Yacc Report

B711134 이승환

April 2021

1 Yacc의 작동방식

먼저 Yacc의 작동방식이다. 이전의 Lex가 정규표현식을 활용하여 어휘 단위로 token을 생성하는 어휘분석기 였다면 Yacc는 BNF(Left Recursion)를 규칙으로 갖는 Parser를 생성하는 즉, 문법규칙에 따라 구문을 분석하는 구문 분석기이다. Yacc는 Lex와 함께 연동되어 사용되며 Lex에서 생성한 token들을 일련의 규칙에 따라서 구문에 따라 분석을 실시한다. Yacc는 자체적으로 생성하는 yyparse()를 호출하여 yylex()또한 함께 호출되는데 이때 Yacc 구문분석기가 요구하는 token 들을 yylex()가 반환하면서 Yacc는 이 반환된 토큰을 토대로 사용자 및 BNF문법 규칙, 구문에 부합하는지 Parsing을 통해 확인하게 된다.

Yacc 파일을 compile(yacc -d)하게 되면 y.tab.h와 y.tab.c를 생성하게 된다. 이때 y.tab.h에는 Yacc에서 정의돈 token들을 가져오게 되고 Lex는 이 y.tab.h를 정의절에서 include하게 되어 어휘분석기가 해당 토큰을 return하게 된다.

이러한 과정으로 Lex와 Yacc가 통합되어 실행되어지며 정리하여 보면 input이 들어왔을때 Lex의 yylex()가 정규표현식에 따라 어휘를 스캔하고 y.tab.h 에서의 지정된 토큰에 맞춰 토큰을 return하여 yyparse()에서 이 토큰들을 이용하여 지정된 규칙에 따라 구문을 분석하는 과정을 갖는다.

2 Yacc의 구조

Yacc의 구조에는 정의절 규칙절 서브루틴절 3개의 절로 구성된다. 이는 Lex와 동일한 부분이다.

먼저 정의절에서는 문법에서 사용할 토큰의 선언, 구문분석기에서 사용할 변수 선언 및 정의, start state 정의, 타입등을 정의한다. 정의절의 앞 부분에서는 일반적으로 프로그램에 사용할 변수등이 선언, 정의된다.

```
%{
    #include <stdio.h>
    int yylex();
    int check = 1;
}
```

정의절의 뒷 부분에서는 토큰, 시작상태, 타입 등이 정의되는데, 여기서 토큰이 야크가 받아들일거라고 정의하는것들을 의미한다. 시작상태(start_state) 는 분석되는 구문의 파싱트리에서 루트노드를 symbol로서 지정하는 것이다. 이때 Symbol이란 이 후에도 기술되겠지만 Lex에서 반환된 토큰 및 사용자 지정 symbol들이 있는데 Yacc의 작동 과정에서 상태 전이, reduce를 해나갈 때 기준이 되는 이름이다.

```
    %start start_state
    %token IDENTIFIER
    %type <double> expression
%%
```

위처럼 정의절 선언이 끝나면 %%를 활용하여 규칙절로 넘어가기전 경계를 지어줘야한다

두번째로 야크의 규칙절은 symbol과 token들의 규칙으로 이루어지는데 좌변은 Non-terminal Symbol들만 올 수 있고 우변은번째로 야크의 규칙절은 symbol과 token들의 규칙으로 이루어지는데 좌변은 Non-terminal Symbol들만 올 수 있고 우변은 Lex에서 받아들여주는 토큰 또는 정의한 Terminal-Symbol들이 올 수 있다. 구조는 좌변 : 우변 Action 형식으로 이루어지는데 읽혀진 토큰들이 우변의 규칙을 만족하게되면 좌변으로 변경되는 과정을 거친다. 이때 이 과정이 이루어지면 braket안의 Action들이 실행되게 된다. 또한 이후 선언된 규칙이 끝나면 항상 ;(세미콜론)을 활용하여 끝내야한다.

```
    declaration_list : declaration_list ',' declaration
    | declaration
    ;
```

이후 규칙절이 끝나면 다시한번 %%로 서브루틴절과 경계를 둔다.

세번째로 서브루틴절은 Lex와 비슷하게 사용자의번째로 서브루틴절은 Lex와 비슷하게 사용자의 필요에 의한 함수가 있으면 이 부분에 정의한다.

```
int main(void)
    {
    printf("first int count is %d\n\n", intcount);
    yyparse();
    printf("now int count is %d\n", intcount);
    return 0;
}
void yyerror(const char *str)
    {
        fprintf(stderr, "error : %s\n", str);
}
```

이때, yyerror()는 yyparse()의 Pasrsing 과정에서 오류가 발생하면 yyerror()함수를 통해 오류메세지가 출력되는데 여기서 주된 오류메세지로는 "syntax error"가 있다.

