Programming Language Assignment

학번 : C019059 이름 : 신영기

May 28, 2025

1 과제 2 : Yacc

과목 : 프로그래밍언어론
학수번호 : 101512-002
이름 : 신영기
학번 : C019059
전공 : 화학공학
복수전공 : 컴퓨터공학

2 Yacc 동작방식

2.1 Lex 코드

Lex코드는 정의절, 규칙절, 서브루틴절로 이루어져있다. 정의절에는 include할 헤더들을 포함하는데 여기서 y.tab.h 파일은 프로그래머가 만든 Yacc의 토큰 번호와 기타 변수들의 정보를 포함하고 있는데 이는 꼭 필요한 정보이기 때문에 include를 해줘야한다. 또한 정규표현식을 활용하여 원하는 데이터의 형태를 간단한 변수로 정의해둘 수 있다. 다음으로 규칙절은 특정 패턴별로 원하는 반환 토큰 규칙을 설정할 수 있다. 마지막으로 서브루틴절은 사용자가 원하는 동작을 지정하는 절로, 토큰 분석을 위한 추가 함수 등을 선언하는 부분이다.

2.2 yacc 코드

yacc파일의 코드도 정의절, 규칙절, 서브루틴절로 이루어져있다. 정의절은 마찬가지로 헤더를 불러오거나 필요한 변수, define등을 수행하는 구간이다. 또한 lex에서 받은 토큰들을 정의 및 파서의 시작 심볼 지정도 한다. 다음으로 규칙절은 bnf문법을 활용하여 문법 구조를 명시하고, 추가적인 코드 작성도 포함할 수 있다. 마지막으로 서브루틴절은 bnf문법에 맞게 실행한 결과에 대해 프로그래머가 지정한 동작을 수행하거나 예외처리를 선언해둘 수 있다.

2.3 yacc 동작 방식

정리하면 yacc는 정의절, 규칙절, 서브루틴절으로 이루어져있다. 정의절은 보통 %{와 %}로 감싸져있다. 하지만 규칙절의 시작을 알리는 %% 전까지는 모두 정의절로 해석된다. 다음으로 규칙절은 %%로 감싸져있고, 규칙절이 끝난 다음은 모두 서브루틴절이다.

lex파일은 어휘 분석기를 만드는 도구로, 주어진 입력스트림을 프로그래머가 지정한 토큰단위로 반환해주는 역할을 한다. yacc는 구문 분석기를 만드는 도구로, lex에서 생성한 토큰들을 문법에 부합하는지 검사하는 역할을 한다.

yacc의 동작방식은 결국 lex가 분석한 토큰들을 stack에 저장한 후 stack의 내용에 따라 yacc파일의 규칙절에 지정된대로 state가 변하게 된다. 각 토큰이 어떤 조합과 순서로 들어오는지에 따라서 지정된 state로 변경되고, 만약 해당 토큰의 조합과 순서가 지정된 state에 등록되어있지 않다면, SYNTAX 에러를 반환하게된다. 또한 하나의 토큰 조합이 두 가지의 state를 동시에 가실 수 있는 경우는 모호성이 발생해 shift/reduce 에러가 발생한다. 이는 하나의 입력이 두개 이상의 tree를 가지는 경우로 규칙절의 내용을 적절히 수정하여 고칠 수 있다.

3 구현 코드

ANSI C Grammar의 내용을 그대로 가져온 부분은 추가 설명 주석을 달지 않았습니다.

