프로그래밍 언어 HW3

B811079 방병훈

20230510

1 YACC에 대한 설명

1.1 YACC란?

Yacc란 Yet Another Compiler Compiler의 약자로 LEX와 같이 AT&T Bell 연구소에서 Mark Lesk가 UNIX 시스템의 유틸리티로 개발한 소프트웨어이다. Yacc는 BNF와 같은 형식의 규칙의 항목으로부터 파서를 만들어내는 프로그램이고, UNIX 시스템의 표준 파서 생성기이다. LEX가 정규표현식을 확인한다면, YACC는 문법을 확인한다. Yacc는 lex로 구한 토큰들의 관계를 구성하는 구문 분석기를 생성하는 툴이다. 따라서, lex가 이뤄진 후에 작동을 하며 lex가 생성한 어휘 분석기 yylex()를 내부적으로 호출한다. Yacc specification 파일 안에서 유저 루틴 내에서 yyparse()를 호출하면 yyparse()가 yylex()를 호출해서 주어진 문법에 부합하는지 검사한다. 여기서 파싱이란 주어진 문법 규칙에 입력 토큰들이 일치하는지 확인하는 과정이고 문법은 토큰들의 유효한 순서를 나타내고 순서가 일치할 경우 어떻게 해야할지 나타낸다. 즉, 토큰 자체는 lex에서 나오고 이 토큰들간의 순서인 문법은 Yacc에 지정된다.

1.2 YACC의 구조

정의절
정의절에서는 문법에서 사용할 토큰을 선언하고 구문분석기(parser)에서 사용할 타입들을 선언한다. 또한, 헤더파일과 c언어 코드 구동에 필요한 변수를 선언하고, 함수들을 전방선언한다. 'y.tab.h'의 경우 프로그래머가 만든 Yacc 토큰 번호와 기타 변수의 정보를 가지고있다. 또한 %start로 start symbol을 정의한다. % 와 % 로 코드의 블록이 형성되며 lexical analyzer에서 리턴되는 토큰들을 %token으로 정의한다.
규칙절
문법의 규칙을 정하는 부분이다. BNF와 유사한 방식으로 문법 규칙을 서술한다. 단말기호(Terminal symbol)은 lexical analyzer가 입력으로부터 리턴한 기호로 규칙의 왼쪽에 올수 없고 보통 대문자이다. 비단말기호(non-terminal symbol)은 문법 규칙의 이름을 부여하기 위한 기호로 보통 소문자이다.
서브루틴절
c언어 코드 부분으로 보통 main 함수 내에서 lex의 input file인 yyin으로부터 토큰을 읽어들여서 정해진 규칙에 따라서 파싱하는 구문 분석기 함수인 yyparse()를 호출한다. 또한, yyparse()에서 파싱 에러가 발생하면 서브루틴절에 정의된 yyerror()

함수를 호출한다. 따라서, yyerror() 함수를 작성함으로서 프로그래머의 역량에 따라 추가적인 오류 처리가 가능하다.

