

HW3

C311054 박민서

2025년 5월 16일

1 Fuction

1.1 함수 정의

```
external_declaration
: function_definition
  {funcnt++; /* 함수 정의 카운트 */
  intcnt+=inttmp; charcnt+=chartmp; pocnt+=potmp; arrcnt+=arrtmp;
  inttmp=chartmp=potmp=arrtmp=0;}
| declaration
| preprocessor
;

function_definition
: declaration_specifiers declarator compound_statement
| declarator compound_statement
;
```

함수의 정의는 yacc에서 function.definition이라는 심볼이다. 이 심볼은 external_declaration으로 무조건 reduce되기 때문에 이 때 카운트해준다. 함수의 전방선언은 declaration으로 reduce되기 때문에 카운트되지 않는다.

1.2 함수 사용

```
postfix_expression
: primary_expression
| postfix_expression '[' expression ']'
| postfix_expression '(' ')' {funcnt++;} /*함수 사용*/
| postfix_expression '(' argument_expression_list ')' {funcnt++;} /*함수사용*/
| postfix_expression '.' IDENTIFIER {opcnt++;}
| postfix_expression PTR_OP IDENTIFIER {opcnt++;}
| postfix_expression INC_OP {opcnt++;}
| postfix_expression DEC_OP {opcnt++;}
;
```

함수의 사용은 '함수이름(인자)'의 형태이기 때문에 해당 부분에서 카운트한다.

2 Operation

연산자 토큰이 사용되는 곳에서 카운트하면 된다.

2.1 참조연산자, 후위증감연산자

```
postfix_expression
: primary_expression
| postfix_expression '[' expression ']'
| postfix_expression '(' ')' {funcnt++;}
| postfix_expression '(' argument_expression_list ')' {funcnt++;}
| postfix_expression '.' IDENTIFIER {opcnt++;} /*참조*/
| postfix_expression PTR_OP IDENTIFIER {opcnt++;} /*참조*/
| postfix_expression INC_OP {opcnt++;} /*후위증가*/
| postfix_expression DEC_OP {opcnt++;} /*후위감소*/
;
```

2.2 전위증감연산자

```
unary_expression
: postfix_expression
| INC_OP unary_expression {opcnt++;} /*전위증가*/
| DEC_OP unary_expression {opcnt++;} /*전위감소*/
| unary_operator cast_expression
| SIZEOF unary_expression
| SIZEOF '(' type_name ')'
;
```

2.3 cast연산자

```
cast_expression
: unary_expression
| '(' type_name ')' cast_expression {opcnt++;}
;
```

2.4 산술연산자

```
multiplicative_expression
: cast_expression
| multiplicative_expression '*' cast_expression {opcnt++;}
| multiplicative_expression '/' cast_expression {opcnt++;}
| multiplicative_expression '%' cast_expression {opcnt++;}
;

additive_expression
: multiplicative_expression
| additive_expression '+' multiplicative_expression {opcnt++;}
| additive_expression '-' multiplicative_expression {opcnt++;}
;
```

2.5 비트연산자

```
shift_expression
: additive_expression
| shift_expression LEFT_OP additive_expression {opcnt++;}
| shift_expression RIGHT_OP additive_expression {opcnt++;}
;

and_expression
: equality_expression
| and_expression '&' equality_expression {opcnt++;}
;

exclusive_or_expression
: and_expression
| exclusive_or_expression '^' and_expression {opcnt++;}
;

inclusive_or_expression
: exclusive_or_expression
| inclusive_or_expression '|' exclusive_or_expression {opcnt++;}
;
```

2.6 논리연산자

```
relational_expression
: shift_expression
| relational_expression '<' shift_expression {opcnt++;}
| relational_expression '>' shift_expression {opcnt++;}
| relational_expression LE_OP shift_expression {opcnt++;}
| relational_expression GE_OP shift_expression {opcnt++;}
;

equality_expression
: relational_expression
| equality_expression EQ_OP relational_expression {opcnt++;}
| equality_expression NE_OP relational_expression {opcnt++;}
;
```

2.7 관계연산자

```
logical_and_expression
: inclusive_or_expression
| logical_and_expression AND_OP inclusive_or_expression {opcnt++;}
;

logical_or_expression
: logical_and_expression
| logical_or_expression OR_OP logical_and_expression {opcnt++;}
;
```

