYPE, makeIdentifier("float"));

char\_type = setTypeAndKindOfDeclarator(

makeType(T\_ENUM), ID\_TYPE, makeIdentifier("char"));

void\_type = setTypeAndKindOfDeclarator(

makeType(T\_VOID), ID\_TYPE, makeIdentifier("void"));

string\_type = setTypeElementType(makeType(T\_POINTER), char\_type);

int\_type->size = 4;

int\_type->check = TRUE;

float\_type->size = 4;

float\_type->check = TRUE;

char\_type->size = 1;

char\_type->check = TRUE;

void\_type->size = 0;

void\_type->check = TRUE;

string\_type->size = 4;

string\_type->check = TRUE;

// printf(char \*, ...) library function

setDeclaratorTypeAndKind(

makeIdentifier("printf"),

setTypeField(

setTypeElementType(makeType(T\_FUNC), void\_type),

linkDeclaratorList(

setDeclaratorTypeAndKind(makeDummyIdentifier(), string\_type, ID\_PARAM),

setDeclaratorKind(makeDummyIdentifier(), ID\_PARAM)

)

),

ID\_FUNC);

// scanf(char \*, ...) library function

setDeclaratorTypeAndKind(

makeIdentifier("scanf"),

setTypeField(

setTypeElementType(makeType(T\_FUNC), void\_type),

linkDeclaratorList(

setDeclaratorTypeAndKind(makeDummyIdentifier(), string\_type, ID\_PARAM),

setDeclaratorKind(makeDummyIdentifier(), ID\_PARAM)

)

),

ID\_FUNC);

// malloc(int) library function

setDeclaratorTypeAndKind(

makeIdentifier("malloc"),

setTypeField(

setTypeElementType(makeType(T\_FUNC), string\_type),

setDeclaratorTypeAndKind(makeDummyIdentifier(), int\_type, ID\_PARAM)

),

ID\_FUNC);

}

*void* syntax\_error(*int* *i*, *char* \**s*) {

syntax\_err++;

printf("line %d: syntax error:", line\_no);

switch (i) {

case 11 :

printf("illegal referencing struct or union identifier %s", s);

break;

case 12 :

printf("redeclaration of identifier %s", s);

break;

case 13 :

printf("undefined identifier %s", s);

break;

case 14 :

printf("illegal type specifier in formal parameter");

break;

case 20 :

printf("illegal storage class in type specifiers");

break;

case 21 :

printf("illegal function declarator");

break;

case 22 :

printf("conflicting parm type in prototype function %s", s);

break;

case 23 :

printf("empty parameter name");

break;

case 24 :

printf("illegal declaration specifiers");

break;

case 25 :

printf("illegal function specifiers");

break;

case 26 :

printf("illegal or conflicting return type in function %s", s);

break;

case 31 :

printf("undefined type for identifier %s", s);

break;

case 32 :

printf("incomplete forward reference for identifier %s", s);

break;

default :

printf("unknown");

break;

}

if (strlen(yytext) == 0) printf(" at end\n");

else printf(" near %s\n", yytext);

}

**SEMANTIC.C**

#include "type.h"

*float* atof();

*void* semantic\_analysis(A\_NODE \*);

*void* set\_literal\_address(A\_NODE \*);

*int* put\_literal(A\_LITERAL, *int*);

*void* sem\_program(A\_NODE \*);

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 \*);

*int* sem\_A\_TYPE(A\_TYPE \*);

*int* sem\_declaration\_list(A\_ID \*, *int*);

*int* sem\_declaration(A\_ID \*, *int*);

*void* sem\_arg\_expr\_list(A\_NODE \*, A\_ID \*);

A\_ID \*getStructFieldIdentifier(A\_TYPE \*, *char* \*);

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 isStructOrUnionType(A\_TYPE \*);

BOOLEAN isFunctionType(A\_TYPE \*);

BOOLEAN isScalarType(A\_TYPE \*);

