1. 구문분석기 프로그램을 설치하고 여러 가지 C 프로그램을 입력으로 실험하시오.

1-1. 소스코드

[main.c]

```
#include <stdio.h>
#include <stdlib.h>
#include "type.h"
extern FILE *yyin;
extern int syntax_err;
extern A_NODE *root;
FILE *fout;
void initialize();
void print_ast();
extern FILE *yyin;

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("source file not given\n");
        exit(1);
    }

    if ((yyin = fopen(argv[argc-1],"r")) == NULL) {
        printf("can not open input file: %s\n",argv[argc-1]);
        exit(1);
    }

    printf("\nstart syntax analysis\n"); initialize();
    yyparse();
    if (syntax_err)
        exit(1);
    print_ast(root);
    exit(0);
    return 0;
}
```

[print.c]

```
#include <stdio.h>
#include "type.h"
char * node_name[] = {
"N_NULL",
"N_PROGRAM", "N_EXP_IDENT", "N_EXP_INT_CONST", "N_EXP_FLOAT_CONST",
"N_EXP_CHAR_CONST",
"N_EXP_STRING_LITERAL", "N_EXP_ARRAY", "N_EXP_FUNCTION_CALL",
"N_EXP_STRUCT", "N_EXP_ARROW",
"N_EXP_POST_INC", "N_EXP_POST_DEC", "N_EXP_PRE_INC", "N_EXP_PRE_DEC",
"N_EXP_AMP", "N_EXP_STAR",
"N_EXP_NOT", "N_EXP_PLUS", "N_EXP_MINUS", "N_EXP_SIZE_EXP",
"N_EXP_SIZE_TYPE", "N_EXP_CAST",
"N_EXP_MUL", "N_EXP_DIV", "N_EXP_MOD", "N_EXP_ADD", "N_EXP_SUB",
"N_EXP_LSS", "N_EXP_GTR",
"N_EXP_LEQ", "N_EXP_GEQ", "N_EXP_NEQ", "N_EXP_EQL", "N_EXP_AND",
"N_EXP_OR", "N_EXP_ASSIGN",
"N_ARG_LIST",
"N_ARG_LIST_NIL",
"N_STMT_LABEL_CASE", "N_STMT_LABEL_DEFAULT", "N_STMT_COMPOUND",
"N_STMT_EMPTY",
"N_STMT_EXPRESSION", "N_STMT_IF", "N_STMT_IF_ELSE", "N_STMT_SWITCH",
"N_STMT_WHILE",
"N_STMT_DO", "N_STMT_FOR", "N_STMT_RETURN", "N_STMT_CONTINUE",
"N_STMT_BREAK",
"N_FOR_EXP", "N_STMT_LIST", "N_STMT_LIST_NIL",
"N_INIT_LIST", "N_INIT_LIST_ONE", "N_INIT_LIST_NIL"
};
void print_ast(A_NODE *);
void prt_program(A_NODE *, int);
void prt_initializer(A_NODE *, int);
void prt_arg_expr_list(A_NODE *, int);
void prt_statement(A_NODE *, int);
```

```
void prt_statement_list(A_NODE *, int);
void prt_for_expression(A_NODE *, int);
void prt_expression(A_NODE *, int);
void prt_A_TYPE(A_TYPE *, int);
void prt_A_ID_LIST(A_ID *, int);
void prt_A_ID(A_ID *, int);
void prt_A_ID_NAME(A_ID *, int);
void prt_STRING(char *, int);
void prt_integer(int, int);
void print_node(A_NODE *,int);
void print_space(int);
extern A_TYPE *int_type, *float_type, *char_type, *void_type, *string_type;
void print_node(A_NODE *node, int s) {
    print_space(s);
    printf("%s\n", node_name[node->name]);
}
void print_space(int s) {
    int i;
    for(i=1; i<=s; i++) printf("| ");
}
void print_ast(A_NODE *node) {
    printf("======= syntax tree ==========\n");
    prt_program(node,0);
}
void prt_program(A_NODE *node, int s) {
    print_node(node,s);
    switch(node->name) {
        case N_PROGRAM:
            prt_A_ID_LIST(node->clink, s+1);
            break;
        default :
            printf("****syntax tree error******");
    }
}
```

```
void prt_initializer(A_NODE *node, int s) {
    if(!node)
        printf("No more node at s==%d\n", s);
        return;

    print_node(node,s);
    switch(node->name) {
    case N_INIT_LIST:
        prt_initializer(node->llink, s+1);
        prt_initializer(node->rlink, s+1);
        break;
    case N_INIT_LIST_ONE:
        prt_expression(node->clink, s+1);
        break;
    case N_INIT_LIST_NIL:
        break;
    default :
        printf("****syntax tree error******");
    }
}
void prt_expression(A_NODE *node, int s) {
    print_node(node,s);
    switch(node->name)
    {
        case N_EXP_IDENT : prt_A_ID_NAME(node->clink, s+1); break;