2.8 대입연산자

```
assignment_expression
: conditional_expression
| unary_expression assignment_operator assignment_expression {opcnt++;}
;

assignment_operator
: '='
| MUL_ASSIGN
| DIV_ASSIGN
| MOD_ASSIGN
| ADD_ASSIGN
| SUB_ASSIGN
| LEFT_ASSIGN
| RIGHT_ASSIGN
| AND_ASSIGN
| XOR_ASSIGN
| OR_ASSIGN
;
```

모든 대입연산자는 assignment_operator로 reduce되기 때문에 assignment_operator가 reduce되는 곳에서 한번만 세면 된다.

3 Int, Char, Pointer, Array

조건이 많아서 까다로운 부분이였다. 조건은 다음과 같다.

- 변수의 갯수는 포인터가 1개일 때는 카운트, 2개 이상이면 카운트하지 않는다.
- 배열에서 변수도 카운트한다.
- 함수의 파라미터에서도 카운트한다. 단, 전방 선언에서는 중복으로 세지 않도록 해야 한다.
- 함수의 리턴값에서는 세지 않는다.
- 포인터함수에서는 포인터와 함수만 세고 변수는 세지 않는다.

이 조건들을 구분해내기 위해 direct_declarator부분에서 각 종류에 따라 심볼 값을 할당했다.

```
declarator
: pointer direct_declarator      {if($1==1)$=$2+5;else $=$2+10;}
| direct_declarator             {$=$1;}
;

direct_declarator
: IDENTIFIER                    {$=$0;}
| '(' declarator ')'            {if($2>=5)$=$1;else $=$2;}
| direct_declarator '[' constant_expression ']' {$=$2;}
| direct_declarator '[' ']'     {$=$2;}
| direct_declarator '(' parameter_type_list ')' {if($1==1)$=$4;else $=$3;}
| direct_declarator '(' identifier_list ')'     {$=$0;}
;
```

```

    | direct_declarator '(' ')'
    ;
pointer
: '*'
| '*' type_qualifier_list
| '*' pointer
| '*' type_qualifier_list pointer
;

```

{if(\$1==1)\$\$=4;else \$\$=3;}

{\$\$=1;}

{\$\$=1;}

{\$\$=\$2+1;}

{\$\$=\$3+1;}

direct_declarator의 심볼 값에서 0은 식별자, 2는 배열, 3은 함수를 나타낸다. 1과 4는 포인터함수를 처리하기 위한 값인데, 먼저 declarator를 보자. declarator는 pointer의 심볼 값(*의 갯수를 나타내도록 되어있음)을 보고 direct_declarator에 포인터가 1개 붙으면 direct_declarator의 심볼 값에 5를 더하고 포인터가 2개 이상이면 10을 더한다. 포인터가 없으면 direct_declarator의 심볼 값을 그대로 갖는다. direct_declarator의 심볼 값은 0부터 4까지 있기 때문에 5와 10을 더하면 모든 값들이 다른 수로 구분된다.

만약 direct_declarator의 2번째 규칙에서 괄호 안 declarator의 심볼 값이 5이상이라면 포인터가 하나이상 있다는 것이고 이 상태를 심볼 값 1로 나타낸다. 포인터가 없는 경우는 기존 심볼 값을 그대로 사용한다.

direct_declarator의 5번째와 7번째 규칙은 함수를 나타내는데 이 때 규칙안의 direct_declarator부분이 심볼 값 1을 갖는다면 전체 구조는 (* direct_declarator)(파라미터)가 되므로 포인터함수가 된다. 포인터 함수는 심볼 값 4로 나타낸다.

declarator가 가질 수 있는 심볼 값은 다음과 같다.

0	1	2	3	4
ID	(*)	~[]	~()	(*)()
5	6	7	8	9
*(ID)	*(*)	*~[]	*~()	*(*)()
10	11	12	13	14
ID	**(*)	**~[]	**~()	**()()

그림 1: declarator symbol value

declarator가 reduce되는 곳은 init_declarator, struct_declarator, parameter_declaration, function_definition

이렇게 네 개이다.