3.1 hw3.l파일

```
D   [0-9]
L   [a-zA-Z_]
H   [a-fA-F0-9]
E   [Ee][+-]?{D}+
FS   (f|F|l|L)
IS   (u|U|l|L)*
ID      [A-Za-z_][A-Za-z0-9_]*

%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include "y.tab.h"
```

```
void count();
void comment();
void lineComment();
int check_type();
int is_typedef_name(const char *s);
int typedef_start_flag; // typedef의 시작과 끝을 확인하기 위한 flag
입니다.
int curly_brace_flag; // 구조체의 시작과 끝을 확인하기 위한 flag입
니다.
/* 심볼 테이블 최대 개수 */
#define MAX_SYMBOLS 1024

// 사용자 지정 변수, 자료형을 따로 저장해두기 위해 심볼테이블을 활용했
습니다.
/* typedef 이름 테이블 */
static char *typedef_names[MAX_SYMBOLS];
static int   typedef_count = 0;
/* #define 이름 테이블 */
static char *define_names[MAX_SYMBOLS];
static int   define_count = 0;


/* typedef 이름 추가 */
void add_typedef(const char *s) {
    if (typedef_count < MAX_SYMBOLS)
        typedef_names[typedef_count++] = strdup(s);
}

/* typedef 여부 확인 */
int is_typedef(const char *s) {
    for (int i = 0; i < typedef_count; i++)
        if (strcmp(s, typedef_names[i]) == 0)
            return 1;
    return 0;
}

/* #define 이름 추가 */
void add_define(const char *s) {
    if (define_count < MAX_SYMBOLS)
        define_names[define_count++] = strdup(s);
}

/* #define 여부 확인 */
int is_define(const char *s) {
```

```
    for (int i = 0; i < define_count; i++)
        if (strcmp(s, define_names[i]) == 0)
            return 1;
    return 0;
}

// typedef 시작할 때 flag 초기화 함수
void start_typedef(void)
{
 typedef_start_flag = 1;
 curly_brace_flag = -1;
}
%}


%%
"/*"   { comment(); }  // 여러줄 주석부분 무시를 위한 함수 호출후 토
큰 반환하지 않음
"//"   { lineComment(); } // 한줄 주석 무시를 위한 함수 호출후 토큰 반
환하지 않음
^"#include".*\n    { ; } // #include 로 헤더를 호출하는 부분은 개행
까지 무시하고 토큰은 반환하지 않는 방법으로 syntax에러를 방지했습니다.
^"#define"[ \t]+{ID}.*\n {
 char namebuf[128];
    sscanf(yytext, "#%*s %127s", namebuf);
    add_define(namebuf);
 /*
  define으로 전처리 선언된 변수의 이름을 저장하여 사용자 지정 type으
로 저장합니다.
 */
}
"typedef"  { count(); start_typedef(); return(TYPEDEF);}
"auto"   { count(); return(AUTO); }
"break"   { count(); return(BREAK); }
"case"   { count(); return(CASE); }
"char"   { count(); return(CHAR); }
"const"   { count(); return(CONST); }
"continue"  { count(); return(CONTINUE); }
"default"  { count(); return(DEFAULT); }
"do"   { count(); return(DO); }
"double"  { count(); return(DOUBLE); }
"else if"   { count(); return(ELSEIF); }
"else"   { count(); return(ELSE); }
"enum"   { count();
```

```
 if (typedef_start_flag)
  typedef_start_flag = 2;
 return(ENUM);
 //typedef_start_flag를 괄호로 감싸져 있다는 것을 나타내는 2로 설정
합니다.
 }
"extern"  { count(); return(EXTERN); }
"float"   { count(); return(FLOAT); }
"for"   { count(); return(FOR); }
"goto"   { count(); return(GOTO); }
"if"   { count(); return(IF); }