1.3 YACC의 동작 방식

Yacc는 stack과 state machine을 기반으로 구동된다. 우선 yacc에서 stack은 파싱 트리를 구축하는데 사용되고 구문 분석 중에는 임시 데이터를 저장하고 규칙을 적용하여 구문 요소를 구축하는 데 사용된다. state machine은 구문 분석가 동작하는데 필요하다. State machine은 상태와 상태간의 전이로 구성이 되는데 여기서 상태란 구문 규칙이 적용되는 것이다. 이 과정을 통해 state machine은 현재 상태와 입력 토큰에 따라 다음 동작을 결정한다. Yacc의 작동 과정은 우선 yacc 파일에서 문법을 작성하는 것으로 시작되고 여기서 작성된 문법으로 구문을 정의하고 파서를 생성한다. 이후 입력 토큰과 현재 상태에 따라 다음 동작을 결정하는 규칙의 집합인 구문 분석 테이블을 생성한다. 그 다음 구문 분석기 코드를 생성하는데 구문 분석 테이블을 기반으로 입력받은 데이터를 분석하고 stack을 활용해서 파싱트리를 구축한다. 구문 분석 과정은 우선 구문 분석기가 입력받은 데이터에서 토큰을 읽고 현재 상태와 입력 토큰으로 구문 분석 테이블을 기반으로 다음 동작을 진행한다. 이 과정에서 stack을 조작하거나 상태를 전이시키고 오류를 처리한다. 여기서 stack을 조작할때 stack에 토큰이나 구문 규칙을 저장한다. 또한, 구문 규칙을 적용할 때 구문 분석기는 현재 입력 토큰과 stack에 저장된 내용을 비교해서 적용 가능한 구문 규칙을 찾는다. 이 과정에서 적합한 토큰들이 스택에서 pop 되고, stack에는 구문 규칙의 왼쪽 항목이 들어간다. 구체적으로 구문 규칙이 완성되지 않는 토큰을 stack에 넣고 현재 읽은 stack에 따라 다음 상태로 전이하는 것을 shift라고 한다. 또, 구문 규칙의 오른쪽이 완성된 경우 parser가 오른쪽 기호들을 stack에서 꺼내서 좌변의 기호들을 stack에 저장하고 stack에 저장된 기호들에 따라 다시 새로운 상태로 전이하는 것을 reduce라고 한다. 위의 과정을 통해서 syntax error가 발생하거나 데이터를 모두 읽으면 파싱이 종료되고 결과를 리턴한다.