BOOLEAN isPointerType(A\_TYPE \*);

BOOLEAN isPointerOrArrayType\_semantic(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*);

*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*) {

*int* i;

switch(node->name) {

case N\_PROGRAM :

i = sem\_declaration\_list(node->*clink*, 12);

node->value = global\_address;

break;

default :

semantic\_error(90, node->*line*);

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\_PARAM :

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*); // index of literal table

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*); // index of literal table

result = string\_type;

break;

case N\_EXP\_ARRAY :

t1 = sem\_expression(node->*llink*);

t2 = sem\_expression(node->*rlink*);

// usual binary conversion

t = convertUsualBinaryConversion(node);

t1 = node->llink->type;

t2 = node->rlink->type;

if (isPointerOrArrayType\_semantic(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\_STRUCT :

t = sem\_expression(node->*llink*);

id = getStructFieldIdentifier(t, node->*rlink*);

if (id) {

result = id->type;

if (node->llink->value && !isArrayType(result))

lvalue=TRUE;

}

else

semantic\_error(37, node->*line*);

node->rlink = id;

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*);

// usual unary conversion

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 experssion 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\_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

if ((node->clink->name != N\_EXP\_IDENT ||

((A\_ID \*)node->clink->clink)->kind != ID\_PARAM) &&

(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 = int\_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 (isStructOrUnionType(result) || 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))

t = 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))

t = convertUsualBinaryConversion(node);

else if (!isCompatiblePointerType(t1, t2) &&

(!isPointerType(t1) || !isConstantZeroExp(node->*rlink*)) &&

(!isPointerType(t2) || !isConstantZeroExp(node->*llink*)))

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 if modifiable lvalue

if (!isModifiableLvalue(node->*llink*))

semantic\_error(60, node->*line*);

t = sem\_expression(node->*rlink*);

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 || isVoidType(node->type) || 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)

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\_SWITCH :

t = sem\_expression(node->*llink*);

if (!isIntegralType(t))

semantic\_error(50, node->*line*);

local\_size = sem\_statement(node->*rlink*, addr, ret, TRUE, TRUE, cnt);

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\_STRUCT :

id=t->field;

while (id) {

if(t == id->type) {

semantic\_error(94, id->line);

break;

}

if(isStructOrUnionType(id->type)){

A\_ID \*id2 = id->type->field;

while(id2){

if(t == id2->type){

semantic\_error(94, id2->line);

break;

}

id2 = id2->link;

}

}

result+=sem\_declaration(id,result);

id = id->link;

}

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 and size

if (t->expr) { // skip prototype declaration

i = i+sem\_statement(t->expr, i, t->element\_type, FALSE, FALSE, FALSE);

}

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) {

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\_PARAM :

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 \*getStructFieldIdentifier(A\_TYPE \**t*, *char* \**s*) {

A\_ID \*id = NIL;

if (isStructOrUnionType(t)) {

id = t->field;

while (id) {

if (strcmp(id->name, s) == 0)

break;

id = id->link;

}

}

return (id);

}

A\_ID \*getPointerFieldIdentifier(A\_TYPE \**t*, *char* \**s*) {

A\_ID \*id = NIL;

if (t && t->kind == T\_POINTER) {

t = t->element\_type;

if (isStructOrUnionType(t)) {

id = t->field;

while (id) {

if (strcmp(id->name, s) == 0)

break;

id = id->link;

}

}

}

return (id);

}

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 (isCompatiblePointerType(t1->element\_type, t2->element\_type));

else

return (FALSE);

}

A\_NODE \*convertScalarToInteger(A\_NODE \**node*) {

if (isFloatType(node->type)) {

semantic\_error(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\_error(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\_error(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\_error(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\_error(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 (isStructOrUnionType(t1) && 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 <--- t2

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\_FUNC)

return (TRUE);

else

return (FALSE);