"int"   { count(); return(INT); }
"long"   { count(); return(LONG); }
"register"  { count(); return(REGISTER); }
"return"  { count(); return(RETURN); }
"short"   { count(); return(SHORT); }
"signed"  { count(); return(SIGNED); }
"sizeof"  { count(); return(SIZEOF); }
"static"  { count(); return(STATIC); }
"struct"  { count();
 if (typedef_start_flag)
  typedef_start_flag = 2;
 return(STRUCT);
 //typedef_start_flag를 괄호로 감싸져 있다는 것을 나타내는 2로 설정
합니다.
 }
"switch"  { count(); return(SWITCH); }
"union"   { count();
 if (typedef_start_flag)
  typedef_start_flag = 2;
 return(UNION);
 //typedef_start_flag를 괄호로 감싸져 있다는 것을 나타내는 2로 설정
합니다.
 }
"unsigned"  { count(); return(UNSIGNED); }
"void"   { count(); return(VOID); }
"volatile"  { count(); return(VOLATILE); }
"while"   { count(); return(WHILE); }

{L}({L}|{D})*  { count(); return(check_type()); }

0[xX]{H}+{IS}?  { count(); return(CONSTANT); }
0{D}+{IS}?  { count(); return(CONSTANT); }
{D}+{IS}?  { count(); return(CONSTANT); }
```

```
L?'(\\.|[^\\'])+' { count(); return(CONSTANT); }

{D}+{E}{FS}?  { count(); return(CONSTANT); }
{D}*"."{D}+({E})?{FS}? { count(); return(CONSTANT); }
{D}+"."{D}*({E})?{FS}? { count(); return(CONSTANT); }

L?\"(\\.|[^\\"])*\" { count(); return(STRING_LITERAL); }

"..."   { count(); return(ELLIPSIS); }
">>="   { count(); return(RIGHT_ASSIGN); }
"<<="   { count(); return(LEFT_ASSIGN); }
"+="   { count(); return(ADD_ASSIGN); }
"-="   { count(); return(SUB_ASSIGN); }
"*="   { count(); return(MUL_ASSIGN); }
"/="   { count(); return(DIV_ASSIGN); }
"%="   { count(); return(MOD_ASSIGN); }
"&="   { count(); return(AND_ASSIGN); }
"^="   { count(); return(XOR_ASSIGN); }
"|="   { count(); return(OR_ASSIGN); }
">>"   { count(); return(RIGHT_OP); }
"<<"   { count(); return(LEFT_OP); }
"++"   { count(); return(INC_OP); }
"--"   { count(); return(DEC_OP); }
"->"   { count(); return(PTR_OP); }
"&&"   { count(); return(AND_OP); }
"||"   { count(); return(OR_OP); }
"<="   { count(); return(LE_OP); }
">="   { count(); return(GE_OP); }
"=="   { count(); return(EQ_OP); }
"!="   { count(); return(NE_OP); }
";"   { count(); return(';'); }
("{"|"<%")  { count();
 if (typedef_start_flag == 2)
 {
  if (curly_brace_flag == -1)
   curly_brace_flag = 0;
  curly_brace_flag++;
 // 만약 괄호로 감싸져 있어야하는 자료형인 경우에는 자료형의 선언 시작
과 끝을 구분하기 위해 curly_brace_flag를 이용하여 괄호의 짝을 맞춥니
다.
 }
 return('{'); }
("}"|"%>")  { count();
 if (typedef_start_flag == 2)
```

```
  curly_brace_flag--;
 // 만약 괄호로 감싸져 있어야하는 자료형인 경우에는 자료형의 선언 시작
과 끝을 구분하기 위해 curly_brace_flag를 이용하여 괄호의 짝을 맞춥니
다.
 return('}'); }
","   { count(); return(','); }
":"   { count(); return(':'); }
"="   { count(); return('='); }
"("   { count(); return('('); }