        case N_EXP_INT_CONST : prt_integer(node->clink, s+1); break;
        case N_EXP_FLOAT_CONST : prt_STRING(node->clink, s+1); break;
        case N_EXP_CHAR_CONST : prt_integer(node->clink, s+1); break;
        case N_EXP_STRING_LITERAL : prt_STRING(node->clink, s+1); break;
        case N_EXP_ARRAY : prt_expression(node->llink, s+1);
prt_expression(node->rlink, s+1); break;
        case N_EXP_FUNCTION_CALL : prt_expression(node->llink, s+1);
prt_arg_expr_list(node->rlink, s+1); break;
```

```
        case N_EXP_STRUCT :
        case N_EXP_ARROW : prt_expression(node->llink, s+1);
prt_STRING(node->rlink, s+1); break;
        case N_EXP_POST_INC :
        case N_EXP_POST_DEC :
        case N_EXP_PRE_INC :
        case N_EXP_PRE_DEC :
        case N_EXP_AMP :
        case N_EXP_STAR :
        case N_EXP_NOT :
        case N_EXP_PLUS :
        case N_EXP_MINUS :
        case N_EXP_SIZE_EXP : prt_expression(node->clink, s+1); break;
        case N_EXP_SIZE_TYPE : prt_A_TYPE(node->clink, s+1); break;
        case N_EXP_CAST : prt_A_TYPE(node->llink, s+1);
prt_expression(node->rlink, s+1); break;
        case N_EXP_MUL :
        case N_EXP_DIV :
        case N_EXP_MOD :
        case N_EXP_ADD :
        case N_EXP_SUB :
        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 : prt_expression(node->llink, s+1);
prt_expression(node->rlink, s+1); break;
        default :
            printf("****syntax tree error******");
    }
```

```
}
void prt_arg_expr_list(A_NODE *node, int s) {
    print_node(node,s);
    switch(node->name) {
        case N_ARG_LIST : prt_expression(node->llink, s+1);
prt_arg_expr_list(node->rlink, s+1); break;
        case N_ARG_LIST_NIL : break;
        default :
            printf("****syntax tree error******");
    }
}
void prt_statement(A_NODE *node, int s) {
print_node(node,s);
switch(node->name) {
    case N_STMT_LABEL_CASE : prt_expression(node->llink, s+1);
prt_statement(node->rlink, s+1); break;
    case N_STMT_LABEL_DEFAULT : prt_statement(node->clink, s+1); break;
    case N_STMT_COMPOUND: if(node->llink) prt_A_ID_LIST(node->llink, s+1);
prt_statement_list(node->rlink, s+1); break;
    case N_STMT_EMPTY: break;
    case N_STMT_EXPRESSION: prt_expression(node->clink, s+1); break;
    case N_STMT_IF_ELSE: prt_expression(node->llink, s+1);
prt_statement(node->clink, s+1); prt_statement(node->rlink, s+1); break;
    case N_STMT_IF:
    case N_STMT_SWITCH: prt_expression(node->llink, s+1);
prt_statement(node->rlink, s+1); break;
    case N_STMT_WHILE: prt_expression(node->llink, s+1);
prt_statement(node->rlink, s+1); break;
    case N_STMT_DO: prt_statement(node->llink, s+1);
prt_expression(node->rlink, s+1); break;
    case N_STMT_FOR: prt_for_expression(node->llink, s+1);
prt_statement(node->rlink, s+1); break;
    case N_STMT_CONTINUE: break;
    case N_STMT_BREAK: break;
```

```
    case N_STMT_RETURN: if(node->clink) prt_expression(node->clink, s+1);
break;
    default : printf("****syntax tree error******");
    }
}
void prt_statement_list(A_NODE *node, int s) {
print_node(node,s);
switch(node->name) {
    case N_STMT_LIST: prt_statement(node->llink, s+1);
prt_statement_list(node->rlink, s+1); break;
    case N_STMT_LIST_NIL: break;
    default :
    printf("****syntax tree error******");
    }
}
void prt_for_expression(A_NODE *node, int s) {
    print_node(node,s);
    switch(node->name) {
    case N_FOR_EXP :
        if(node->llink) prt_expression(node->llink, s+1);