}

BOOLEAN isFunctionType(A\_TYPE \**t*) {

if (t && t->kind == T\_FUNC)

return (TRUE);

else

return (FALSE);

}

BOOLEAN isStructOrUnionType(A\_TYPE \**t*) {

if (t && (t->kind == T\_STRUCT || t->kind == T\_UNION))

return (TRUE);

else

return (FALSE);

}

BOOLEAN isPointerType(A\_TYPE \**t*) {

if (t && t->kind == T\_POINTER)

return (TRUE);

else

return (FALSE);

}

BOOLEAN isPointerOrArrayType\_semantic(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\_STRUCT :

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;

}

return (result);

}

*void* semantic\_error(*int* *i*, *int* *ll*, *char* \**s*) {

semantic\_err++;

printf("\*\*\* semantic error at line %d: ", ll);

switch (i) {

//errors in expression

case 13:

printf("%arith type expr required in unary operation\n");

break;

case 18:

printf("illegal constant expression\n");

break;

case 19:

printf("illegal identifier %s in constant expression\n", s);

break;

case 21:

printf("illegal type in function call expression\n");

break;

case 24:

printf("incompatible type in additive expression\n");

break;

case 27:

printf("scalar type expr required in expression\n");

break;

case 28:

printf("arith type expression required in binary operation\n");

break;

case 29:

printf("integral type expression required in expression\n");

break;

case 31:

printf("pointer type expr required in pointer operation\n");

break;

case 32:

printf("array type required in array expression\n");

break;

case 34:

printf("too many arguments in function call\n");

break;

case 35:

printf("too few arguments in function call\n");

break;

case 37:

printf("illegal struct field identifier in struct reference expr\n");

break;

case 38:

printf("illegal kind of identifier %s in expression\n", s);

break;

case 39:

printf("illegal type size in sizeof operation\n");

break;

case 40:

printf("illegal expression type in relational operation\n");

break;

// case 41:

// printf("incompatible type in literal\n");

// break;

// errors in statement

case 49:

printf("scalar type expr required in middle of for-expr\n");

break;

case 50:

printf("integral type expression required in statement\n");

break;

case 51:

printf("illegal expression type in case label\n");

break;

case 57:

printf("not permitted type conversion in return expression\n");

break;

case 58:

printf("not permitted type casting in expression\n");

break;

case 59:

printf("not permitted type conversion in argument\n");

break;

case 60:

printf("expression is not an lvalue\n");

break;

case 71:

printf("case label not within a switch statement\n");

break;

case 72:

printf("default label not within a switch statement\n");

break;

case 73:

printf("break statement not within loop or switch stmt\n");

break;

case 74:

printf("continue statement not within a loop\n");

break;

// errors in type & declarator

case 80:

printf("undefined type\n");

break;

case 81:

printf("integer type expression required in enumerator\n");

break;

case 82:

printf("illegal array size or type\n");

break;

case 83:

printf("illegal element type of array declarator\n");

break;

case 84:

printf("illegal type in struct or union field\n");

break;

case 85:

printf("invalid function return type\n");

break;

case 86:

printf("illegal array size or empty array\n");

break;

case 89:

printf("unknown identifier kind: %s\n", s);

break;

// misc errors

case 90:

printf("fatal compiler error in parse result\n");

break;

case 93:

printf("too many literals in source program\n");

break;

case 94:

printf("self-referential struct declartion\n", i);

break;

default:

printf("unknown\n");

break;

}

}

*void* semantic\_warning(*int* *i*, *int* *ll*) {

printf("--- warning at line %d: ", ll);

switch (i) {

case 11:

printf("incompatible types in assignment expression\n");

break;

case 12:

printf("incompatible types in argument or return expr\n");

break;

case 14:

printf("incompatible types in binary expression\n");

break;

case 16:

printf("integer type expression is required\n");

break;

default:

printf("unknown\n");

break;