2 코드

2.1 Lex 코드

```
    %{//필요한 함수를 전방선언하고 출력을 위한 header를 작성하였다.
#include <stdio.h>
#include "y.tab.h"
void comment();//주석을 처리하기 위한 함수
%}

//패턴을 단순화하기 위해서 변수를 선언하였다.
D [0-9] // 0-9사이의 숫자 하나를 D로 나타낼 수 있다.
L [a-zA-Z_] // 영어 대소문자 한 글자와 언더바(_)를 L로 나타낼 수 있다.
H [a-fA-F0-9] // 16진수를 H로 나타낸다.
E [Ee][+-]?{D}+ // Scientific notation을 숫자로 표현한다.
FS (f|F|l|L) // 부동소수점 표현을 FS로 나타낼 수 있다.
IS          (u|U|l|L)* // 부호없는수 표현을 IS로 나타낼 수 있다.

%%
"#include".*\n {;} //전처리문 제거를 위한 코드
"#define".*\n {;} //전처리문 제거를 위한 코드
"//".*\n { ; } //주석 제거를 위한 코드
"/*" { comment(); } //주석 제거를 위한 코드
//아래 키워드에 해당하는 토큰들은 미리 정의되어있다. 그 미리 정의된 토큰들
을 리턴하는 코드이다.
"auto" {  return(AUTO); }//"auto"라는 키워드를 입력받으면, Lexical Analyzer
가 AUTO라는 토큰을 리턴한다.
"break" {  return(BREAK); }
"case" {  return(CASE); }
"char" {  return(CHAR); }
"const" {  return(CONST); }
"continue" {  return(CONTINUE); }
"default" {  return(DEFAULT); }
"do" {  return(DO); }
"double" {  return(DOUBLE); }
"else" {  return(ELSE); }
"enum" {  return(ENUM); }
"extern" {  return(EXTERN); }
"float" {  return(FLOAT); }
"for" {  return(FOR); }
"goto" {  return(GOTO); }
"if" {  return(IF); }
"int" {  return(INT); }
"long" {  return(LONG); }
"register" {  return(REGISTER); }
"return" {  return(RETURN); }
```

```
"short" {  return(SHORT); }
"signed" {  return(SIGNED); }
"sizeof" {  return(SIZEOF); }
"static" {  return(STATIC); }
"struct" {  return(STRUCT); }
"switch" {  return(SWITCH); }
"typedef" {  return(TYPEDEF); }
"union" {  return(UNION); }
"unsigned" {  return(UNSIGNED); }
"void" {  return(VOID); }
"volatile" {  return(VOLATILE); }
"while" {  return(WHILE); }

//IDENTIFIER 처리부이다. 영어 한글자 뒤에 영문자나 숫자가 0개 이상 붙는 형
태들은 IDENTIFIER를 리턴한다.
{L}({L}|{D})* {  return(IDENTIFIER); }

//CONSTANT 처리부이다.
0[xX]{H}+{IS}? {  return(CONSTANT); }//16진수와 IS가 붙으면 상수로 간
주하고 CONSTANT를 리턴한다.
0{D}+{IS}? {  return(CONSTANT); }//8진수와 IS가 붙으면 상수로 간주하고 CONSTANT
를 리턴한다.
{D}+{IS}? {  return(CONSTANT); }10진수와 IS가 붙으면 상수로 간주하고 CONSTANT
를 리턴한다.
L?'(\\.|[^\\'])+'{  return(CONSTANT); }리터럴 상수는 CONSTANT를 리턴
한다.

//Scientific notation은 CONSTANT를 반환한다.
{D}+{E}{FS}? {  return(CONSTANT); }
{D}*"."{D}+({E})?{FS}? {  return(CONSTANT); }
{D}+"."{D}*({E})?{FS}? {  return(CONSTANT); }

//" "로 묶인 내용은 문자열로 간주하고 STRING_LITERAL을 리턴한다.
L?\"(\\.|[^\\"])*\" {  return(STRING_LITERAL); }

//두 글자 이상인 키워드들을 처리하는 부분이다.
"..." {  return(ELLIPSIS); }
">>=" {  return(RIGHT_ASSIGN); }
"<<=" {  return(LEFT_ASSIGN); }
"+=" {  return(ADD_ASSIGN); }
"-=" {  return(SUB_ASSIGN); }
"*=" {  return(MUL_ASSIGN); }
"/=" {  return(DIV_ASSIGN); }
"%=" {  return(MOD_ASSIGN); }
"&=" {  return(AND_ASSIGN); }
"^=" {  return(XOR_ASSIGN); }
```

```
"|=" {  return(OR_ASSIGN); }
">>" {  return(RIGHT_OP); }
"<<" {  return(LEFT_OP); }
"++" {  return(INC_OP); }
"--" {  return(DEC_OP); }
"->" {  return(PTR_OP); }
"&&" {  return(AND_OP); }
"||" {  return(OR_OP); }
"<=" {  return(LE_OP); }
">=" {  return(GE_OP); }
"==" {  return(EQ_OP); }
"!=" {  return(NE_OP); }

//한 글자인 키워드를 처리하는 부분이다. 한 글자는 따로 토큰을 생성하 필요 없
이 바로 전달하면 된다.
";" {  return(';'); }
("{"|"<%") {  return('{'); }
("}"|"%>") {  return('}'); }
"," {  return(','); }
":" {  return(':'); }
"=" {  return('='); }
"(" {  return('('); }
")" {  return(')'); }
("["|"<:") {  return('['); }
("]"|":>") {  return(']'); }
"." {  return('.'); }
"&" {  return('&'); }
"!" {  return('!'); }
"~" {  return('~'); }
"-" {  return('-'); }
"+" {  return('+'); }
"*" {  return('*'); }
"/" {  return('/'); }
"%" {  return('%'); }
"<" {  return('<'); }
">" {  return('>'); }
"^" {  return('^'); }
"|" {  return('|'); }
"?" {  return('?'); }

[ \t\v\n\f] { ; }
. { /* 나머지 잔잔바리들을 처리하는 부분이다. */; }

%%
//yywrap()은 파일이나 입력 끝에 호출되어서 1을 반환하면 파싱(구문 해석)
을 정지한다.
```

```
int yywrap()
{
return 1;
}
//주석을 처리하기 위한 함수이다. //를 활용하는 주석문의 경우 한 줄이지만 /* */
를 활용하는 주석문의 경우
//한 줄 이상 작성할 수 있다. 따라서 한 줄 이상 작성되는 경우를 처리하기 위
해 정의한 함수이다.
// */ 을 만나거나 NULL문자를 만나면 함수가 종료된다.
void comment()
{
char c, c1;
loop:
while ((c = input()) != '*' && c != 0) ;
if ((c1 = input()) != '/' && c != 0){
unput(c1);
goto loop;
}
}
```

