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.

1. 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.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

* + 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.

1. 테스트

* 올바른 C언어 코드
  + Hello World 출력

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* + - Main 함수에서 printf(“Hello World!\n”)를 수행했다.
    - Syntax tree가 나오고 semantic tree가 이어서 출력된다.
  + int \*fun() 함수

텍스트, 디스플레이, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* + - 파라미터로 int a를 넘겨받고 지역 변수로 int x를 선언하는 int \*fun()함수를 선언한다.
  + Enum 선언 테스트

텍스트, 전자제품, 스크린샷, 디스플레이이(가) 표시된 사진

자동 생성된 설명

* + - 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

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* + - * 오류가 검출되지 않고 시멘틱 검사가 진행됐다.
    - gcc

텍스트, 스크린샷, 폰트이(가) 표시된 사진

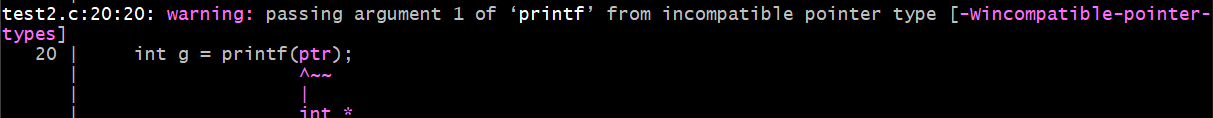
자동 생성된 설명

* + - * unary expression에 맞지 않는 type이라는 error 메시지를 출력했다.
  + 21: illegal type in function call expression
    - a.exe

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* + - * function call이 검출됐지만, 에러 메시지를 보내지 않았다.
    - gcc

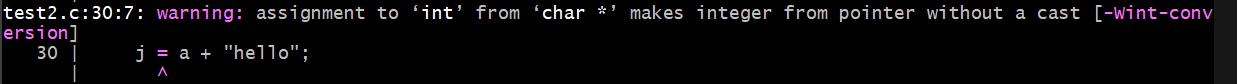


* + - * printf()함수의 호환되지 않는 포인터 타입 에러 메시지를 출력했다.
  + 28: arithmetic type expression required in binary operation
    - a.exe

텍스트, 전자제품, 스크린샷, 디스플레이이(가) 표시된 사진

자동 생성된 설명

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



* + - * 에러가 아닌 경고가 발생했다.
  + 29: integral type expression required in array subscript or binary operation
    - a.exe

텍스트, 디스플레이, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* + - * arr변수까지 검출하다가 segmentation fault가 발생했다.
    - gcc

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

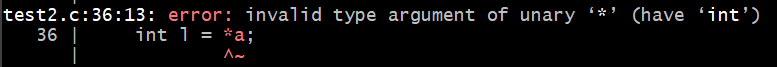
자동 생성된 설명

* + - * 배열의 인덱스로 정수형만 올 수 있다는 에러 메시지를 출력했다.
  + 29: pointer type expression required in pointer operation
    - a.exe

텍스트, 전자제품, 스크린샷, 디스플레이이(가) 표시된 사진

자동 생성된 설명

* + - gcc



* + - * unary expression \*에 대한 타입이 맞지 않다는 에러 메시지를 출력하고 있다.
  + 90: fatal compiler error in parse result
    - a.exe

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

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

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

* + - * 똑같이 52번째 줄에서 에러가 발생함을 보이고 있다.

1. 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);

}