        if(node->clink) prt_expression(node->clink, s+1);
        if(node->rlink) prt_expression(node->rlink, s+1);
        break;
    default :
        printf("****syntax tree error******");
    }
}
void prt_integer(int a, int s) {
    print_space(s);
    printf("%d\n", a);
}
void prt_STRING(char *str, int s) {
    print_space(s);
    printf("%s\n", str);
```

```
}

char
*type_kind_name[]={"NULL","ENUM","ARRAY","STRUCT","UNION","FUNC","POINTER","V
OI D"};

void prt_A_TYPE(A_TYPE *t, int s) {
    print_space(s);
    if (t==int_type)
        printf("(int)\n");
    else if (t==float_type)
        printf("(float)\n");
    else if (t==char_type)
        printf("(char %d)\n",t->size);
    else if (t==void_type)
        printf("(void)");
    else if (t->kind==T_NULL)
        printf("(null)");
    else if (t->prt)
        printf("(DONE:%x)\n",t);
    else
        switch (t->kind) {
            case T_ENUM: t->prt=TRUE;
                printf("ENUM\n");
                print_space(s); printf("| ENUMERATORS\n");
prt_A_ID_LIST(t->field,s+2);
                break;
            case T_POINTER: t->prt=TRUE;
                printf("POINTER\n");
                print_space(s); printf("| ELEMENT_TYPE\n");
prt_A_TYPE(t->element_type,s+2);
                break;
            case T_ARRAY: t->prt=TRUE;
                printf("ARRAY\n");
```

```
                print_space(s); printf("| INDEX\n"); if (t->expr)
                prt_expression(t->expr,s+2); else {
                print_space(s+2); printf("(none)\n");} print_space(s); printf("|
ELEMENT_TYPE\n"); prt_A_TYPE(t->element_type,s+2);
                break;
            case T_STRUCT: t->prt=TRUE;
                printf("STRUCT\n");
                print_space(s); printf("| FIELD\n"); prt_A_ID_LIST(t->field,s+2);
                break;
            case T_UNION: t->prt=TRUE;
                printf("UNION\n");
                print_space(s); printf("| FIELD\n"); prt_A_ID_LIST(t->field,s+2);
                break;
            case T_FUNC: t->prt=TRUE;
                printf("FUNCTION\n");
                print_space(s);
                printf("| PARAMETER\n");
                prt_A_ID_LIST(t->field,s+2);
                print_space(s);
                printf("| TYPE\n");
                prt_A_TYPE(t->element_type,s+2);
                if (t->expr) {
                    print_space(s);
                    printf("| BODY\n");
                    prt_statement(t->expr,s+2);
                }
            }
}

void prt_A_ID_LIST(A_ID *id, int s) {
    while (id) { prt_A_ID(id,s);
        id=id->link;
    }
}
```

```
char
*id_kind_name[]={"NULL","VAR","FUNC","PARM","FIELD","TYPE","ENUM","STRUCT","EN
UM _LITERAL"};

char *spec_name[]={"NULL","AUTO","TYPEDEF","STATIC"};

void prt_A_ID_NAME(A_ID *id, int s) {
    print_space(s);
    printf("(ID=\"%s\") TYPE:%x KIND:%s SPEC=%s LEV=%d \n", id->name,
id->type,
    id_kind_name[id->kind], spec_name[id->specifier],id->level);
}

void prt_A_ID(A_ID *id, int s) {
    print_space(s);
    printf("(ID=\"%s\") TYPE:%x KIND:%s SPEC=%s LEV=%d\n", id->name,
id->type,
    id_kind_name[id->kind], spec_name[id->specifier],id->level);
    if (id->type) {
        print_space(s);
        printf("| TYPE\n");
        prt_A_TYPE(id->type,s+2);
    }
    if (id->init)
    {
        print_space(s);
        printf("| INIT\n");
        if (id->kind==ID_ENUM_LITERAL)
            prt_expression(id->init,s+2);
    }
    else
    prt_initializer(id->init,s+2);
}
```