2.2 YACC 코드

```
    %{
#include <stdio.h>
int yylex();
//제시된 요구사항의 항목들을 카운트하기 위한 정수형 변수들
int functioncnt = 0;//함수정의, 함수사용,printf 와 같은 내장함수 사용횟
수 카운팅.
int operatorcnt = 0;//연산자의 개수
int intcnt = 0;//정수형 변수의 개수
int charcnt = 0;//char형 변수의 개수
int pointercnt = 0;//포인터로 선언된 변수의 개수
int arraycnt = 0;//배열의 개수
int arraycheck = 0;//이차원 배열을 배열 2개로 잘못 해석하지 않도록 하는 변
수
int selectioncnt = 0;//선택문의 개수
int loopcnt = 0;//반복문의 개수
int returncnt = 0;//리턴문의 개수
//int형이나 char형의 경우 int a,b;와 같이 선언되는 경우도 카운트하기 위
한 변수 선언
int intcheck = 0;//만약 IDENTIFIER가 정수면 1로 변경
int charcheck = 0;//만약 IDENTIFIER가 char형 이면 1로 변경
%}
//// Lexical Analyzer에 미리 정의되어 있는 Token들이다.

%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 ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

//translation_unit이라는 non-terminal에서 구문 해석이 시작된다.
%start translation_unit
%%

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

postfix_expression
: primary_expression
| postfix_expression '[' expression ']'
// 매개변수가 없는 함수가 선언될 경우를 카운트해야한다. functioncnt를 1 증
가시킨다.
| postfix_expression '(' ')'{functioncnt++;}
// 매개변수가 있는 함수가 선언되면 카운트를 해야한다. functioncnt를 1 증
가시킨다.
| postfix_expression '(' argument_expression_list ')'{functioncnt++;}
//참조 연산자이므로 operatorcnt를 1 증가시킨다.
| postfix_expression '.' IDENTIFIER {operatorcnt++;}
//참조 연산자이므로 operatorcnt를 1 증가시킨다.
| postfix_expression PTR_OP IDENTIFIER {operatorcnt++;}
//++가 후위연산자로 사용되는 경우이므로 operatorcnt를 1 증가시킨다.
| postfix_expression INC_OP {operatorcnt++;}
//--가 후위연산자로 사용되는 경우이므로 operatorcnt를 1 증가시킨다.
| postfix_expression DEC_OP {operatorcnt++;}
;

argument_expression_list
: assignment_expression
| argument_expression_list ',' assignment_expression
;

unary_expression
```

```
: postfix_expression
//++가 전위연산자로 사용되는 경우이므로 operatorcnt를 1 증가시킨다.
| INC_OP unary_expression {operatorcnt++;}
//--가 전위연산자로 사용되는 경우이므로 operatorcnt를 1 증가시킨다.
| DEC_OP unary_expression {operatorcnt++;}
| unary_operator cast_expression
| SIZEOF unary_expression
| SIZEOF '(' type_name ')'
;

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

cast_expression
: unary_expression
//형변환 연산자를 사용시 operatorcnt를 1 증가시킨다.
| '(' type_name ')' cast_expression {operatorcnt++;}
;

multiplicative_expression
: cast_expression
//산술연산자를 사용시 operatorcnt를 1 증가시킨다.
| multiplicative_expression '*' cast_expression {operatorcnt++;}
| multiplicative_expression '/' cast_expression {operatorcnt++;}
| multiplicative_expression '%' cast_expression {operatorcnt++;}
;

additive_expression
: multiplicative_expression
//산술연산자를 사용시 operatorcnt를 1 증가시킨다.
| additive_expression '+' multiplicative_expression {operatorcnt++;}
| additive_expression '-' multiplicative_expression {operatorcnt++;}
;

shift_expression
: additive_expression
//비트연산자 사용시 operatorcnt를 1 증가시킨다.
| shift_expression LEFT_OP additive_expression {operatorcnt++;}
| shift_expression RIGHT_OP additive_expression {operatorcnt++;}
;
```

```
relational_expression
: shift_expression
