

A4 Supporting Functions for Syntax Analysis: syntax.c

```
#include <stdio.h>
#include <stdlib.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=NULL;
int syntax_err=0;
int line_no=1;
int current_level=0;
A_NODE *makeNode (NODE_NAME,A_NODE *,A_NODE *,A_NODE *);
A_NODE *makeNodeList (NODE_NAME,A_NODE *,A_NODE *);
A_ID *makeIdentifier(char *);
A_ID *makeDummyIdentifier();
A_TYPE *makeType(T_KIND);
A_SPECIFIER *makeSpecifier(A_TYPE *,S_KIND);
A_ID *searchIdentifier(char *,A_ID *);
A_ID *searchIdentifierAtCurrentLevel(char *,A_ID *);
A_SPECIFIER *updateSpecifier(A_SPECIFIER *, A_TYPE *, S_KIND);
void checkForwardReference();
void setDefaultSpecifier(A_SPECIFIER *);
A_ID *linkDeclaratorList(A_ID *,A_ID *) ;
A_ID *getIdentifierDeclared(char *);
A_TYPE *getTypeOfStructOrEnumRefIdentifier(T_KIND,char *,ID_KIND);
A_ID *setDeclaratorInit(A_ID *,A_NODE *);
A_ID *setDeclaratorKind(A_ID *,ID_KIND);
A_ID *setDeclaratorType(A_ID *,A_TYPE *);
A_ID *setDeclaratorElementType(A_ID *,A_TYPE *);
A_ID *setDeclaratorTypeAndKind(A_ID *,A_TYPE *,ID_KIND);
A_ID *setDeclaratorListSpecifier(A_ID *,A_SPECIFIER *);
A_ID *setFunctionDeclaratorSpecifier(A_ID *, A_SPECIFIER *);
```

```

A_ID    *setFunctionDeclaratorBody(A_ID *, A_NODE *);
A_ID    *setParameterDeclaratorSpecifier(A_ID *, A_SPECIFIER *);
A_ID    *setStructDeclaratorListSpecifier(A_ID *, A_TYPE *);
A_TYPE  *setTypeNamespecifier(A_TYPE *, A_SPECIFIER *);
A_TYPE  *setTypeElementType(A_TYPE *,A_TYPE *);
A_TYPE  *setTypeField(A_TYPE *,A_ID *);
A_TYPE  *setTypeExpr(A_TYPE *,A_NODE *);
A_TYPE  *setTypeAndKindOfDeclarator(A_TYPE *,ID_KIND,A_ID *);
A_TYPE  *setTypeStructOrEnumIdentifier(T_KIND,char *,ID_KIND);
BOOLEAN isNotSameFormalParameters(A_ID *, A_ID *);
BOOLEAN isNotSameType(A_TYPE *, A_TYPE *);
BOOLEAN isPointerOrArrayType(A_TYPE *);
void syntax_error();
void initialize();
// make new node for syntax tree
A_NODE *makeNode (NODE_NAME n, A_NODE *a, A_NODE *b, A_NODE *c) {
    A_NODE *m;
    m = (A_NODE*)malloc(sizeof(A_NODE));
    m->name=n;
    m->llink=a;
    m->clink=b;
    m->rlink=c;
    m->type=NIL;
    m->line=line_no;
    m->value=0;
    return (m);
}
A_NODE *makeNodeList (NODE_NAME n, A_NODE *a, A_NODE *b) {
    A_NODE *m,*k;
    k=a;
    while (k->rlink)
        k=k->rlink;
    m = (A_NODE*)malloc(sizeof(A_NODE));
    m->name=k->name;

```

```

        m->llink=NIL;
        m->clink=NIL;
        m->rlink=NIL;
        m->type=NIL;
        m->line=line_no;
        m->value=0;
        k->name=n;
        k->llink=b;
        k->rlink=m;
        return(a);
    }
    // make a new declarator for identifier
    A_ID *makeIdentifier(char *s) {
        A_ID *id;
        id = malloc(sizeof(A_ID));
        id->name = s;
        id->kind = 0;
        id->specifier = 0;
        id->level = current_level;
        id->address = 0;
        id->init = NIL;
        id->type = NIL;
        id->link = NIL;
        id->line = line_no;
        id->value=0;
        id->prev = current_id;
        current_id=id;
        return(id);
    }
    // make a new declarator for dummy identifier
    A_ID *makeDummyIdentifier() {
        A_ID *id;
        id = malloc(sizeof(A_ID));
        id->name = "";