}

}

**main.c**

#include <stdio.h>

#include <stdlib.h>

#include "type.h"

extern *int* syntax\_err;

extern *int* semantic\_err;

extern A\_NODE \*root;

*void* initialize();

*void* print\_ast();

*void* print\_sem\_ast();

*void* semantic\_analysis();

*int* main(*int* *argc*, *char* \*\**argv*) {

initialize();

yyparse();

if(syntax\_err) exit(1);

print\_ast(root);

semantic\_analysis(root);

if (semantic\_err) exit(1);

print\_sem\_ast(root);

printf("성공적으로 파싱을 마쳤습니다.\n프로그램을 종료합니다.\n");

}

**구조체의 자기 참조 여부 검사**

case T\_STRUCT :

id=t->field;

while (id) {

if(t == id->type) {

semantic\_error(99, id->line);

break;

}

if(isStructOrUnionType(id->type)){

A\_ID \*id2 = id->type->field;

while(id2){

if(t == id2->type){

semantic\_error(94, id2->line);

break;

}

id2 = id2->link;

}

}

result+=sem\_declaration(id,result);

id = id->link;

}

break;

의미 분석 단계에서 T\_STRUCT 타입에 해당하는 구조체 정의에 대해, 구조체의 필드가 자기 자신을 직접 또는 간접적으로 참조하는지를 검사하여 의미 오류를 유발하도록 설계하였다. 이는 구조체가 무한 재귀적으로 자신을 포함하는 정의를 방지하고, 타입의 완전성을 보장한다.

예를 들어, struct S { struct T t; }; struct T { struct S x; } 처럼 두 구조체가 서로를 참조하는 순환 참조를 감지하고 94번 오류를 통해 보고된다.

# Test Case

**Success Test Case**

**( 편의상 Syntax Graph는 제외했습니다. )**

1. 기본 변수 선언

*int* main() {

*int* a;

*char* c;

*float* f;

return 0;

}

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

2. 배열 선언

*int* main() {

*int* arr[5];

*char* str[10];

return 0;

}

**텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

3. 중첩 배열 선언 및 초기화

*int* main() {

*int* mat[2][3] = { {1, 2, 3}, {4, 5, 6} };

return 0;

}

**텍스트, 스크린샷, 메뉴이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

4. 구조체 정의

*struct* Point {

*int* x;

*int* y;

};

*int* main() {

return 0;

}

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

5. 구조체 선언 + 변수

*struct* Point {

*int* x;

*int* y;

};

*int* main() {

*struct* Point p1;

return 0;

}

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

6. 구조체 초기화

*struct* Point {

*int* x;

*int* y;

};

*int* main() {

*struct* Point p2 = {1, 2};

return 0;

}

**텍스트, 스크린샷, 폰트, 메뉴이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

7. 공용체 선언

*union* Data {

*int* i;

*float* f;

*char* str[20];

};

*int* main() {

return 0;

}

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

8. 열거형 정의

*enum* Color {

RED,

GREEN,

BLUE

};

*int* main() {

return 0;

}

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

9. 열거형과 값 지정

*enum* Weekday {

MON = 1, TUE, WED = 5

};

*int* main() {

return 0;

}

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

10. 함수 선언

*int* sum(*int* *a*, *int* *b*);

*int* main() {

return 0;

}

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

11. 함수 정의

*int* sum(*int* *a*, *int* *b*) {

return *a* + *b*;

}

*int* main() {

*int* result = sum(3, 4);

return 0;

}

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

12. if 문

*int* main() {

*int* x = 5;

if (x > 0)

x--;

else

x++;

return 0;

}

텍스트, 스크린샷, 메뉴이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

13. while 반복문

*int* main() {

*int* n = 3;

while(n > 0) {

n--;

}

return 0;

}

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

14. do-while 반복문

*void* count(*int* *n*) {

do {

n--;

} while(n > 0);

}

*int* main() {

return 0;

}

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

15. for 반복문

*void* printNums() {

*int* i;

for(i = 0; i < 10; i++) {

;