")"   { count(); return(')'); }
("["|"<:")  { count(); return('['); }
("]"|":>")  { count(); return(']'); }
"."   { count(); return('.'); }
"&"   { count(); return('&'); }
"!"   { count(); return('!'); }
"~"   { count(); return('~'); }
"-"   { count(); return('-'); }
"+"   { count(); return('+'); }
"*"   { count(); return('*'); }
"/"   { count(); return('/'); }
"%"   { count(); return('%'); }
"<"   { count(); return('<'); }
">"   { count(); return('>'); }
"^"   { count(); return('^'); }
"|"   { count(); return('|'); }
"?"   { count(); return('?'); }

[ \t\v\n\f]  { count(); }
.   { /* ignore bad characters */ }

%%

//입력 스트림의 끝(EOF)에 도달하면 입력을 종료시키는 함수
int yywrap()
{
 return(1);
}

/*
comment함수는 여러 줄 주석을 처리하기 위한 함수로 "*"문자가 나오기 전
까지 모든 문자를 무시하고
이후 다음 문자가 "/"이라면 종료하고 아니라면 다시 "*"문자 확인 과정으
로 돌아가서 반복합니다.
*/
```

```
void comment()
{
 char c, c1;

 while (1)
 {
  c = input();
  if (c == '*')
  {
   if ((c1 = input()) != '/' && c != 0)
    unput(c1);
   else
    break;
  }
 }
}

void lineComment()
{
 char c, c1;

 while ((c = input()) != '\n' && c != 0)
  ;
}

/*
아래는
ANSI C grammar, LEX에 있던 부분으로 현재 열 번호를 저장하여 에러메세
지에 활용할 때 사용하였습니다.
*/
int column = 0;
void count()
{
 int i;

 for (i = 0; yytext[i] != '\0'; i++)
  if (yytext[i] == '\n')
   column = 0;
  else if (yytext[i] == '\t')
   column += 8 - (column % 8);
  else
   column++;
}
```

```
/*
아래 함수는 변수의 자료형을 확인하는 함수로 각 case에 맞는 flag를 확
인 및 수정 후 알맞은 자료형을 반환합니다.
*/
int check_type()
{
 // 만약 typedef가 앞에 붙어있는 자료형인 경우 세 개의 케이스로 나눠
서 처리하였습니다.
 if (typedef_start_flag)
 {
  /*
  typedef_start_flag가 2라서 중괄호를 이용하려 선언하는 자료형이고,
  curly_brace_flag가 0으로 모든 중괄호의 쌍이 맞게 끝난 경우는
  선언이 완료되고 별칭이 붙는 경우이다.
  따라서 이 경우는 플래그를 초기화하고 자료형을 추가한다.
  */
  if (typedef_start_flag == 2 && curly_brace_flag == 0)
  {
   typedef_start_flag = 0;
   add_typedef(yytext);
  }
  /*
  typedef_start_flag가 1인 경우는 중괄호가 필요하지 않는 자료형 혹은 변
수이므로 바로 선언이 완료된 것으로
  판단하고 모든 플래그를 초기화하고 자료형을 추가한다.
  */
  else if (typedef_start_flag == 1)
  {
   typedef_start_flag = 0;
   curly_brace_flag = 0;
   add_typedef(yytext);
  }
  /*
  마지막으로 typedef_start_flag가 2이므로 중괄호가 필요한 자료형인데 curly_brace_flag
가 -1인 경우는
  아직 중괄호를 만나지 않고 해당 자료형의 이름만 만난 부분이므로, 현재 이
름이 선언되었다는 표시로
  curly_brace_flag를 0으로 바꾸고 별칭을 만날 때까지 다시 어휘 분석
을 진행한다.
  */
  else if (typedef_start_flag == 2 && curly_brace_flag == -1)
   curly_brace_flag = 0;
 }
 /*
```

```
 아래 조건문은 현재 입력 단어가 typedef, define로 이전에 선언된 변수 혹
은 자료형인지 확인 후 각각에 맞는 토큰을 반환하고
 만약 둘 다 아니라면 일반 식별자인 IDENTIFIER을 반환한다.
 */
 if (is_typedef(yytext)) {
        return TYPE_NAME;   /* typedef 이름 */
    }
    else if (is_define(yytext)) {
        return CONSTANT;   /* #define 상수 이름 */
    }
    else {
        return IDENTIFIER; /* 일반 식별자 */
    }
}
```