```

```

    id->kind = 0;
    id->specifier = 0;
    id->level = current_level;
    id->address = 0;
    id->init = NIL;
    id->type = NIL;
    id->link = NIL;
    id->line = line_no;
    id->value=0;
    id->prev =0;
    return(id);
}
// make a new type
A_TYPE *makeType(T_KIND k) {
    A_TYPE *t;
    t = malloc(sizeof(A_TYPE));
    t->kind = k;
    t->size=0;
    t->local_var_size=0;
    t->element_type = NIL;
    t->field = NIL;
    t->expr = NIL;
    t->check=FALSE;
    t->prt=FALSE;
    t->line=line_no;
    return(t);
}
// make a new specifier
A_SPECIFIER *makeSpecifier(A_TYPE *t,S_KIND s) {
    A_SPECIFIER *p;
    p = malloc(sizeof(A_SPECIFIER));
    p->type = t;
    p->stor=s;
    p->line=line_no;

```

```

        return(p);
    }
    A_ID *searchIdentifier(char *s, A_ID *id) {
        // 명칭 목록 id 에서 명칭 s 를 탐색
        //
        return(id);
    }
    A_ID *searchIdentifierAtCurrentLevel(char *s, A_ID *id) {
        // 명칭 목록 id 에서 현재 level 과 같은 level에 명칭 s 가 있는지를 탐색
        while (id) {
            //
            //
            id=id->prev;
        }
        return(id);
    }
    void checkForwardReference() {
        A_ID *id;
        A_TYPE *t;
        id=current_id;
        while (id) {
            if (id->level<current_level)
                break;
            t=id->type;
            if (id->kind==ID_NULL)
                syntax_error(31,id->name);
            else if ((id->kind==ID_STRUCT || id->kind==ID_ENUM)
                && t->field==NIL)
                syntax_error(32,id->name);
            id=id->prev;
        }
    }
    // set default specifier
    void setDefaultSpecifier(A_SPECIFIER *p) {

```

```

    A_TYPE *t;
    if (p->type==NIL)
        p->type=int_type;
    if (p->stor==S_NULL)
        p->stor=S_AUTO;
}
// merge & update specifier
A_SPECIFIER *updateSpecifier(A_SPECIFIER *p, A_TYPE *t, S_KIND s) {
    if (t)
        if ( p->type)
            if (p->type==t)
                ;
            else
                syntax_error(24);
        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 declarator list id1 & id2
A_ID *linkDeclaratorList(A_ID *id1, A_ID *id2) {
    A_ID * m = id1;
    if (id1==NIL)
        return(id2);
    while(m->link)
        m=m->link;
    m->link=id2;
    return (id1);
}

```

```

}
// check if identifier is already declared in primary expression
A_ID *getIdentifierDeclared(char *s) {
    A_ID *id;
    id=searchIdentifier(s,current_id);
    if(id==NIL)
        syntax_error(13,s);
    return(id);
}
// get type of struct identifier
A_TYPE * getTypeOfStructOrEnumRefIdentifier(T_KIND k,char *s, ID_KIND kk) {
    A_TYPE *t;
    A_ID * id;
    id=searchIdentifier(s,current_id);
    if (id)
        if (id->kind==kk && id->type->kind==k)
            return(id->type);
        else
            syntax_error(11,s);
    // make a new struct (or enum) identifier
    t=makeType(k);
    id=makeIdentifier(s);
    id->kind=kk;
    id->type=t;
    return(t);
}
// set declarator init (expression tree)
A_ID *setDeclaratorInit(A_ID *id, A_NODE *n) {
    id->init=n;
    return(id);
}
// set declarator kind
A_ID *setDeclaratorKind(A_ID *id, ID_KIND k) {
    A_ID *a;

```

```

        a=searchIdentifierAtCurrentLevel(id->name,id->prev);
        if (a)
            syntax_error(12,id->name);
        id->kind=k;
        return(id);
    }
    // set declarator type
    A_ID *setDeclaratorType(A_ID *id, A_TYPE *t) {
        id->type=t;
        return(id);
    }
    // set declarator type (or element type)
    A_ID *setDeclaratorElementType(A_ID *id, A_TYPE *t) {
        A_TYPE *tt;
        // 명칭 목록의 마지막 타입으로 t를 연결
        if (id->type==NIL)
            id->type=t;
        else {
            // ...
        }
        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;
        // storage class 검사
        // specifier 가 생략된 경우 보정
        // check function identifier immediately before '('

```



```

// 함수의 리턴 타입을 완성하고 명칭의 종류를 ID_FUNC 로 지정한다

// 함수명칭으로 중복선언 검사
// 프로토타입이 있는경우 파라미터와 리턴 타입등 검사
// 파라미터를 함수 내에서 사용할수 있게 스코프 조정
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;
    // 중복 선언 검사
    // 명칭의 타입 완성
    // 명칭의 종류로 ID_TYPE, ID_FUNC, 및 ID_VAR 등을 구분
    while (a) {
        //...
        //...
        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) {
    // 중복선언 검사

    // 파라미터의 storage class 와 void type 여부 검사
    // 파라미터의 타입 완성
    // 명칭의 종류 결정