3 Yacc의 작동원리

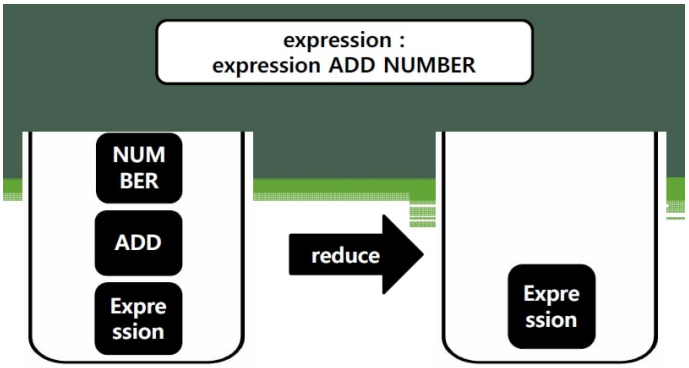

Figure 1: Yacc 작동원리

Yacc는 내부적으로 Stack과 State Machine을 기반으로 작동을 한다. Yacc가 token들을 불러들여 규칙절의 규칙에 따라 Parsing을 할 때, 예를 들어 1 + 1이라는 input값이 주어진다 했을 때, input은 모두 Lex에 의해 token인 NUMBER ADD NUMBER 로 반환된다. 이때 Yacc는 반환된 토큰을 Stack을 이용해서 하나 쌓아올리며 저장하는데 이 과정을 Shift라고 표현한다. 또한 Yacc의 규칙절에 있는 규칙에 의해 Stack 안에 있는 값의 변화가 오는데 예시로 expression : NUMBER라는 규칙이 있으면 처음 반환되어 Stack에 저장되어있던 토큰 NUMBER는 Yacc에 의해 Stack에서 빠지고 expression이라는 Symbol이 추가되게 된다. 이 과정을 Reduce라 표현한다. 이 전에 기술 했던 좌변 : 우변 구조 또한 이 Reduce과정을 표현한 구조이다.
또한 Yacc는 State Machine이라는 기능을 갖고 있는데, Yacc에서의 State Machine은 각 토큰별 규칙의 작동에서 전이 상태를 정의하는 기능으로 각 token을 갖고 규칙에 따라 쌓인 Stack을 state로 나타내게 된다. 여기서 reduce/shift의 과정을 지켜볼 수 있다. 이때 적절하지 못한 token이나 정의되지않은 규칙등을 받게되면 Syntax Error로 이어지게 된다.

Yacc에는 여러 에러 메시지를 나타 낼 수 있는데 Shift/Reduce 라는 에러이다. state machine은 규칙에 따라 특정 상태의 전이를 계속하게 되는데 state machine이 모호한 상황에 맞닥드리게 되면 이 오류메시지를 표시한다. Shift/Reduce 경고는 단순하게 다음 전이 상태로 Shift를 해야할지 Reduce를 해야할지 모르는 상태일때 표시되게 된다.
또, 하나의 오류메세지로 Reduce/Reduce 가 있다. 이것 또한 같은 반환 토큰에 대하여 규칙절에서 두개 이상의 파싱트리가 존재할 때, Yacc가 이를 어디로 Reduce해야할지 몰라서 생기는 경고이다.

결론적으로 Yacc는 Stack과 State Machine을 이용하여 구문 Parsing을 하는데 이때 구문의 분석은 State Machine에서 reduce규칙으로 계속해서 상태전이가 이루어지다가 일전에 지정한 Start state에

도달하게되면 구문 분석을 종료한다론적으로 Yacc는 Stack과 State Machine을 이용하여 구문 Parsing을 하는데 이때 구문의 분석은 State Machine에서 reduce규칙으로 계속해서 상태전이가 이루어지다가 일전에 지정한 Start state에 도달하게되면 구문 분석을 종료하고 토큰을 받아들이는걸 중지한다.

4 Yacc 컴파일

```
$ lex ex_lex.l          //lex.yy.c 생성
$ yacc -d ex_yacc.y     //-d 옵션으로 y.tab.c, y.tab.h 생성
$ cc y.tab.c lex.yy.c -o yacc   //cc 컴파일러로 yacc라는 실행파일 생성
$ ./yacc < test.c       //input test.c 로 yacc실행
```

5 Yacc 과제 분석

이 장부터는 Yacc과제 파일(hw3.y)의 코드를 전부 하나하나 분석한다. 분석에 앞서 규칙은 강의록의 ANSI-C Grammar을 그대로 copy하여 사용하였으며 필요한 부분은 규칙절의 코드 변경으로 해결하였다. 또한 Lex파일 hw3.l 또한 ANSI-C Grammar 강의록 링크를 그대로 가져와 필요한 부분만 새로운 Regex를 추가하여 수정하였으므로 lex파일에 대해선 수정한 부분만 기술 하도록하겠다.