3.2 hw3.y파일

```
%{

#include <stdio.h>

int ary[9] = {0,0,0,0,0,0,0,0,0};

//배열의 인덱스를 헷갈리지 않게하기위해 가독성을 높이는 전처리를 실시
하였습니다.
#define FUNCTION 0
#define OPERATOR 1
#define INTEGER  2
#define CHARACTER 3
#define POINTER  4
#define ARRAY  5
#define SELECTION 6
#define LOOP  7
#define RET   8

int yylex(void);
struct YYLTYPE;
int idx; // 어떤 자료형의 count를 증가시켜야할지 나타내는 인덱스 변수
입니다.
int pointerFlag; // 이 변수가 포인터 변수인지 자료형 변수인지 확인하
는 플래그입니다.
int arrayFlag; // 배열을 대괄호를 기준으로 셀 때 다차원 배열을 중복으
로 count하지 않기 위한 flag입니다.
```

```
void yyerror(const char *msg); // 디버깅 용으로 사용하였습니다.
%}

%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSEIF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN
%start translation_unit

%%
/* function declaration */
translation_unit
 : external_declaration
 | translation_unit external_declaration
 ;


function_definition
 : declaration_specifiers declarator compound_statement
 | declarator compound_statement
 ;

external_declaration
 : function_definition
 | declaration
 ;

declaration_specifiers
 : storage_class_specifier
 | storage_class_specifier declaration_specifiers
 | type_specifier
 | type_specifier declaration_specifiers
 | type_qualifier
 | type_qualifier declaration_specifiers
 ;
```

```
/*
 예전 c문법은 변수의 선언은 무조건 가장 앞에서만 가능했고 테스트코드처
럼 선언문과 할당문이 섞일 수 없었습니다.
 따라서 declaration_list 후 statement_list가 나올 수 있었던 기존의 코
드를 수정하여 두 심볼이 섞여서
 나올 수 있는 compound_list를 새로 만들었습니다.
*/
compound_statement
 : '{' compound_list '}'
 ;

compound_list
 : declaration
 | statement
 | compound_list declaration
 | compound_list statement
 ;

/* function call */
// 소괄호가 들어간 부분은 함수 호출 또는 선언 부분이므로 하수 호출 카
운트를 증가하였습니다.
// 또한 포인터 오퍼레이터, 증가, 감소 연산이 있는 부분은 연산자 호출을 증
가하였습니다.
postfix_expression
 : primary_expression
 | postfix_expression '[' expression ']'
 | postfix_expression '(' ')' {ary[FUNCTION]++;}
 | postfix_expression '(' argument_expression_list ')' {ary[FUNCTION]++;}
 | postfix_expression '.' IDENTIFIER {ary[OPERATOR]++;} // 구조체 접
근 연산자
 | postfix_expression PTR_OP IDENTIFIER  {ary[OPERATOR]++;} // 포
인터 연산자
 | postfix_expression INC_OP {ary[OPERATOR]++;} // 증가연산자
 | postfix_expression DEC_OP {ary[OPERATOR]++;} // 감소연산자
 ;


unary_expression
 : postfix_expression
 | INC_OP unary_expression {ary[OPERATOR]++;} // 증가 연산자
 | DEC_OP unary_expression {ary[OPERATOR]++;} // 감소 연산자
 | unary_operator cast_expression
 | SIZEOF unary_expression
 | SIZEOF '(' type_name ')'
```

```
assignment_expression
 : conditional_expression
 | unary_expression assignment_operator assignment_expression
 ;

/* operation */
unary_operator
 : '&'
 | '*'
 | '+'
 | '-'
 | '~'
 | '!'
 ;

cast_expression
 : unary_expression
 | '(' type_name ')' cast_expression {ary[OPERATOR]++;} // 캐스팅
은 연사자로 취급하여 카운트하였습니다.
 ;

multiplicative_expression
 : cast_expression
 | multiplicative_expression '*' cast_expression {ary[OPERATOR]++;} // 곱
하기 연산자 카운트
 | multiplicative_expression '/' cast_expression {ary[OPERATOR]++;} // 나
누기 연산자 카운트
 | multiplicative_expression '%' cast_expression {ary[OPERATOR]++;} // 나
머지 연산자 카운트
 ;

additive_expression
 : multiplicative_expression
 | additive_expression '+' multiplicative_expression {ary[OPERATOR]++;} // 더
하기 연산자 카운트
 | additive_expression '-' multiplicative_expression {ary[OPERATOR]++;} // 빼
기 연산자 카운트
 ;

shift_expression