```

```

        return(id);
    }
A_ID *setStructDeclaratorListSpecifier(A_ID *id, A_TYPE *t) {
    A_ID *a;
    a=id;
    while (a) {
        // 구조체 필드 명칭의 중복선언 검사
        // 필드명칭의 타입완성
        // 명칭의 종류 결정
        // ...
        a=a->link; }
    return(id);
}
// set type name specifier
A_TYPE *setTypeSpecifier(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;
    // t 의 마지막 원소의 타입으로 s 타입을 연결
    // ...
    return(t);
}
// set type field
A_TYPE *setTypeField(A_TYPE *t, A_ID *n) {
    t->field=n;
    return(t);
}

```

```

// set type initial value (expression tree)
A_TYPE *setTypeExpr(A_TYPE *t, A_NODE *n) {
    t->expr=n;
    return(t);
}

// set type of struct identifier
A_TYPE *setTypeStructOrEnumIdentifier(T_KIND k, char *s, ID_KIND kk) {
    A_TYPE *t;
    A_ID *id, *a;
    // check redeclaration or forward declaration
    a=searchIdentifierAtCurrentLevel(s,current_id);
    if (a)
        if (a->kind==kk && a->type->kind==k)
            if (a->type->field)
                syntax_error(12,s);
            else
                return(a->type);
        else
            syntax_error(12,s);
    // make a new struct (or enum) identifier
    id=makeIdentifier(s);
    t=makeType(k);
    id->type=t;
    id->kind=kk;
    return(t);
}

// set type and kind of identifier
A_TYPE *setTypeAndKindOfDeclarator(A_TYPE *t, ID_KIND k, A_ID *id) {
    if (searchIdentifierAtCurrentLevel(id->name,id->prev))
        syntax_error(12,id->name);
    id->type=t;
    id->kind=k;
    return(t);
}

```

```

// check function parameters with prototype
BOOLEAN isNotSameFormalParameters(A_ID *a, A_ID *b) {
    if (a==NIL)        // no parameters in prototype
        return(FALSE);
    while(a) {
        if (b==NIL || isNotSameType(a->type,b->type))
            return(TRUE);
        a=a->link;
        b=b->link; }
    if (b)
        return(TRUE);
    else
        return(FALSE);
}

BOOLEAN isNotSameType(A_TYPE *t1, A_TYPE *t2) {
    if (isPointerOrArrayType(t1) || isPointerOrArrayType(t2))
        return (isNotSameType(t1->element_type,t2->element_type));
    else
        return (t1!=t2);
}

void initialize() {
    // primitive data types
    int_type=setTypeAndKindOfDeclarator(
        makeType(T_ENUM),ID_TYPE,makeIdentifier("int"));
    float_type=setTypeAndKindOfDeclarator(
        makeType(T_ENUM),ID_TYPE,makeIdentifier("float"));
    char_type= setTypeAndKindOfDeclarator(
        makeType(T_ENUM),ID_TYPE,makeIdentifier("char"));
    void_type=setTypeAndKindOfDeclarator(
        makeType(T_VOID),ID_TYPE,makeIdentifier("void"));
    string_type=setTypeElementType(makeType(T_POINTER),char_type);
    int_type->size=4;        int_type->check=TRUE;
    float_type->size=4;      float_type->check=TRUE;
    char_type->size=1;       char_type->check=TRUE;
}

```

```

void_type->size=0;      void_type->check=TRUE;
string_type->size=4;    string_type->check=TRUE;
// printf(char *, ...) library function
setDeclaratorTypeAndKind(
    makeIdentifier("printf"),
    setTypeField(
        setTypeElementType(makeType(T_FUNC),void_type),
        linkDeclaratorList(
setDeclaratorTypeAndKind(makeDummyIdentifier(),string_type,ID_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 parm type in prototype function %s",s);
        break;
case 23: printf("empty parameter name"); break;
case 24: printf("illegal declaration specifiers"); break;
case 25: printf("illegal function specifiers"); break;
case 26: printf("illegal or conflicting return type in function %s",s);
        break;
case 31: printf("undefined type for identifier %s",s); break;
case 32: printf("incomplete forward reference for identifier %s",s);
        break;
default: printf("unknown"); break;
}
if (strlen(yytext)==0)
    printf(" at end\n");
else
    printf(" near %s\n", yytext);
}

```