5.1 Yacc hw3.y

Yacc파일의 규칙절을 보자. 코드의 상단에 가까울 수록 파스트리의 끝 부분을 향하고 우선순위가 높다. 반면 코드 하단으로 내려갈 수록 루트 시작상태에 가까워진다 즉, 우선순위가 낮다. 모든 우변에서 좌변으로 넘어가는 reduce 규칙은 printf() 를 통하여 규칙의 Action파트에 기술해 놓았으므로 Counting과 관련된 코드와 새로추가 또는 변경한 규칙들에 대해서만 주석을 달아 표기하도록 하겠다.

```
    %{
#include <stdio.h>
int yylex();
int ary[9] = {0,0,0,0,0,0,0,0,0};
int check = 0;      //pointer 개수를 위한 변수
int int_check = 0;  //int의 유무를 따지기 위한 변수
int int_check2 = 0; //한 파스트리에서 int의 개수를 구하기 위한 변수
%}
// 토큰 정의
%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 ENDIF SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN
%start translation_unit         //non-terminal 시작상태를 정의 (Ansi-C기본값)
```

```
%%
primary_expression
: IDENTIFIER    {printf("IDENT -> primary\n");}
| CONSTANT  {printf("CONSTANT -> primary\n");}
| STRING_LITERAL    {printf("STRING_LITERAL -> primary\n");}
| '(' expression ')'    {printf("'(' expression ')' -> primary\n");}
;

// 식별자 상수 문자열상수 (수식) 이 primary_expression으로 Reduce된다


postfix_expression
: primary_expression    {printf("primary -> postfix\n");}
| postfix_expression '[' expression ']' {printf("postfix '[' expression ']' -> postfix\n");}

//함수 사용 counting ary[0]++
| postfix_expression '(' ')'    {ary[0]++;printf("postfix() -> postfix\n");}

//전달인자가 있는 함수의 사용 counting ary[0]++
| postfix_expression '(' argument_expression_list ')'
{ary[0]++;printf("postfix_expression '(' argument_expression_list ')'->postfix\n");}

//참조연산자 . 을 counting ,연산자의 개수를 파악하기 위해 printf문 추가
| postfix_expression '.' IDENTIFIER {ary[1]++;printf("########HERE:
%d",ary[1]);printf("postfix '.' IDENT->postfix\n");}

//참조연산자 ->을 counting
| postfix_expression PTR_OP IDENTIFIER  {ary[1]++;printf("########HERE:
%d",ary[1]);printf("postfix PTR_OP IDENT->postfix\n");}

//증감연산자 변수++ counting
| postfix_expression INC_OP {ary[1]++;printf("########HERE: %d",ary[1]);printf("postfix
INC_OP->postfix\n");}
```

```
//증감연산자 변수-- counting
| postfix_expression DEC_OP {ary[1]++;printf("########HERE: %d",ary[1]);printf("postfix
DEC_OP->postfix\n");}
;

argument_expression_list
: assignment_expression {printf("assignment -> argument_expression list\n");}
| argument_expression_list ',' assignment_expression
{printf("argument_expression_list ',' assignment_expression\n");}
;

//단항연산 expression
unary_expression
: postfix_expression    {printf("postfix -> unary_expression\n");}

//++전위 증감연산자 counting, 우선순위는 위에있는 후위증감연산자가 더 높음
| INC_OP unary_expression   {ary[1]++;printf("########HERE: %d",ary[1]);printf("INC_OP
unary->unary\n");}

//--전위 증감연산자 counting
| DEC_OP unary_expression   {ary[1]++;printf("########HERE: %d",ary[1]);printf("INC_OP
unary->unary\n");}
| unary_operator cast_expression    {printf("unary_operator cast_expression ->
unary\n");}
| SIZEOF unary_expression   {printf("SIZEOF unary -> unary\n");}
| SIZEOF '(' type_name ')'  {printf("SIZEOF (type_name) -> unary\n");}
;

//단항연산자
unary_operator
: '&'
| '*'
| '+'
| '-'
```

```
| '~'
| '!'
;

cast_expression
: unary_expression  {printf("unary -> cast_expression\n");}

//cast 연산자 구문 counting
| '(' type_name ')' cast_expression {ary[1]++;printf("########HERE:
%d",ary[1]);printf("'(' type_name ')' cast_expression -> cast_expression\n");}
;

multiplicative_expression
: cast_expression   {printf("cast_expression -> multiplicative\n");}

    //산술연산자 * counting
| multiplicative_expression '*' cast_expression {ary[1]++;printf("########HERE:
%d",ary[1]);printf("multiplicative * cast -> multiplicative\n");}

//산술연산자 / counting
| multiplicative_expression '/' cast_expression {ary[1]++;printf("########HERE:
%d",ary[1]);printf("multiplicative / cast -> multiplicative\n");}

//산술 연산자 % counting
| multiplicative_expression '%' cast_expression {ary[1]++;printf("########HERE:
%d",ary[1]);printf("multiplicative % cast -> multiplicative\n");}
;

additive_expression
    //left-recursion을 만족한 상태로 왼쪽부터 해석한다.
    //additive_expression이 파스트리 윗부분이므로 우선순위 *,/,%에 비해 낮음
: multiplicative_expression {printf("multiplicative -> addictive\n");}

//산술 연산자 + counting
```

```
| additive_expression '+' multiplicative_expression {ary[1]++;printf("########HERE:
%d",ary[1]);printf("addictive + multiplicative -> addictive\n");}

//산술 연산자 - counting
| additive_expression '-' multiplicative_expression {ary[1]++;printf("########HERE:
%d",ary[1]);printf("addictive - multiplicative -> addictive\n");}
;

shift_expression
: additive_expression   {printf("addictive -> shift_expression\n");}

//비트 연산자 << counting
| shift_expression LEFT_OP additive_expression  {ary[1]++;printf("########HERE:
%d",ary[1]);printf("shift_expression LEFT_OP additive -> shift\n");}

//비트 연산자 >> counting
| shift_expression RIGHT_OP additive_expression {ary[1]++;printf("########HERE:
%d",ary[1]);printf("shift_expression RIGHT_OP addictive -> shift\n");}
;

relational_expression
: shift_expression  {printf("shift -> relational\n");}

//논리 연산자 < counting
| relational_expression '<' shift_expression    {ary[1]++;printf("########HERE:
%d",ary[1]);printf("relational_expression '<' shift_expression ->
relational_expression\n");}

//논리 연산자 > counting
| relational_expression '>' shift_expression    {ary[1]++;printf("########HERE:
%d",ary[1]);printf("relational_expression '>' shift_expression ->
relational_expression\n");}

//논리 연산자 <= counting
```

```
| relational_expression LE_OP shift_expression  {ary[1]++;printf("########HERE:
%d",ary[1]);printf("relational LE_OP shift -> shift\n");}

//논리 연산자 >= counting
| relational_expression GE_OP shift_expression  {ary[1]++;printf("########HERE:
%d",ary[1]);printf("realtional GE_OP shift -> shift\n");}
;

equality_expression
: relational_expression {printf("relational -> equality\n");}

//논리 연산자 == counting
| equality_expression EQ_OP relational_expression   {ary[1]++;printf("########HERE:
%d",ary[1]);printf("equality EQ_OP relational -> equality\n");}

//논리 연산자 != counting
| equality_expression NE_OP relational_expression   {ary[1]++;printf("########HERE:
%d",ary[1]);printf("equality NE_OP relational -> equality\n");}
;

and_expression
: equality_expression   {printf("equality -> and\n");}

//비트 연산자 & counting
| and_expression '&' equality_expression    {ary[1]++;printf("########HERE:
%d",ary[1]);printf("and -> equality\n");}
;

exclusive_or_expression
: and_expression    {printf("and -> exclusive_or\n");}

//비트 연산자 ^ counting
| exclusive_or_expression '^' and_expression    {ary[1]++;printf("########HERE:
%d",ary[1]);printf("exclusive_or '^' and -> exclusive_or\n");}
```

```
;

inclusive_or_expression
: exclusive_or_expression   {printf("exclusive_or -> inclusive_or\n");}

//비트 연산자 | counting
| inclusive_or_expression '|' exclusive_or_expression   {ary[1]++;printf("########HERE:
%d",ary[1]);printf("inclusive_or '|' exclusive_or -> inclusive\n");}
;

logical_and_expression
: inclusive_or_expression   {printf("inclusive_or -> logical_and\n");}

//관계 연산자 && counting
| logical_and_expression AND_OP inclusive_or_expression {ary[1]++;printf("########HERE:
%d",ary[1]);printf("logical_and_expression AND_OP inclusive_or_expression ->
logical_and\n");}
;

logical_or_expression
: logical_and_expression    {printf("logical_and_expression ->
logical_or_expression\n");}

//관계 연산자 || counting
| logical_or_expression OR_OP logical_and_expression    {ary[1]++;printf("########HERE:
%d",ary[1]);printf("logical_or_expression OR_OP logical_and_expression ->
logical_or_expression\n");}
;

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