[parse.y]

```
%{
    #define YYSTYPE_IS_DECLARED 1
    typedef long YYSTYPE;
    #include "type.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;
%}

%start      program
%token      IDENTIFIER TYPE_IDENTIFIER
    FLOAT_CONSTANT INTEGER_CONSTANT CHARACTER_CONSTANT
STRING_LITERAL
    PLUS MINUS PLUSPLUS MINUSMINUS BAR AMP BARBAR AMPAMP ARROW
    SEMICOLON LSS GTR LEQ GEQ EQL NEQ DOTDOTDOT
    LP RP LB RB LR RR PERIOD COMMA EXCL STAR SLASH PERCENT ASSIGN
COLON
    AUTO_SYM STATIC_SYM TYPEDEF_SYM
    STRUCT_SYM ENUM_SYM SIZEOF_SYM UNION_SYM
    IF_SYM ELSE_SYM WHILE_SYM DO_SYM FOR_SYM CONTINUE_SYM
BREAK_SYM RETURN_SYM
    SWITCH_SYM CASE_SYM DEFAULT_SYM

%%

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_list_opt
    :                                           {$$ = NIL;}
    | declaration_list                          {$$ = $1;}
    ;
declaration_list
    : declaration                               {$$ = $1;}
    | declaration_list declaration          {$$ = linkDeclaratorList($1, $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                              {$$ = S_TYPEDEF;}
    ;
init_declarator_list_opt
    :                                          {$$ = 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_type_specifier                         {$$ = $1;}
    | enum_type_specifier                            {$$ = $1;}
    | TYPE_IDENTIFIER                                 {$$ = $1;}
    ;
struct_type_specifier
    : struct_or_union IDENTIFIER                      {$$ =
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                                {$$ = $1;}
    ;
enum_type_specifier
    : ENUM_SYM IDENTIFIER                           {$$ =
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
    | 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_opt
    :                                               {$$ = NIL;}
    | parameter_type_list                           {$$ = $1;}
    ;
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

{$$=setParameterDeclaratorSpecifier(setDeclaratorType(makeDummyIdentifier(),$2),
$1) ;}
    ;
abstract_declarator_opt
    :                                       {$$ = NIL;}
    | abstract_declarator                   {$$ = $1;}
    ;
abstract_declarator
    : direct_abstract_declarator            {$$ = $1;}
    | pointer                               {$$=makeType(T_POINTER);}
    | pointer direct_abstract_declarator
{$$=setTypeElementType($2,makeType(T_POINTER));}
    ;
direct_abstract_declarator
    : LP abstract_declarator RP              {$$=$2;}
    | LB constant_expression_opt RB
{$$=setTypeExpr(makeType(T_ARRAY),$2);}
    | direct_abstract_declarator LB constant_expression_opt RB

{$$=setTypeElementType($1,setTypeExpr(makeType(T_ARRAY),$3));}
    | LP parameter_type_list_opt RP
{$$=setTypeExpr(makeType(T_FUNC),$2);}
    | direct_abstract_declarator LP parameter_type_list_opt RP

{$$=setTypeElementType($1,setTypeExpr(makeType(T_FUNC),$3));}
    ;
statement_list_opt
    :
```

```
{$$=makeNode(N_STMT_LIST_NIL,NIL,NIL,NIL);}
    | statement_list                                {$$=$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);}

    ;
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_i
d=$2;current_level--;}
    ;
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