//논리연산자 사용시 operatorcnt를 1 증가시킨다.
| relational_expression '<' shift_expression {operatorcnt++;}
| relational_expression '>' shift_expression {operatorcnt++;}
| relational_expression LE_OP shift_expression {operatorcnt++;}
| relational_expression GE_OP shift_expression {operatorcnt++;}
;

equality_expression
: relational_expression
//논리연산자를 사용시 operatorcnt를 1 증가시킨다.
| equality_expression EQ_OP relational_expression {operatorcnt++;}
| equality_expression NE_OP relational_expression {operatorcnt++;}
;

and_expression
: equality_expression
//비트연산자를 사용시 operatorcnt를 1 증가시킨다.
| and_expression '&' equality_expression {operatorcnt++;}
;

exclusive_or_expression
: and_expression
//비트연산자를 사용시 operatorcnt를 1 증가시킨다.
| exclusive_or_expression '^' and_expression {operatorcnt++;}
;

inclusive_or_expression
: exclusive_or_expression
//비트연산자를 사용시 operatorcnt를 1 증가시킨다.
| inclusive_or_expression '|' exclusive_or_expression {operatorcnt++;}
;

logical_and_expression
: inclusive_or_expression
//관계연산자를 사용시 operatorcnt를 1 증가시킨다.
| logical_and_expression AND_OP inclusive_or_expression {operatorcnt++;}
;

logical_or_expression
: logical_and_expression
//관계연산자를 사용시 operatorcnt를 1 증가시킨다.
| logical_or_expression OR_OP logical_and_expression {operatorcnt++;}
;
```

```
conditional_expression
: logical_or_expression
| logical_or_expression '?' expression ':' conditional_expression
;

assignment_expression
: conditional_expression
//대입연산자를 사용시 operatorcnt를 1 증가시킨다.
| unary_expression assignment_operator assignment_expression {operatorcnt++;}
;

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

expression
: assignment_expression
| expression ',' assignment_expression
;

constant_expression
: conditional_expression
;

declaration
: declaration_specifiers ';'
//type_specifier에서 만약 변수형이 int형이면 intcheck = 1로, char형이
면 charcheck = 1로 변경해서
//int형 변수와 char형 변수의 개수를 카운트했다.
| declaration_specifiers init_declarator_list ';' {
if(intcheck == 1){
intcnt++;
}
if(charcheck == 1){
charcnt++;
```

```
}
intcheck = 0;
charcheck = 0;
}
;

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

init_declarator_list
: init_declarator
//type_specifier에서 만약 변수형이 int형이면 intcheck = 1로, char형이
면 charcheck = 1로 변경해서
//int형 변수와 char형 변수의 개수를 카운트했다.
| init_declarator_list ',' init_declarator{
if(intcheck == 1){
intcnt++;
}
if(charcheck == 1){
charcnt++;
}
}
;

init_declarator
: declarator
//대입연산자를 사용시 operatorcnt를 1 증가시킨다.
| declarator '=' initializer {operatorcnt++;}
;

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

type_specifier
: VOID
```

```
//char형이 나왔으므로 charcheck = 1로 변경해서 char형임을 나타낸다.
| CHAR {charcheck = 1;}
| SHORT
//int형이 나왔으므로 intcheck = 1로 변경해서 int형임을 나타낸다.
| INT {intcheck = 1;}
| LONG
| FLOAT
| DOUBLE
| SIGNED
| UNSIGNED
| struct_or_union_specifier
| enum_specifier
| TYPE_NAME
;

//아래의 코드들로 struct에 대한 처리가 가능하다.
struct_or_union_specifier
: struct_or_union IDENTIFIER '{' struct_declaration_list '}'
| struct_or_union '{' struct_declaration_list '}'