```
assignment_expression
: conditional_expression    {printf("conditional -> assignment\n");}

//Right-recursion 확인가능, 즉 Lex가 오른쪽부터 봄
| unary_expression assignment_operator assignment_expression
{printf("unary_expression assignment_operator assignment_expression -> assignment\n");}
;

//대입연산자 전부 counting
assignment_operator
: '='   {ary[1]++;printf("########HERE: %d",ary[1]);printf("'=' -> assignment\n");}
| MUL_ASSIGN    {ary[1]++;printf("########HERE: %d",ary[1]);printf("MUL_ASSIGN ->
assignment\n");}
| DIV_ASSIGN    {ary[1]++;printf("########HERE: %d",ary[1]);printf("DIV_ASSIGN ->
assignment\n");}
| MOD_ASSIGN    {ary[1]++;printf("########HERE: %d",ary[1]);printf("MOD_ASSIGN ->
assignment\n");}
| ADD_ASSIGN    {ary[1]++;printf("########HERE: %d",ary[1]);printf("ADD_ASSIGN ->
assignment\n");}
| SUB_ASSIGN    {ary[1]++;printf("########HERE: %d",ary[1]);printf("SUB_ASSIGN ->
assignment\n");}
| LEFT_ASSIGN   {ary[1]++;printf("########HERE: %d",ary[1]);printf("LEFT_ASSIGN ->
assignment\n");}
| RIGHT_ASSIGN  {ary[1]++;printf("########HERE: %d",ary[1]);printf("RIGHT_ASSIGN ->
assignment\n");}
| AND_ASSIGN    {ary[1]++;printf("########HERE: %d",ary[1]);printf("AND_ASSIGN ->
assignment\n");}
| XOR_ASSIGN    {ary[1]++;printf("########HERE: %d",ary[1]);printf("XOR_ASSIGN ->
assignment\n");}
| OR_ASSIGN {ary[1]++;printf("########HERE: %d",ary[1]);printf("OR_ASSIGN ->
assignment\n");}
;
```