{$$=makeNode(N_STMT_BREAK,NIL,NIL,NIL);}
    ;
arg_expression_list_opt
    :
{$$=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

{$$=makeNodeList(N_ARG_LIST,$1,$3);}
    ;
constant_expression_opt
    :                                        {$$=NIL;}
    | constant_expression                     {$$=$1;}
    ;
constant_expression
    : expression                             {$$=$1;}
    ;
expression
    : comma_expression                         {$$=$1;}
    ;
comma_expression
    : assignment_expression                   {$$=$1;}
    ;
```

```
assignment_expression
    : conditional_expression                    {$$=$1;}
    | unary_expression ASSIGN assignment_expression

{$$=makeNode(N_EXP_ASSIGN,$1,NIL,$3);}
    ;
conditional_expression
    : logical_or_expression                     {$$=$1;}
    ;
logical_or_expression
    : logical_and_expression                    {$$=$1;}
    | logical_or_expression BARBAR logical_and_expression

{$$=makeNode(N_EXP_OR,$1,NIL,$3);}
    ;
logical_and_expression
    : bitwise_or_expression                     {$$=$1;}
    | logical_and_expression AMPAMP bitwise_or_expression

{$$=makeNode(N_EXP_AND,$1,NIL,$3);}
    ;
bitwise_or_expression
    : bitwise_xor_expression                    {$$=$1;}
    ;
bitwise_xor_expression
    : bitwise_and_expression                    {$$=$1;}
    ;
bitwise_and_expression
    : equality_expression                       {$$=$1;}
    ;
equality_expression
    : relational_expression                     {$$=$1;}
    | equality_expression EQL relational_expression
```

```
{$$=makeNode(N_EXP_EQL,$1,NIL,$3);}
    | equality_expression NEQ relational_expression

{$$=makeNode(N_EXP_NEQ,$1,NIL,$3);}
    ;
relational_expression
    : shift_expression                              {$$=$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

{$$=makeNode(N_EXP_GEQ,$1,NIL,$3);}
    ;
shift_expression
    : additive_expression                           {$$=$1;}
    ;
additive_expression
    : multiplicative_expression                     {$$=$1;}
    | additive_expression PLUS multiplicative_expression

{$$=makeNode(N_EXP_ADD,$1,NIL,$3);}
    | additive_expression MINUS multiplicative_expression

{$$=makeNode(N_EXP_SUB,$1,NIL,$3);} ;
multiplicative_expression
    : cast_expression                               {$$=$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

{$$=makeNode(N_EXP_MOD,$1,NIL,$3);}
    ;
cast_expression
    : unary_expression                          {$$=$1;}
    | LP type_name RP cast_expression
{$$=makeNode(N_EXP_CAST,$2,NIL,$4);}
    ;
unary_expression
    : postfix_expression                        {$$=$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);}
```

```
    ;
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
{$$=makeNode(N_EXP_POST_DEC,NIL,$1,NIL);}
    ;
primary_expression
    : IDENTIFIER
{$$=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                                  {$$=$2;}
    ;
type_name
    : declaration_specifiers abstract_declarator_opt
                                                        {$$=setTypeNameSpecifier($2,
$1);}
```

```
    ;

%%
extern char *yytext;

yyerror(char *s)
{
    syntax_err++;
    printf("line %d: %s near %s \n", line_no, s,yytext);
}
```