}

}

*int* main() {

printNums();

return 0;

}

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

16. 다양한 표현식

*int* main() {

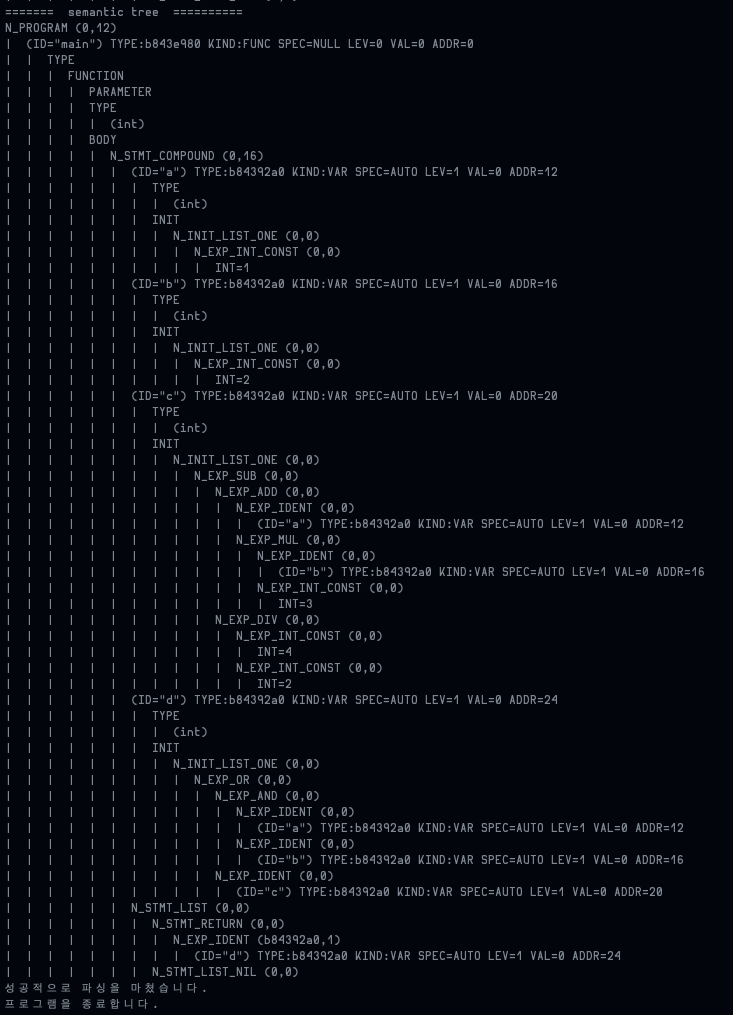
*int* a = 1, b = 2;

*int* c = a + b \* 3 - 4 / 2;

*int* d = a && b || c;

return d;

}



17. sizeof 연산자

*struct* Point {

*int* x;

*int* y;

};

*int* sizeTest() {

return sizeof(*int*) + sizeof(*struct* Point);

}

*int* main() {

sizeTest();

return 0;