| struct_or_union IDENTIFIER
;

struct_or_union
: STRUCT
| UNION
;

struct_declaration_list
: struct_declaration
| struct_declaration_list struct_declaration
;

struct_declaration
: specifier_qualifier_list struct_declarator_list ';'
;

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

struct_declarator_list
: struct_declarator
| struct_declarator_list ',' struct_declarator
```

```
;

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 {operatorcnt++;}
;

type_qualifier
: CONST
| VOLATILE
;

declarator
: pointer direct_declarator{
//이차원 배열을 서로 다른 배열 두개로 오해하지 않도록 arraycheck로 이미 카
운트한
//배열을 표시해서 더 이상 카운트하지 않는다. 배열이 아니라면 arraycheck = 0
이지만
//배열이라면 arraycheck > 0이라서 arraycheck != 0이므로 배열의 개수가 1 증
가한다.
if(arraycheck != 0) {
arraycnt++;
arraycheck=0;
}
}
| direct_declarator{
//이차원 배열을 서로 다른 배열 두개로 오해하지 않도록 arraycheck로 이미 카
운트한
//배열을 표시해서 더 이상 카운트하지 않는다. 배열이 아니라면 arraycheck = 0
```

```
이지만
//배열이라면 arraycheck > 0이라서 arraycheck != 0이므로 배열의 개수가 1 증
가한다.
if(arraycheck != 0) {
arraycnt++;
arraycheck=0;
}
}
;

direct_declarator
: IDENTIFIER
| '(' declarator ')'
//배열을 나타내는 []이 나오면 arraycheck를 1 증가시킨다.
| direct_declarator '[' constant_expression ']'{arraycheck++;}
//배열을 나타내는 []이 나오면 arraycheck를 1 증가시킨다.
| direct_declarator '[' ']'{arraycheck++;}
//새로 parameter_list에서 선언되는 int형, char형 변수를 카운트해주기 위
해서 intcheck = 0, charcheck = 0으로 초기화해준다.
| direct_declarator '(' parameter_type_list ')' {
intcheck = 0;
charcheck = 0;
}
| direct_declarator '(' identifier_list ')'
| direct_declarator '(' ')'
;

pointer
//pointer에 해당하는 부분이 나오면 pointercnt를 1 증가시키게했다. 여기서 pointercnt
를 통해 포인터의 개수를 카운트하므로
//이후에 포인터의 개수를 추가적으로 카운트할 필요가 없다.
: '*'{pointercnt++;}
| '*' type_qualifier_list {pointercnt++;}
| '*' pointer {pointercnt++;}
| '*' type_qualifier_list pointer {pointercnt++;}
;

type_qualifier_list
: type_qualifier
| type_qualifier_list type_qualifier
;


parameter_type_list
: parameter_list
| parameter_list ',' ELLIPSIS
```

```
;

parameter_list
: parameter_declaration
| parameter_list ',' parameter_declaration
;

parameter_declaration
//매개변수의 자료형이 int형이나  char형이면
//각각 intcnt와 charcnt를 증가시킨다.
: declaration_specifiers declarator {
if(intcheck == 1){
intcnt++;
}
if(charcheck == 1){
charcnt++;
}
}
//매개변수의 자료형이 int형이나  char형이면
//각각 intcnt와 charcnt를 증가시킨다.
| declaration_specifiers abstract_declarator {
if(intcheck == 1){
intcnt++;
}
if(charcheck == 1){
charcnt++;
}
}
| declaration_specifiers
;

identifier_list
: IDENTIFIER
| identifier_list ',' IDENTIFIER
;

type_name
: specifier_qualifier_list
| specifier_qualifier_list abstract_declarator
;

abstract_declarator
: pointer
| direct_abstract_declarator
| pointer direct_abstract_declarator
;
```

```
direct_abstract_declarator
: '(' abstract_declarator ')'
| '[' ']'
| '[' constant_expression ']'
| direct_abstract_declarator '[' ']'
| direct_abstract_declarator '[' constant_expression ']'
| '(' ')'
| '(' parameter_type_list ')'