```
expression
: assignment_expression {printf("assignment -> expression\n");}
| expression ',' assignment_expression  {printf("expression ',' assignment ->
expression\n");}
;

constant_expression
: conditional_expression    {printf("conditional -> constant_expression\n");}
;

//마지막 변수 선언 부분
declaration
: declaration_specifiers ';'    {printf("declaration_specifier ';' -> declaration\n");}

//int_check변수는 int operator가 존재할때 1로 assign됨 (default는 0)
//따라서 int operator가 존재하면 ary[2]에 int_check2만큼을 더해줌
//이때, int_check2에는 한 파싱트리에서 존재하는 int 변수 선언 횟수가 assign되어있다
//구문 예시 ex) int a,b =0; or int a,b;
//공통되는 마지막 parse tree 제일 상위 노드에서 value assign
//다음 parse tree를 위해이후 Action이 끝나기전에 int_check, int_check2
//둘다 0으로 초기화한다
//Char도 동일
| declaration_specifiers init_declarator_list ';'   {if(char_check == 1){ary[3] +=
char_check2;}char_check=0;char_check2=0;if(int_check==1)ary[2] +=
int_check2;int_check2=0;int_check=0;printf("declaration_specifiers init_declarator_list
';' -> declaration\n");}
;

declaration_specifiers
: storage_class_specifier   {printf("storage_class_specifier ->
declaration_specifier\n");}
| storage_class_specifier declaration_specifiers    {printf("storage_class_specifier
declaration_specifiers -> declaration_specifier\n");}
| type_specifier    {printf("type_specifier -> declaration_specifier\n");}
```

```
| type_specifier declaration_specifiers {printf("type_specifier declaration_specifiers
-> declaration_specifier\n");}
| type_qualifier    {printf("type_qualifier -> declaration_specifier\n");}
| type_qualifier declaration_specifiers {printf("type_qualifier declaration_specifiers
-> declaration_specifier\n");}
;

init_declarator_list
: init_declarator   {printf("init_declaration -> init_declaration_list\n");}
| init_declarator_list ',' init_declarator  {printf("init_declarator_list ','
init_declarator -> init_declaration_list\n");}
;

//변수 종류 두가지가 있는 규칙
init_declarator

    //해당 규칙까지 상태전이가 일어날때
    //int형 operator가 존재하여 int_check가 1로 assign되었을시
    //조건문에 따라 int_check2 값이 1씩 증가한다
    //구문 예시 ex) n
    //즉, 변수가 있으면 counting한다
    //Char count도 정확히 같은 방식이다.
: declarator    {if(char_check == 1){char_check2++;}if(int_check ==
1){int_check2++;}printf("declaratior -> init_declarator\n");}

//위와 동일 int_check = 1일때 즉, int operator를 앞에서 미리 parse했을때
//변수 = 초기화 형식
//구문 예시 ex) n = 0
//declarator '=' initializer 된 변수들의 개수도 count한다
//Char count도 정확히 같은 방식이다
| declarator '=' initializer    {ary[1]++;if(char_check==1){char_check2++;}if(int_check
== 1){int_check2++;}printf("declarator '=' initializer\n");}
;
```

```
storage_class_specifier
: TYPEDEF   {printf("TYPEDEF -> storage_class_specifier\n");}
| EXTERN    {printf("EXTERN -> storage_class_specifier\n");}
| STATIC    {printf("STATIC -> storage_class_specifier\n");}
| AUTO  {printf("AUTO -> storage_class_specifier\n");}
| REGISTER  {printf("REGISTER -> storage_class_specifier\n");}
;

type_specifier
: VOID  {printf("VOID -> type_specifier\n");}

// CHAR 토큰 발견 시 char_check = 1;
| CHAR  {char_check=1;printf("CHAR -> type_specifier\n");}
| SHORT {printf("SHORT -> type_specifier\n");}

// INT 토큰 발견 시 int_check =1;
| INT   {int_check=1;printf("@@@@@INT COME@@@@@");printf("INT -> type_specifier\n");}
| LONG  {printf("LONG -> type_specifier\n");}
| FLOAT {printf("FLOAT -> type_specifier\n");}
| DOUBLE    {printf("DOUBLE -> type_specifier\n");}
| SIGNED    {printf("SIGNED -> type_specifier\n");}
| UNSIGNED  {printf("UNSIGNED -> type_specifier\n");}
| struct_or_union_specifier {printf("struct_or_union_specifier -> type_specifier\n");}
| enum_specifier    {printf("enum_specifier -> type_specifier\n");}
| TYPE_NAME {printf("TYPE_NAME -> type_specifier\n");}
;

struct_or_union_specifier
: struct_or_union IDENTIFIER '{' struct_declaration_list '}'
{printf("struct_or_union IDENTIFIER '{' struct_declaration_list '}' ->
struct_or_union_specifier\n");}
| struct_or_union '{' struct_declaration_list '}'   {printf("struct_or_union '{'
struct_declaration_list '}' ->struct_or_union_specifier\n");}
| struct_or_union IDENTIFIER    {printf("struct_or_union IDENTIFIER
```