 : additive_expression
 | shift_expression LEFT_OP additive_expression {ary[OPERATOR]++;} // 쉬
프트 연산자 카운트
```

```
 | shift_expression RIGHT_OP additive_expression {ary[OPERATOR]++;} // 쉬
프트 연산자 카운트
 ;

relational_expression
 : shift_expression
 | relational_expression '<' shift_expression {ary[OPERATOR]++;} // 비
교연산자 카운트
 | relational_expression '>' shift_expression {ary[OPERATOR]++;} // 비
교연산자 카운트
 | relational_expression LE_OP shift_expression {ary[OPERATOR]++;} // 비
교연산자 카운트
 | relational_expression GE_OP shift_expression {ary[OPERATOR]++;} // 비
교연산자 카운트
 ;

equality_expression
 : relational_expression
 | equality_expression EQ_OP relational_expression {ary[OPERATOR]++;} // 비
교연산자 카운트
 | equality_expression NE_OP relational_expression {ary[OPERATOR]++;} // 비
교연산자 카운트
 ;

and_expression
 : equality_expression
 | and_expression '&' equality_expression {ary[OPERATOR]++;} // 비
트연산자 카운트
 ;

exclusive_or_expression
 : and_expression
 | exclusive_or_expression '^' and_expression {ary[OPERATOR]++;} // 비
트연산자 카운트
 ;

inclusive_or_expression
 : exclusive_or_expression
 | inclusive_or_expression '|' exclusive_or_expression {ary[OPERATOR]++;} // 비
트연산자 카운트
 ;

logical_and_expression
 : inclusive_or_expression
```

```
 | logical_and_expression AND_OP inclusive_or_expression {ary[OPERATOR]++;} // 논
리연산자 카운트
 ;

logical_or_expression
 : logical_and_expression
 | logical_or_expression OR_OP logical_and_expression {ary[OPERATOR]++;} // 논
리연산자 카운트
 ;

conditional_expression
 : logical_or_expression
 | logical_or_expression '?' expression ':' conditional_expression // 삼
항연산자는 카운트하지 않았습니다.
 ;

/* operation */
assignment_operator
 : '=' {ary[OPERATOR]++;} // 대입연산자 증가
 | MUL_ASSIGN {ary[OPERATOR]++;} // 산술대입연산자 증가
 | DIV_ASSIGN {ary[OPERATOR]++;} // 산술대입연산자 증가
 | MOD_ASSIGN {ary[OPERATOR]++;} // 산술대입연산자 증가
 | ADD_ASSIGN {ary[OPERATOR]++;} // 산술대입연산자 증가
 | SUB_ASSIGN {ary[OPERATOR]++;} // 산술대입연산자 증가
 | LEFT_ASSIGN {ary[OPERATOR]++;} // 시프트대입연산자 증가
 | RIGHT_ASSIGN {ary[OPERATOR]++; } // 시프트대입연산자 증가
 | AND_ASSIGN {ary[OPERATOR]++;} // 비트대입연산자 증가
 | XOR_ASSIGN {ary[OPERATOR]++;} // 비트대입연산자 증가
 | OR_ASSIGN {ary[OPERATOR]++;} // 비트대입연산자 증가
 ;
/*
타입의 종류별로 인덱스를 구분하는데 출력에 필요로하는 int와 char의 인
덱스만 저장하고 나머지는 인덱스를 -1로 만들어서
카운트 할 type이 아니라는 것을 저장해둠.
*/

type_specifier
 : VOID {idx = -1;}
 | CHAR {idx = CHARACTER;}
 | SHORT {idx = -1;}
 | LONG {idx = -1;}
 | DOUBLE {idx = -1;}
 | SIGNED {idx = -1;}
 | INT {idx = INTEGER;}
```

```
 | UNSIGNED {idx = -1;}
 | FLOAT {idx = -1;}
 | TYPE_NAME {idx = -1;}
 | struct_or_union_specifier {idx = -1;}
 | enum_specifier {idx = -1;}
 ;

/* iteration count */
//while과 do while, for문은 LOOP로 카운트한다.
iteration_statement
 : WHILE '(' expression ')' statement {ary[LOOP]++;}
 | DO statement WHILE '(' expression ')' ';' {ary[LOOP]++;}
 | FOR '(' expression_statement expression_statement ')' statement {ary[LOOP]++;}
 | FOR '(' expression_statement expression_statement expression ')' statement {ary[L
 ;

statement
 : compound_statement
 | expression_statement
 | selection_statement
 | iteration_statement
 | jump_statement
 ;

/* for counting pointer */
declarator
 : pointer direct_declarator {
  ary[POINTER]++;
  if (arrayFlag)
  {
   ary[ARRAY]++;
   arrayFlag = 0;
   idx = -1;
  }
  }
  /*
```