3.1 parameter_declaration

parameter_declaration을 먼저 보겠다.

```
parameter_declaration
: declaration_specifiers declarator
{
  if($2==0||$2==1||$2==2||$2==5||$2==7)
  {
    if($1==2)chartmp++;if($1==4)inttmp++;
  }
  if($2==1||$2==5||$2==6||$2==7||$2==10||$2==11||$2==12) potmp++;
  if($2==2||$2==7||$2==12) arrtmp++;
}
| declaration_specifiers abstract_declarator
| declaration_specifiers
;
```

그림 1을 보면 변수는 0,1,2,5,7에서 카운트, 포인터는 1,5,6,7,8,10,11,12에서 카운트, 배열은 2,7,12에서 카운트해야한다. 마지막 2열은 함수이므로 카운트에서 제외한다. 변수는 declaration_specifiers의 심볼 값에 따라 int,char 각각 세어준다.

parameter_declaration은 함수의 정의에 들어갈 수도 있고 전방선언에 들어갈 수도 있다. 만약 전방선언이 된다면 파라미터에서 변수, 포인터, 배열 모두 카운트하면 안된다. 따라서 이 가능성을 염두하고 바로 카운트값을 올리는 대신 tmp라는 별개의 변수로 세어놓는다.

translation_unit은 external_declaration의 반복이고 external_declaration에는 function_definition과 declaration만 존재한다. 함수의 전방선언은 declaration으로 reduce되고 함수의 정의는 function_definition으로 reduce된다. 따라서 이 둘 중 하나로 결정되는 external_declaration에서 tmp값을 cnt에 더해줄 수 있다.

3.2 function_definition

```
external_declaration
: function_definition
{funcnt++;
 intcnt+=inttmp; charcnt+=chartmp; pocnt+=potmp; arrcnt+=arrtmp;
 inttmp=chartmp=potmp=arrtmp=0;}
| declaration
| preprocessor
;

function_definition
: declaration_specifiers declarator compound_statement
| declarator compound_statement
;
```

declarator가 function_definition으로 reduce될 경우는 함수의 리턴값이기 때문에 이 때는 세지 않는다.

3.3 init_declarator

```

init_declarator
: declarator
{
chartmp=0; inttmp=0; arrtmp=0; potmp=0;
$$=0;
if($1%5==4) funcnt++;
if($1!=0&&$1!=2&&$1%5!=3) pocnt++;
if($1==0||$1==1||$1==2||$1==5||$1==7) $$=1;
if($1%5==2) arrcnt++;
}
| declarator '=' initializer
{opcnt++;
chartmp=0; inttmp=0; arrtmp=0; potmp=0;
$$=0;
if($1%5==4) funcnt++;
if($1!=0&&$1!=2&&$1%5!=3) pocnt++;
if($1==0||$1==1||$1==2||$1==5||$1==7) $$=1;
if($1%5==2) arrcnt++;
}
;

```

declarator가 parameter_declaration이 아닌 다른 곳으로 reduce된 경우는 tmp의 값을 0으로 초기화 해준다. 함수의 파라미터가 아닌 일반 선언에서는 변수의 갯수를 바로 세지 않고 변수를 셀 수 있는지 여부를 심볼 값으로 전달한다. (\$\$가 1이라는 것은 변수로 카운팅 가능함을 알린다.) 이는 아직 변수의 타입을 모르기 때문이고, int a,b와 같은 여러 변수를 동시에 한번에 선언할 때의 카운팅도 가능하게 한다. 함수,포인터,배열의 갯수는 바로 세어준다.

```

init_declarator_list
: init_declarator                {$$=$$+1;}
| init_declarator_list ',' init_declarator {$$=$$+1+$3;}
;

```

init_declarator_list의 심볼 값은 init_declarator 심볼 값의 1의 갯수를 세도록 되어있다.

```

declaration_specifiers
: storage_class_specifier                {$$=0;}
| storage_class_specifier declaration_specifiers {$$=$$+2;}
| type_specifier                        {$$=$$+1;}
| type_specifier declaration_specifiers {$$=$$+1;}
| type_qualifier                       {$$=0;}
| type_qualifier declaration_specifiers {$$=$$+2;}
;