```
->struct_or_union_specifier\n");}
;

struct_or_union
: STRUCT    {printf("STRUCT -> struct_or_union\n");}
| UNION {printf("UNION -> struct_or_union\n");}
;

struct_declaration_list
: struct_declaration    {printf("struct_declaration_list -> struct_declaration\n");}
| struct_declaration_list struct_declaration    {printf("struct_declaration_list
struct_declaration -> struct_declaration\n");}
;

struct_declaration
: specifier_qualifier_list struct_declarator_list ';'   {printf("specifier_qualifier_list struct_d
struct_declaration\n");}
;

specifier_qualifier_list
: type_specifier specifier_qualifier_list   {printf("type_specifier
specifier_qualifier_list -> specifier_qualifier_list\n");}
| type_specifier    {printf("type_qualifier -> specifier_qualifier_list\n");}
| type_qualifier specifier_qualifier_list   {printf("type_qualifier
specifier_qualifier_list -> specifier_qualifier_list\n");}
| type_qualifier    {printf("type_qualifier -> specifier_qualifier_list\n");}
;

struct_declarator_list
: struct_declarator {printf("struct_declarator -> struct_declarator_list\n");}
| struct_declarator_list ',' struct_declarator  {printf("struct_declarator_list ','
struct_declarator -> struct_declarator_list\n");}
;
```

```
struct_declarator
: declarator    {printf("declarator -> struct_declarator\n");}
| ':' constant_expression   {printf("':' constant_expression -> struct_declarator\n");}
| declarator ':' constant_expression    {printf("declarator ':' constant_expression ->
struct_declarator\n");}
;

enum_specifier
: ENUM '{' enumerator_list '}'  {printf("ENUM '{' enumerator_list '}' ->
enum_specifier\n");}
| ENUM IDENTIFIER '{' enumerator_list '}'   {printf("ENUM IDENTIFIER '{'
enumerator_list '}' -> enum_specifier\n");}
| ENUM IDENTIFIER   {printf("ENUM IDENTIFIER -> enum_specifier\n");}
;

enumerator_list
: enumerator    {printf("enmerator -> enumerator_list\n");}
| enumerator_list ',' enumerator    {printf("enmerator_list ',' enumerator ->
enmuerator_list\n");}
;

enumerator
: IDENTIFIER    {printf("IDENTIFIER -> enmerator\n");}
| IDENTIFIER '=' constant_expression    {printf("IDENTIFIER '=' constatnt_expression ->
enmerator\n");}
;

type_qualifier
: CONST {printf("CONST -> type_qualifier\n");}
| VOLATILE  {printf("VOLATILE -> type_qualifier\n");}
;

declarator
```

```
    //pointer 포함 선언문 발견 시 ptr_check = 1;
    //pointer 규칙에 직접하지 않고 declarator 규칙에서 count를 하므로
    //한번의 count로 parameter, 변수 둘다 check 가능
    //또한 ** 반복 중복체크를 피할 수 있음
: pointer direct_declarator {ptr_check = 1;printf("pointer direct_declarator ->
declarator\n");}
| direct_declarator {printf("direct_declarator -> declarator\n");}
;

//선언자 변수 선언
direct_declarator

    //선언자 변수 선언에서 ptr_check = 1일시, 즉 pointer operator가
    //parse tree안에 있을 때 ary[4]++
: IDENTIFIER    {if(ptr_check==1)ary[4]++;check=0;printf("IDENTIFIER ->
direct_declarator\n");}
| '(' declarator ')'    {printf("'(' declarator ')' -> direct_declarator\n");}

//direct_declarator '[' constant_expression ']' 구문의 배열 발견 시
| direct_declarator '[' constant_expression ']' {ary[5]++;printf("direct_declarator '['
constant_expression ']' -> direct_declarator\n");}

//direct_declarator '[' ']' 구문의 배열 발견 시
| direct_declarator '[' ']' {ary[5]++;printf("direct_declarator '[' ']' ->
direct_declarator\n");}

//INT check시에 int형 함수로 인해 int_check = 1이 되었지만
//int형 함수는 count하지 않으므로 다른 구문에 영향을 막기위해
//int_check = 0 assign
| direct_declarator '(' parameter_type_list ')' {int_check =0;printf("direct_declarator
'(' parameter_type_list ')' -> direct_declarator\n");}
| direct_declarator '(' identifier_list ')' {printf("direct_declarator '('
identifier_list ')' -> direct_declarator\n");}
| direct_declarator '(' ')' {printf("direct_declarator '(' ')' ->
```