[syntax.c]

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "type.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 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 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 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 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;
}

// specifier merge & update
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);
    else
        p->type=t;

    if (s) {
        if (p->stor)
            if(s==p->stor)
                ;
            else
                syntax_error(24);
        else
            p->stor=s;
        }
    return (p);
}

// link two declaraor 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 id
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)
{
    // ** to be completed ** -> completed
    A_TYPE *tt;
    if(id->type == NIL)      // if id has no type, then assign that by argument
immediately
        id->type = t;
    else                      // else type will be added into the tail of the id
    {
        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);
    setDefaultSpecifier(p);
// check check if there is a function identifier immediately before '('
// ** to be completed ** -> completed, (and this was the midterm ex. t_t)
    if(id->type->kind != T_FUNC) {
        syntax_error(21);
        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
```

```
            // ** to be completed ** -> completed
            if(isNotSameFormalParameters(a->type->field, id->type->field))
                // check all the field elements about types and different
parmeter numbers
                syntax_error(22, a->name);
            if(isNotSameType(a->type->element_type, id->type->element_type))
                // return type is located in 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);
        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);
            // ** to be completed ** -> completed
            a = setDeclaratorElementType(a, p->type);
```

```
            if(p->stor == S_TYPEDEF)
                a->kind = ID_TYPE;
            else if(a->type->kind == T_FUNC)
                a->kind = ID_TYPE;
            else
                a->kind = ID_VAR;
            a->specifier = p->stor;      // id's specifier should be storage class
of A_SPECIFIER's 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 papameter storage class && void type
    if (p->stor || p->type==void_type)
        syntax_error(14);
    // ** to be completed ** -> completed
    // check parameter storage class && void type
    if(p->stor || p->type == void_type)
        syntax_error(14);
    setDefaultSpecifier(p);
    id = setDeclaratorElementType(id, p->type);
    id->kind = ID_PARM;
    return(id);
}

A_ID *setStructDeclaratorListSpecifier(A_ID *id, A_TYPE *t) {
    A_ID *a;
```

```
    a=id;
    while (a) {
        // ** to be completed ** -> completed
        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);
    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 iIdentifier
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 kinf 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 protype
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) {
    return (t && (t->kind == T_POINTER || t->kind == T_ARRAY));
}
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; float_type->size=4; char_type->size=1;
    int_type->check=TRUE; float_type->check=TRUE; 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_PARM
                ),
                setDeclaratorKind(
                    makeDummyIdentifier(),
                    ID_PARM
                )
            )
        ),
        ID_FUNC
    );
    // scanf(char *, ...) library function
    setDeclaratorTypeAndKind(
        makeIdentifier("scanf"),
```

```
        setTypeField(
            setTypeElementType(
                makeType(T_FUNC),
                void_type
            ),
            linkDeclaratorList(
                setDeclaratorTypeAndKind(
                    makeDummyIdentifier(),
                    string_type,
                    ID_PARM
                ),
                setDeclaratorKind(
                    makeDummyIdentifier(),
                    ID_PARM
                )
            )
        ),
        ID_FUNC
);
// malloc(int) library function
setDeclaratorTypeAndKind(
        makeIdentifier("malloc"),
        setTypeField(
            setTypeElementType(
                makeType(T_FUNC),
                string_type
            ),
            setDeclaratorTypeAndKind(
                makeDummyIdentifier(),
                int_type,
                ID_PARM
            )
        ),
        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 parameter 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 prototype 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);
}
```

[scan.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(TYPEDEF_SYM); }
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(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);
}
"//"[^\n]* { }

%%

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);
    }
}
```

```
yywrap()
{
    return(1);
    // 1 means done, 0 means need more implementation
    // WARNING :: 0 causes infinite loop when token exausted.
}
```