| direct_abstract_declarator '(' ')'
//새로 parameter_list에서 선언되는 int형, char형 변수를 카운트해주기 위
해서 intcheck = 0, charcheck = 0으로 초기화해준다.
| direct_abstract_declarator '(' parameter_type_list ')'{
intcheck = 0;
charcheck = 0;
}
;

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

initializer_list
: initializer
| initializer_list ',' initializer
;

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

labeled_statement
: IDENTIFIER ':' statement
| CASE constant_expression ':' statement
| DEFAULT ':' statement
;

compound_statement
: '{' '}'
```

```
| '{' statement_list '}'
| '{' declaration_list '}'
| '{' declaration_list statement_list '}'
;

declaration_list
: declaration
| declaration_list declaration
;

statement_list
: statement
| statement_list statement
//기존의 변수의 전방 선언만 허용하는 문법을 변수 선언 위치를 자유롭게 하기 위
해 statement_list와
//statement사이에 declaration_list를 넣었다. 이를 통해 i = 0; 이후에 int j = 10;
과 같이
//변수가 전방선언이 필수가 아니라 코드 진행 중간에 선언될 수 있게 만들었다.

| statement_list declaration_list statement
;

expression_statement
: ';'
| expression ';'
;

selection_statement
//선택문이 나오면 selectioncnt를 1 증가시킨다.
: IF '(' expression ')' statement {selectioncnt++;}
| SWITCH '(' expression ')' statement {selectioncnt++;}
;

iteration_statement
//반복문이 나오면 loopcnt를 1 증가시킨다.
: WHILE '(' expression ')' statement {loopcnt++;}
| DO statement WHILE '(' expression ')' ';'{loopcnt++;}
| FOR '(' expression_statement expression_statement ')' statement {loopcnt++;}
| FOR '(' expression_statement expression_statement expression ')' statement
{loopcnt++;}
;

jump_statement
//리턴문이 나오면 returncnt를 1 증가시킨다.
: GOTO IDENTIFIER ';'
| CONTINUE ';'
```

```
| BREAK ';'
| RETURN ';'{returncnt++;}
| RETURN expression ';'{returncnt++;}
;

translation_unit
: external_declaration
| translation_unit external_declaration
;

external_declaration
: function_definition
| declaration
;

function_definition
//함수가 만들어지는 경우 functioncnt를 1 증가시킨다. 새로운 함수가 만들
어지는 경우 int형과 char형이 존재하는지 다시 파악하고
//카운트해야하므로 intcheck과 charcheck을 0으로 초기화한다.
: declaration_specifiers declarator declaration_list compound_statement
{
functioncnt++;
intcheck = 0;
charcheck = 0;
}
| declaration_specifiers declarator compound_statement {
functioncnt++;
intcheck = 0;
charcheck = 0;
}
| declarator declaration_list compound_statement {
functioncnt++;
intcheck = 0;
charcheck = 0;
}
| declarator compound_statement {
functioncnt++;
intcheck = 0;
charcheck = 0;
}
;

%%

int main(void){
//yyparse()함수를 실행시켜 yylex()를 통해 토큰을 받아들이고 분석한다.
```

```
//yyparse() 함수의 실행이 끝나면 변수의 값을 출력한다.
yyparse();
printf("function = %d\n", functioncnt);
printf("operator = %d\n", operatorcnt);
printf("int = %d\n", intcnt);
printf("char = %d\n", charcnt);
printf("pointer = %d\n", pointercnt);
printf("array = %d\n", arraycnt);
printf("selection = %d\n", selectioncnt);
printf("loop = %d\n", loopcnt);
printf("return = %d\n", returncnt);

return 0;
}

void yyerror(const char *str){
fprintf(stderr, "error: %s\n",str);
}
```