```
direct_declarator\n");}
;

pointer
: '*'   {printf("'*' -> pointer\n");}
| '*' type_qualifier_list   {printf("'*' type_qualifier_list -> pointer\n");}
| '*' pointer   {printf("'*' pointer -> pointer\n");}
| '*' type_qualifier_list pointer   {printf("'*' type_qualifier_list pointer ->
pointer\n");}
;

type_qualifier_list
: type_qualifier    {printf("type_qualifier -> type_qualifier_list\n");}
| type_qualifier_list type_qualifier    {printf("type_qualifier_list type_qualifier ->
type_qualifier_list\n");}
;


parameter_type_list
: parameter_list {printf("parameter_list -> parameter_type_list\n");}
| parameter_list ',' ELLIPSIS   {printf("parameter_list ',' ELLIPSIS ->
parameter_type_list\n");}
;

parameter_list
: parameter_declaration {printf("parameter_declaration -> parameter_list\n");}
| parameter_list ',' parameter_declaration  {printf("parameter_list ','
parameter_declaration -> parameter_list\n");}
;

parameter_declaration
```

    //parameter로 int형 변수를 받는 구문을 count
    //위와 같은 방법이지만 parameter_declaration으로 reduce되는것에 차이

```
    //마지막 int_check = 0 초기화   //마지막 int_check = 0 초기화
: declaration_specifiers declarator {if(int_check == 1)ary[2]++;printf("@@@@@@INT :
%d", ary[2]);int_check=0;printf("declaration_specifiers declarator ->
parameter_declaration\n");}
| declaration_specifiers abstract_declarator    {printf("declaration_specifiers
abstract_declarator -> parameter_declaration\n");}
| declaration_specifiers    {printf("declaration_specifiers ->
parameter_declaration\n");}
;

identifier_list
: IDENTIFIER    {printf("IDENTIFIER -> identifier_list\n");}
| identifier_list ',' IDENTIFIER    {printf("identifier_list ',' IDENTIFIER ->
identifier_list\n");}
;

type_name
: specifier_qualifier_list  {printf("specifier_qualifier_list -> identifier_list\n");}
| specifier_qualifier_list abstract_declarator  {printf("specifier_qualifier_list
abstract_declarator -> identifier_list\n");}
;

abstract_declarator
: pointer   {printf("pointer -> abstract_declarator\n");}
| direct_abstract_declarator    {printf("direct_abstract_declarator ->
abstract_declarator\n");}
| pointer direct_abstract_declarator    {printf("pointer direct_abstract_declarator ->
abstract_declarator");}
;

direct_abstract_declarator
: '(' abstract_declarator ')'   {printf("'(' abstract_declarator ')' ->
direct_abstract_declarator\n");}
| '[' ']'   {printf("'[' ']' -> direct_abstract_declarator\n");}
```

```
| '[' constant_expression ']'   {printf("'[' constant_expression ']' ->
direct_abstract_declarator\n");}
| direct_abstract_declarator '[' ']'    {printf("direct_abstract_declarator '[' ']' ->
direct_abstract_declarator\n");}
| direct_abstract_declarator '[' constant_expression ']'
{printf("direct_abstract_declarator '[' constant_expression ']' ->
direct_abstract_declarator\n");}
| '(' ')'   {printf("'(' ')' -> direct_abstract_declarator\n");}
| '(' parameter_type_list ')'   {printf("'(' parameter_type_list ')' ->
direct_abstract_declarator\n");}
| direct_abstract_declarator '(' ')'    {printf("direct_abstract_declarator '(' ')' ->
direct_abstract_declarator\n");}
| direct_abstract_declarator '(' parameter_type_list ')'
{printf("direct_abstract_declarator '(' parameter_type_list ')' ->
direct_abstract_declarator\n");}
;

initializer
: assignment_expression {printf("assignment_expression -> initializer\n");}
| '{' initializer_list '}'  {printf("'{' initializer_list '}' -> initializer\n");}
| '{' initializer_list ',' '}'  {printf("'{' initializer_list ',' '}' ->
initializer\n");}
;

initializer_list
: initializer   {printf("initializer -> initializier_list\n");}
| initializer_list ',' initializer  {printf("initializer_list ',' initializer ->
initializier_list\n");}
;

statement
: labeled_statement {printf("labeled_statement -> statement\n");}
| compound_statement    {printf("compound_statement -> statement\n");}
| expression_statement  {printf("expression_statement -> statement\n");}
```

```
| selection_statement   {printf("selection_statement -> statement\n");}
| iteration_statement   {printf("iteration_statement -> statement\n");}
| jump_statement    {printf("jump_statement -> statement\n");}
;

labeled_statement
: IDENTIFIER ':' statement  {printf("IDENTIFIER ':' statement -> labled_statement\n");}
| CASE constant_expression ':' statement    {printf("CASE constant_expression ':'
statement -> labled_statement\n");}
| DEFAULT ':' statement {printf("DEFAULT ':' statement -> labled_statement\n");}
;

compound_statement
: '{' '}'   {printf("'{' '}' -> compound_statement\n");}
| '{' statement_list '}'    {printf("'{' statement_list '}' -> compound_statement\n");}
| '{' declaration_list '}'  {printf("'{' declaration_list '}' ->
compound_statement\n");}
| '{' declaration_list statement_list '}'   {printf("'{' declaration_list
statement_list '}' -> compound_statement\n");}
;

declaration_list
: declaration   {printf("declaration -> declaration_list\n");}
| declaration_list declaration  {printf("declaration_list declaration ->
declaration_list\n");}
;

statement_list
: statement {printf("statement -> statement_list\n");}
| statement_list statement  {printf("statement_list statement -> statement_list\n");}
;

expression_statement
: ';'   {printf("';' -> expression_statement\n");}
```

```
| expression ';'    {printf("expression ';' -> expression_statement\n");}
;