```

```

type_specifier
: VOID                {$$=1;}
| CHAR                {$$=2;}
| SHORT               {$$=3;}

```

```

| INT          {$$=4;}
| LONG         {$$=5;}
| FLOAT        {$$=6;}
| DOUBLE       {$$=7;}
| SIGNED       {$$=8;}
| UNSIGNED     {$$=9;}
| struct_or_union_specifier {$$=10;}
| enum_specifier {$$=11;}
| TYPE_NAME    {$$=12;}
;

```

declaration_specifiers는 type_specifier값을 전달해준다. type_specifier의 심볼 값은 종류 별로 다른 값을 갖는다.

```

declaration
: declaration_specifiers ','
| declaration_specifiers init_declarator_list ','
    {if($1==4)intcnt+=$2; if($1==2)charcnt+=$2;}
| TYPEDEF declaration_specifiers init_declarator_list ','
    {
    if($2==4)intcnt+=$3; if($2==2)charcnt+=$3;
    typedef_li[typedef_cnt++]=name;
    }
;

```

declaration_specifiers와 init_declarator_list가 만나는 지점인 declaration에서 declaration_specifiers의 심볼 값으로 타입을 파악하고 init_declarator_list의 값 만큼 카운트한다.

3.4 struct_declarator

```

struct_declaration
: specifier_qualifier_list struct_declarator_list ',' {if($1==4)intcnt+=$2; if($1==2)charcnt+=$2;}
;

specifier_qualifier_list
: type_specifier specifier_qualifier_list {$$=$1;}
| type_specifier {$$=$1;}
| type_qualifier specifier_qualifier_list {$$=$2;}
| type_qualifier {$$=0;}
;

struct_declarator_list
: struct_declarator {$$=$1;}
| struct_declarator_list ',' struct_declarator {$$=$1+$3;}
;

struct_declarator
: declarator
{

```



```

        chartmp=0; inttmp=0; arrtmp=0; potmp=0;
        $$=0;
        if($1%5==4) funcnt++;
        if($1!=0&&$1!=2&&$1%5!=3) pocnt++;
        if($1==0||$1==1||$1==2||$1==5||$1==7) $$=1;
        if($1%5==2) arrcnt++;
    }
| ':' constant_expression          {$$=-1;}
| declarator ':' constant_expression
    {
        chartmp=0; inttmp=0; arrtmp=0; potmp=0;
        $$=0;
        if($1%5==4) funcnt++;
        if($1!=0&&$1!=2&&$1%5!=3) pocnt++;
        if($1==0||$1==1||$1==2||$1==5||$1==7) $$=1;
        if($1%5==2) arrcnt++;
    }
;

```

구조체에서는 일반 선언문과 동일하다.

4 조건문, 반복문

```

statement
: labeled_statement
| compound_statement
| expression_statement
| selection_statement {selcnt++;}
| iteration_statement {loopcnt++;}
| jump_statement
;

```

조건문은 selection_statement, 반복문은 iteration_statement이므로 그대로 세면 된다.

5 리턴문

```

jump_statement
: GOTO IDENTIFIER ';'
| CONTINUE ';'
| BREAK ';'
| RETURN ';' {retcnt++;}
| RETURN expression ';' {retcnt++;}
;

```

리턴문은 RETURN 토큰이 쓰인 곳에서 세어준다.

6 변수의 선언 위치

```

compound_statement

```

```

: '{' '}'
| '{' statement_list '}'
| '{' declaration_list '}'
| '{' declaration_list statement_list '}'
;
declaration_list
: declaration
| declaration_list declaration
;

statement_list
: statement
| statement_list statement
;

```

이것은 기존의 ANSI C Yacc 문법이다. declaration_list와 statement_list의 순서가 정해져 있기 때문에 변수의 전방선언만 허용된다.

```

compound_statement
: '{' '}'
| '{' declaration_statement_list '}'
;

declaration_statement_list
: declaration
| declaration declaration_statement_list
| statement
| statement declaration_statement_list
;

```

위와 같이 declaration과 statement의 순서가 무엇이든 허용되도록 고쳤다.

```

iteration_statement
: WHILE '(' expression ')' statement
| DO statement WHILE '(' expression ')' ';'
| FOR '(' expression_statement expression_statement ')' statement
| FOR '(' expression_statement expression_statement expression ')' statement
| FOR '(' declaration expression_statement ')' statement
| FOR '(' declaration expression_statement expression ')' statement
;

```

또한 for문의 괄호 안에서도 선언이 가능하도록 아래 두 줄을 추가했다.