포인터의 수를 여기서 증가시킵니다. 포인터는 여러번 반복되어도 하나의 포인터 카운트만 증가되어야 하므로 '*'의 한 번 이상의 반복으로 이루어진 pointer 심볼이 나오는 경우만 카운트를 증가시키면 중복 증가가 발생하지 않습니다.

또한 대괄호 기준으로 배열 수를 카운트하면 다차원 배열은 여러개의 배열로 중복 카운트 될 수 있기 때문에 대괄호가 나오면 arrayFlag를 1로 바꾸고

그러한 배열이 포함되어 있을 수 있는 direct_declarator 심볼이 반환

되었을 때 플래그를 확인하고 배열 수를 카운트하였습니다.
  또한 포인터의 배열인 경우 저장한 자료형은 int나 char이 아닌 8바이트 포인터이기 때문에 더블포인터와 마찬가지로 자료형의 선언으로 보지 않았습니다.
  그래서 idx를 -1로 설정하여 어떤 자료형의 카운트도 증가시키지 않았습니다.

```
  */
 | direct_declarator {
  if (arrayFlag)
   ary[ARRAY]++;
  arrayFlag = 0;
 }// 이 부분도 위의 array 카운트 논리와 동일합니다.
 ;


pointer
 : '*'
 | '*' type_qualifier_list
 | '*' pointer {idx = -1;}// 자료형 증가 x
 | '*' type_qualifier_list pointer {idx = -1;}// 자료형 증가 x
 ;
/*
 여기서 포인터가 이중 포인터 이상이 되는 순간 int, char의 개수는 증가시키지 않고 포인터의 개수만 증가되어야 하기 때문에
 포인터가 반복되면 증가시킬 자료형을 나타내는 idx의 값을 -1로 변경하여 어떠한 자료형의 카운트도 증가시키지 않는것을
 나타내었습니다.
*/

//대괄호가 존재하면 배열의 선언임을 알 수 있고, 이를 상위 심볼에서 활용하기 위해 arrayFlag를 1로 변경하였습니다.
direct_declarator
 : IDENTIFIER
 | '(' declarator ')'
 | direct_declarator '[' constant_expression ']' {arrayFlag = 1;}
 | direct_declarator '[' ']' {arrayFlag = 1;}
 | direct_declarator '(' parameter_type_list ')' {ary[FUNCTION]++; idx = -1;}
 | direct_declarator '(' identifier_list ')' {ary[FUNCTION]++; idx = -1;}
 | direct_declarator '(' ')' {ary[FUNCTION]++; idx = -1;}
 ;
 /*
 위 세 코드는 함수의 선언 부분으로 이 경우 리턴값을 나타내는 부분에서 int나 char가 올 수 있는데 해당 자료형으로 변수를 선언한 것이 아니라 함수의 리턴값을 명시한 부분입니다.
```

따라서 자료형은 증가하지 않고, 함수의 카운트는 증가해야하기 때문에 함수의 카운트를 증가시키고 idx를 -1로 두어 자료형 카운트를 증가하지 않는 것을 나타내였습니다.
*/