//선택문
selection_statement

    //선택문 구문이므로 상태전이 시 count
: IF '(' expression ')' statement   {ary[6]++;printf("IF '(' expression ')' statement
-> selection_statement\n");}

//Else 구문은 따로 count하지 않으므로 아예 규칙에서 삭제

//선택문 구문이므로 상태전이 시 count
| SWITCH '(' expression ')' statement   {ary[6]++;printf("SWITCH '(' expression ')'
statement -> selection_statement\n");}
;

//반복문
iteration_statement

    //반복문 구문 규칙이므로 상태전이 시 count
: WHILE '(' expression ')' statement    {ary[7]++;printf("WHILE '(' expression ')'
statement -> iteration_statement\n");}

//반복문 구문 규칙이므로 상태전이 시 count
| DO statement WHILE '(' expression ')' ';' {ary[7]++;printf("DO statement WHILE '('
expression ')' ';' -> iteration_statement\n");}

//반복문 구문 규칙이므로 상태전이 시 count
| FOR '(' expression_statement expression_statement ')' statement
{ary[7]++;printf("FOR '(' expression_statement expression_statement ')' statement ->
iteration_statement\n");}

//반복문 구문 규칙이므로 상태전이 시 count
```

```
| FOR '(' expression_statement expression_statement expression ')' statement
{ary[7]++;printf("FOR '(' expression_statement expression_statement expression ')'
statement -> iteration_statement\n");}
;

jump_statement
: GOTO IDENTIFIER ';'   {printf("GOTO IDENTIFIER ';' -> jump_statement\n");}
| CONTINUE ';'  {printf("CONTINUE ';' -> jump_statement\n");}
| BREAK ';' {printf("BREAK ';' -> jump_statement\n");}

//RETURN 토큰 발견 시 count
| RETURN ';'    {ary[8]++;printf("RETURN ';' -> jump_statement\n");}

//RETURN 토큰 발견 시 count
| RETURN expression ';' {ary[8]++;printf("RETURN expression ';' ->
jump_statement\n");}
;

translation_unit
: external_declaration  {printf("external_declaration -> translation_unit\n");}
| translation_unit external_declaration {printf("translation_unit external_declaration
-> translation_unit\n");}
;

external_declaration
: function_definition   {printf("function_definition -> external_declaration\n");}
| declaration   {printf("declaration -> external_declaration\n");}
;

function_definition

    //함수 정의 상태를 나타내는 구문이므로 count
: declaration_specifiers declarator declaration_list compound_statement
{ary[0]++;printf("declaration_specifiers declarator declaration_list
```

```
compound_statement -> function_definition\n");}

| declaration_specifiers declarator compound_statement
{ary[0]++;printf("declaration_specifiers declarator compound_statement ->
function_definition\n");}

| declarator declaration_list compound_statement    {ary[0]++;printf("declarator
declaration_list compound_statement -> function_definition\n");}

| declarator compound_statement {ary[0]++;printf("declarator compound_statement ->
function_definition\n");}

//전처리문 처리 용 구문추가
//include ,define은 declarator이므로 토큰을 추가하였다
//Lex상에서 처리 할 수도 있음 Regex이용 토큰화
    | declarator '<' IDENTIFIER '.' IDENTIFIER '>'

    //Define전처리문 상에서 마지막이 상수인것 처리 구문
    | declarator IDENTIFIER CONSTANT
    ;
%%
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;
```

```
}

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

5.2 Lex hw3.l

Lex 파일은 위에도 기술하였듯 ANSI-C 강의록에 있는 링크에서 발췌하였으므로 수정이 이루어진 부분에 대해서만 주석처리로 기술하도록 하겠다.

```
    %{
#include <stdio.h>
#include "y.tab.h"      //Yacc -d 이후 생성파일
void count();
%}
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)*

%x DELETE
%%
"//".*\n ;       //주석문에 대해 regex를 이용하여
                   //주석문 발견 시 어떠한 Action도 취하지않음

"/*" BEGIN DELETE;       //%x시작상태 설정으로
<DELETE>. ;               //주석박스 안에 있는
<DELETE>\n ;               //모든 인식된 글자, 개행에 대해 아무것도하지않음
<DELETE>"*/" BEGIN INITIAL;  //Default로 상태 변환
```

```
"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" { count(); return(ELSE); }
"endif" { count(); return(ENDIF);}
"enum" { count(); return(ENUM); }
"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(); return(STRUCT); }
"switch" { count(); return(SWITCH); }
"typedef" { count(); return(TYPEDEF); }
"union" { count(); return(UNION); }
"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(); 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('|'); }
"?" { count(); return('?'); }

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

%%

yywrap()
```

```
{
return(1);
}


comment()
{
char c, c1;

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

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

if (c != 0)
putchar(c1);
}


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++;

}


int check_type()
{
/*
* pseudo code --- this is what it should check
*
* if (yytext == type_name)
* return(TYPE_NAME);
*
* return(IDENTIFIER);
*/

/*
* it actually will only return IDENTIFIER
*/

return(IDENTIFIER);
}
```