7 typedef

```

declaration
: declaration_specifiers ';'
| declaration_specifiers init_declarator_list ';' {if($1==4)intcnt+=$2; if($1==2)charcnt+=

```

```

| TYPEDEF declaration_specifiers init_declarator_list ','
{
    if($2==4)intcnt+=$3; if($2==2)charcnt+=$3;
    typedef_li[typedef_cnt++]=name;
}
;

```

기존의 storage_class_specifier에 있던 TYPEDEF 토큰을 처리하기 위해 declaration에 옮겨 따로 규칙을 추가했다. typedef 안에서 명시적으로 선언된 변수도 카운팅을 해주고 typedef_li의 typedef_cnt 인덱스에 타입의 이름을 넣어준다.

```

int check_type()
{
    name=strdup(yytext);
    if(is_typedef(yytext)) return TYPE_NAME;
    else if(is_define(yytext)) return CONSTANT;
    else return IDENTIFIER;
}

```

name은 lex에서 식별자(가 아닐 수도 있는 문자와 숫자의 조합)의 타입을 검사할 때 그 식별자의 이름을 복사한 값이다. typedef로 선언되었는지를 검사하고 맞다면 TYPE_NAME 토큰을 반환한다.

```

extern char*name;
char*typedef_li[100];
int typedef_cnt=0;

int is_typedef(char*s){
    for(int i=0;i<typedef_cnt;i++){
        if(strcmp(s,typedef_li[i])==0)return 1;
    }
    return 0;
}

```

yacc파일에서는 lex에서 선언된 name을 받아주고 is_typedef함수에서 typedef_li를 순회하면서 일치하는 이름이 있는지 검사한다.

8 define

```

"define"          { return(DEFINE);}

"#"              { return(' '); }

```

lex에 위와 같은 규칙을 추가하였다.

define의 경우 typedef와 유사하게 처리한다.

```

char*define_li[100];
int define_cnt=0;
int is_define(char*s){

```

```

        for(int i=0;i<define_cnt;i++){
            if(strcmp(s,define_li[i])==0)return 1;
        }
        return 0;
    }

    define_constant
        : '#' DEFINE IDENTIFIER CONSTANT {define_li[define_cnt++]=name;}
        ;

```

마지막 파라미터가 상수인 경우만 처리했다.

```

int check_type()
{
    name=strdup(yytext);
    if(is_typedef(yytext)) return TYPE_NAME;
    else if(is_define(yytext)) return CONSTANT;
    else return IDENTIFIER;
}

```

lex에서 타입검사 시 define으로 할당되어 있는 값이라면 CONSTNAT 토큰으로 반환해준다.

9 include

```

"include".*"\" { return(INCLUDE); }

preprocessor
    : define_constant
    | '#' INCLUDE
    ;

external_declaration
    : function_definition
    {funcnt++;
    intcnt+=inttmp; charcnt+=chartmp; pocnt+=potmp; arrcnt+=arrtmp;
    inttmp=chartmp=potmp=arrtmp=0;}
    | declaration
    | preprocessor
    ;

```

위와 같이 include와 define도 파싱이 되도록 수정했다.

10 주석

```

"/*" { comment(); }
"//".*"\" {;}

```

```

void comment()
{
    char c, c1;

loop:
    while ((c = input()) != '*' && c != 0);

    if ((c1 = input()) != '/' && c1 != 0)
    {
        unput(c1);
        goto loop;
    }
}

```

주석은 위와 같이 처리하였다. lex의 규칙은 longest matching을 기본으로 하지만 주석은 shorest matching을 해야한다. 그렇지 않으면 주석을 여러개 사용했을 때 그 사이에 끼어있는 일반 코드가 주석처리 될 수 있기 때문이다. shorest matching을 정규표현식으로 처리하는 것은 어렵기 때문에 기존의 comment함수를 이용했다. 그리고 출력은 되지 않도록 putchar는 지웠다.