[type.h]

```
#define NIL 0
typedef enum {FALSE,TRUE} BOOLEAN;

typedef enum e_node_name
{
    N_NULL,
    N_PROGRAM,
    N_EXP_IDENT,
    N_EXP_INT_CONST,
    N_EXP_FLOAT_CONST,
    N_EXP_CHAR_CONST,
    N_EXP_STRING_LITERAL,
    N_EXP_ARRAY,
    N_EXP_FUNCTION_CALL,
    N_EXP_STRUCT,
    N_EXP_ARROW,
    N_EXP_POST_INC,
    N_EXP_POST_DEC,
    N_EXP_PRE_INC,
    N_EXP_PRE_DEC,
    N_EXP_AMP,
    N_EXP_STAR,
    N_EXP_NOT,
    N_EXP_PLUS,
    N_EXP_MINUS,
    N_EXP_SIZE_EXP,
    N_EXP_SIZE_TYPE,
    N_EXP_CAST,
    N_EXP_MUL,
    N_EXP_DIV,
    N_EXP_MOD,
    N_EXP_ADD,
```

```
    N_EXP_SUB,
    N_EXP_LSS,
    N_EXP_GTR,
    N_EXP_LEQ,
    N_EXP_GEQ,
    N_EXP_NEQ,
    N_EXP_EQL,
    N_EXP_AND,
    N_EXP_OR,
    N_EXP_ASSIGN,
    N_ARG_LIST,
    N_ARG_LIST_NIL,
    N_STMT_LABEL_CASE,
    N_STMT_LABEL_DEFAULT,
    N_STMT_COMPOUND,
    N_STMT_EMPTY,
    N_STMT_EXPRESSION,
    N_STMT_IF,
    N_STMT_IF_ELSE,
    N_STMT_SWITCH,
    N_STMT_WHILE,
    N_STMT_DO,
    N_STMT_FOR,
    N_STMT_RETURN,
    N_STMT_CONTINUE,
    N_STMT_BREAK,
    N_FOR_EXP,
    N_STMT_LIST,
    N_STMT_LIST_NIL,
    N_INIT_LIST,
    N_INIT_LIST_ONE,
    N_INIT_LIST_NIL
} NODE_NAME;
```

```
typedef enum
{
    T_NULL,T_ENUM,T_ARRAY,T_STRUCT,T_UNION,T_FUNC,T_POINTER,T_VOID
} T_KIND;

typedef enum
{
    Q_NULL,Q_CONST,Q_VOLATILE
} Q_KIND;

typedef enum
{
    S_NULL,S_AUTO,S_STATIC,S_TYPEDEF,S_EXTERN,S_REGISTER
} S_KIND;
typedef enum
{
    ID_NULL,ID_VAR,ID_FUNC,ID_PARM,ID_FIELD,ID_TYPE,ID_ENUM,
    ID_STRUCT,ID_ENUM_LITERAL
} ID_KIND;

typedef struct s_node
{
    NODE_NAME name;
    int line;
    int value;
    struct s_type *type;
    struct s_node *llink;
    struct s_node *clink;
    struct s_node *rlink;
} A_NODE;

typedef struct s_type
{
    T_KIND kind;
```

```
    int size;
    int local_var_size;
    struct s_type *element_type; struct s_id *field;
    struct s_node *expr;
    int line;
    BOOLEAN check;
    BOOLEAN prt;
} A_TYPE;

typedef struct s_id
{
    char *name;
    ID_KIND kind;
    S_KIND specifier;
    int level;
    int address;
    int value;
    A_NODE *init;
    A_TYPE *type;
    int line;
    struct s_id *prev; struct s_id *link;
} A_ID;

typedef union
{
    int i; float f; char c; char *s;
} LIT_VALUE;

typedef struct lit
{
    int addr; A_TYPE *type; LIT_VALUE value;
} A_LITERAL;

typedef struct
```

```
{
    A_TYPE *type;
    S_KIND stor;
    int line;
} A_SPECIFIER;
```

2-2. 결과

```
student@ubuntu:~/compiler/hw6$ ./a.out
3+4*5
+
    left : 3
        No Node    No Node
    right : *
    left : 4
        No Node    No Node
    right : 5
        No Node    No Node
student@ubuntu:~/compiler/hw6$
```