}

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

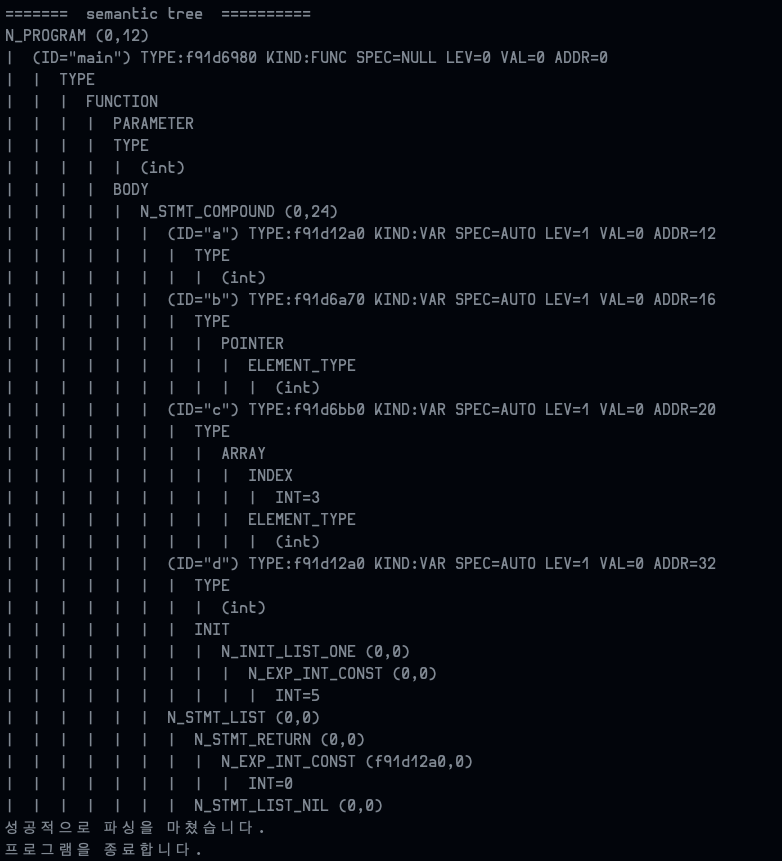
18. 복합 선언 리스트

*int* main() {

*int* a, \*b, c[3], d = 5;

return 0;

}



19. 함수 포인터

*int* func(*int* *num*) {

return 0;

}

*int* main(){

*int* (\*fptr)(*int*) = func;

return 0;

}

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

20. 배열 안의 구조체

*struct* Point {

*int* x;

*int* y;

};

*struct* Point points[2] = { {1,2}, {3,4} };

*int* main() {

*struct* Point s;

return 0;

}

텍스트, 스크린샷, 메뉴이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

21. 구조체 내부 포인터

*struct* Node {

*int* value;

*struct* Node \*next;

};

*int* main() {

*struct* Node n;

return 0;

}

텍스트, 스크린샷, 메뉴, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

위 21개의 예시 상황으로 파싱된 syntax Graph에서 제대로 구문을 분석할 수 있는지 확인했다.

**GCC vs Custom**

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

동일하게 문제 없이 파싱되는 것을 확인할 수 있다.

**Fail Test Case : gcc와 비교**

1. 단항 연산에 산술 타입 필요

-> 13번 에러

*struct* S { *int* x; };

*void* main() {

*struct* S s;

+s;

}

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

2. 잘못된 상수 수식

-> 18번 에러

*enum* E { A = &3 };

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

3. 상수식에 선언되지 않은 식별자 사용

-> 19번 에러

*int* x;

*enum* E { A = x };

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

4. 함수 호출 타입 오류

-> 21번 에러

*int* a;

*void* main() {

a();

}

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

5. 덧셈 수식 타입 불일치

-> 24번 에러

*struct* S { *int* x; };

*int* main() {

*struct* S s1, s2;

return s1 + s2;

}

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

6. if 문 조건이 scalar 아님

-> 27번 에러

*struct* S { *int* x; };

*void* main() {

*struct* S s;

!s;

}

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

7. 산술 이항 연산자 타입 오류

-> 28번 에러

*void* main() {

*int* c;

c = "abc" \* 5;

}

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

8. 배열 크기에 정수 아님

-> 29번 에러

*void* main() {

*float* f = 3.14;

*int* a[10];

a[f] = 1;

}

**텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

9. 인자 과다

-> 34번 에러

*void* f(*int* *a*) {}

*int* main() {

f(1, 2);

return 0;

}

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

10. 인자 부족

-> 35번 에러

*void* f(*int* *a*, *int* *b*) {}

*int* main() {

f(1);

return 0;

}

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.  
텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

11. 잘못된 구조체 필드 접근  
-> 37번 에러

*struct* S { *int* \_dummy; };