//if문과 switch문만 Selection문으로 카운트 하였습니다.
//또한 기존의 if else문의 구성이 아닌, if else if else의 구성으로 변경하였습니다.
```
selection_statement
 : IF '(' expression ')' statement {ary[SELECTION]++;}
 | ELSEIF '(' expression ')' statement
 | ELSE statement
 | SWITCH '(' expression ')' statement {ary[SELECTION]++;}
 ;

//RETURN 토큰이 나오면 return의 카운트를 증가하였습니다.
jump_statement
 : GOTO IDENTIFIER ';'
 | CONTINUE ';'
 | BREAK ';'
 | RETURN ';' {ary[RET]++;}
 | RETURN expression ';' {ary[RET]++;}

declaration
 : declaration_specifiers ';'
 | declaration_specifiers init_declarator_list ';'
 ;

/*
변수의 선언이 이루어지는 심볼로 단순히 INT, CHAR 토큰이 들어왔을 때 증가시키면 더블 포인터, 또는 int a, b;이런 코드에서 정확한 타입 개수를 세지 못합니다.
따라서 선언이 이루어지는 단위로 나눠진 심볼인 init_declarator가 반복될 때마다 선언 자료형이 int, char이라면 카운트를 증가시키는 방식으로 구현하였습니다.
*/
init_declarator_list
 : init_declarator {
  if (idx != -1)
   ary[idx]++;
 }
 | init_declarator_list ',' init_declarator   {
  if (idx != -1)
   ary[idx]++;
```

```
  idx = -1;
 }
 ;

//대입 연산자의 카운트를 증가하였습니다.
init_declarator
 : declarator
 | declarator '=' initializer {ary[OPERATOR]++;}
 ;

initializer
 : assignment_expression
 | '{' initializer_list '}'
 | '{' initializer_list ',' '}'
 ;

initializer_list
 : initializer
 | initializer_list ',' initializer
 ;

type_qualifier
 : CONST
 | VOLATILE
 ;

primary_expression
 : IDENTIFIER
 | CONSTANT
 | STRING_LITERAL
 | '(' expression ')'
 ;

expression
 : assignment_expression
 | expression ',' assignment_expression
 ;


argument_expression_list
 : assignment_expression
 | argument_expression_list ',' assignment_expression
 ;
```

```
type_name
 : specifier_qualifier_list
 | specifier_qualifier_list abstract_declarator
 ;

expression_statement
 : ';'
 | expression ';'
 ;


type_qualifier_list
 : type_qualifier
 | type_qualifier_list type_qualifier
 ;

constant_expression
 : conditional_expression
 ;


parameter_type_list
 : parameter_list
 | parameter_list ',' ELLIPSIS
 ;

parameter_list
 : parameter_declaration
 | parameter_list ',' parameter_declaration
 ;

/*
 여기서 init_declarator와 마찬가지로 매개변수의 선언이 이루어지는 부
분에서도 변수의 타입별 증가가 이루어질 수 있도록
 동일한 로직으로 구현하였습니다.
*/
parameter_declaration
 : declaration_specifiers declarator
 {
  if (idx != -1)
   ary[idx]++;
  idx = - 1;
 }
 | declaration_specifiers abstract_declarator {idx = -1;}
```

```
 | declaration_specifiers
 ;

identifier_list
 : IDENTIFIER
 | identifier_list ',' IDENTIFIER
 ;

specifier_qualifier_list
 : type_specifier specifier_qualifier_list
 | type_specifier
 | type_qualifier specifier_qualifier_list
 | type_qualifier
 ;

abstract_declarator
 : pointer
 | direct_abstract_declarator
 | pointer direct_abstract_declarator
 ;

//아래와같은 배열과 함수타입의 중첩 코드는 나오지 않는다고 말씀해주셔
서 별도 처리는 하지 않았습니다.
direct_abstract_declarator
 : '(' abstract_declarator ')'
 | '[' ']'
 | '[' constant_expression ']'
 | direct_abstract_declarator '[' ']'
 | direct_abstract_declarator '[' constant_expression ']'
 | '(' ')'
 | '(' parameter_type_list ')'
 | direct_abstract_declarator '(' ')'
 | direct_abstract_declarator '(' parameter_type_list ')'
 ;

storage_class_specifier
 : TYPEDEF
 | EXTERN
 | STATIC
 | AUTO
 | REGISTER
 ;

struct_or_union
```

```
 : STRUCT
 | UNION
 ;

struct_or_union_specifier
 : struct_or_union IDENTIFIER '{' struct_declaration_list '}'
 | struct_or_union '{' struct_declaration_list '}'
 | struct_or_union IDENTIFIER
 ;

struct_declaration_list
 : struct_declaration
 | struct_declaration_list struct_declaration
 ;

struct_declaration
 : specifier_qualifier_list struct_declarator_list ';'
 ;

/*
 아래 코드도 init_declarator와 마찬가지로 구조체 내부 변수 선언이 이
루어지는 부분에서도 변수의 타입별 증가가 이루어질 수 있도록
 동일한 로직으로 구현하였습니다.
*/
struct_declarator_list
 : struct_declarator{
  if (idx != -1)
   ary[idx]++;
 }
 | struct_declarator_list ',' struct_declarator{
  if (idx != -1)
   ary[idx]++;
  idx = -1;
 }
 ;

struct_declarator
 : declarator
 | ':' constant_expression
 | declarator ':' constant_expression
 ;

enum_specifier
 : ENUM '{' enumerator_list '}'
```

```
 | ENUM IDENTIFIER '{' enumerator_list '}'
 | ENUM IDENTIFIER
 ;

enumerator_list
 : enumerator
 | enumerator_list ',' enumerator
 ;

enumerator
 : IDENTIFIER
 | IDENTIFIER '=' constant_expression
 ;

%%

void yyerror(const char *msg)
{
    fprintf(stderr,"%s\n", msg);
}

int main(void)
{
 yyparse();
 printf("function = %d\n", ary[0]);
 printf("operator = %d\n", ary[1]);
 printf("int = %d\n", ary[2]);
 printf("char = %d\n", ary[3]);
 printf("pointer = %d\n", ary[4]);
 printf("array = %d\n", ary[5]);
 printf("selection = %d\n", ary[6]);
 printf("loop = %d\n", ary[7]);
 printf("return = %d\n", ary[8]);
 return 0;
}
```