*int* main() {

*struct* S s;

return s.a;

}

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

12. sizeof 허용되지 않는 타입

-> 39번 에러

*int* f() { return 1; }

*int* main() {

return sizeof(f);

}

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

스크린샷, 폰트, 텍스트, 그래픽이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**12번 경우, 왜 GCC가 돌아갔는지?**

함수 이름 f는 sizeof 안에 들어가면 암시적으로 함수 포인터로 변환되지 않고, 오류 없이 함수 자체의 타입으로 계산된다. 즉, C 컴파일러에서는 이를 허용하고 보통 sizeof(void (\*)) 와 동일하게 처리된다. 실제로 gcc는 sizeof의 피연산자가 함수 타입인 경우를 허용하고 sizeof(f)의 결과로 포인터 크기를 리턴해준다.

이 과제 속 시멘틱 분석기는, 함수 이름은 객체가 아니므로 sizeof의 피연사자가 될 수 없다는 입장을 고수한다. 하지만 현재 C 표준 및 대부분의 최신 컴파일러는 허용해준다. 그렇기에 GCC에서는 돌아간다.

13. 관계 연산자에 구조체

-> 40번 에러

*struct* S { *int* x; };

*int* main() {

*struct* S s1, s2;

return s1 > s2;

}

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

14. for 중간 표현식이 scalar 아님

-> 49번 에러

*struct* S { *int* x; };

*int* main() {

*struct* S s;

for (; s; );

return 0;

}

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

15. switch 조건이 정수 아님

-> 50번 에러

*int* main() {

*float* f = 3.0;

switch (f) { }

return 0;

}

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

16. case에 정수 아님

-> 51번 에러

*int* main() {

switch (1) {

case 3.14:

break;

}

return 0;

}

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

17. 반환 타입이 구조체 등 허용하지 않음

-> 57번 에러

*struct* S { *int* x; };

*struct* S f() {

return 0;

}

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

21. 허용되지 않는 캐스팅

-> 58번 에러

*struct* S { *int* x; };

*int* main() {

*struct* S s;

return (*int*)s;

}

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

18. 인자 타입 부적합

-> 59번 에러

*void* f(*int* *x*) {}

*int* main() {

*struct* S { *int* a; } s;

f(s);

return 0;

}

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

19. lvalue 아님

-> 60번 에러

*int* main() {

1 = 2;

return 0;

}

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

20. switch 밖 case

-> 71번 에러

*int* main() {

case 1: break;

return 0;

}

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

21. switch 밖 default

-> 72번 에러

*int* main() {

default: break;

return 0;

}

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

26. switch/loop 밖 break

-> 73번 에러

*int* main() {

break;

return 0;

}

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

22. loop 밖 continue

-> 74번 에러

*int* main() {

continue;

return 0;

}

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

23. enum 값이 정수 아님

-> 81번 에러

*enum* E { A = "hello" };

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

24. 음수 크기 배열

-> 82번 에러

*int* main() {

*int* a[-3];

return 0;

}

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

25. void 배열

-> 83번 에러

*int* main() {

*void* a[10];

return 0;

}

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

26. struct 필드에 void

-> 84번 에러

*struct* S {

*void* a;

};

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

27. 함수 반환 타입 오류

-> 85번 에러

*int* f()[];

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

28. 배열 크기 생략

-> 86번 에러

*int* main() {

*int* a[];

return 0;

}

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

29. 구조체 자기 참조

-> 94번 에러

*struct* s { *int* a; *struct* x{*int* y; *struct* s g;} b; *int* c; } d;

*int* main(){

return 0;

}

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

# Trouble Shooting

**1. 간단한 타이핑 오류**

처음 함수를 선언했을 때 파라미터와 실제 함수를 정의했을 때 파라미터를 실수로 잘못 타이핑 하여 오류가 발생했다.

그 외 큰 오류는 발생하지 않았다.