Lex & Yacc

Joeun Park

May 16th, 20

제 1 절 yacc에 대한 설명

1.1 yacc 개요

yacc는 입력값에 대해 원하는 것을 찾아내는 일과, 그 찾아낸 것들 간의 관계를 따지는 프로그램 작성을 도와준다. 입력값을 의미단위(token)로 나누는 것을 어휘분석(Lexical Analysis)이라 하며, 그런 일을 하는 것을 어휘분석기(Lexical Analyzer or Lexer scanner)라고 한다. 입력값이 의미단위로 나뉘게 되면 프로그램은 그것들간의 관계를 따지게 된다. 예를 들어 C 컴파일러는 토큰의 수식, 문장, 선언문, 블록, 프로시저, 등과 같은 여부를 판별해낸다. 이런 작업이 구문분석이고 그 관계를 정의하는 규칙을 문법(Grammar)이라고 한다.

yacc는 구문 분석기(Parser)를 생성해주는 도구이다. 구문 분석기는 어휘 분석기로부터 받은 의미단위를 문법에 맞는지 검사하는 일을 한다. 사용자가 정의하고자 하는 문법을 C 언어로 기술해 주면 yacc가 자동적으로 구문 분석기를 생성해낸다. 구문 분석기는 의미단위를 가져와서 구문을 분석하는 프로그램이므로 어휘 분석을 하는 프로그램은 Lex로 별도로 구현시켜 두 프로그램을 접목시킨다.

1.2 yacc 문법

Yacc의 grammar file은 Lex Specification과 비슷하다.

```
%{
C Definition Section
%}
```

```
Yacc Definition Section

%%
Grammar Rules Section
%%

User Subroutines
```

- C Definition Section
  이 곳에 쓰여진 내용은 y.tab.c에 그대로 복사된다. Lex에서와 마찬가지로 다른 부분에서 사용할 변수가 있다면 이 곳에서 미리 정의하고 헤더파일을 include 해야한다.
- Yacc Definition Section
  Grammar Rules Section에서 사용하는 토큰, 결합법칙, 변수 및 토큰들의 타입 등을 선언한다.
- Grammar Rules Section
  인식 하려는 문법과 그에 따른 C 언어 규칙을 정의한다. seperator는 :와 — 이 쓰이며 terminator로는 ;이 쓰인다.
  item User Subroutines
  사용자가 만들어서 사용해야 할 함수가 있으면 이곳에 정의한다. 예를 들어 main()함수나 yyerror()등과 같은 함수들이 있다.

1.3 yacc 동작 방식

Lex 혼자서도 어느정도 구문 분석을 하여 수행할 수 있다. Lex 에 의하여 만들어진 lexer 자신도 얼마든지 yacc 없이 특정한 문장을 읽어 우리가 원하는 동작을 구현할 수도 있다. lexer 에서 토큰이 하나하나 나올때 마다 특정한 행동을 수행하도록 하면 된다. 그러나 계산기와 같은 동작을 수행할 때 우리는 산술연산의 우선순위같은것을 정해 줘야 한다. 이런 c 코드를 만들기 위해서는 많은 노력이 필요하다. 즉 lexer에 의하여 나오는 토큰들을 수집하여, 수집된 토큰들을 나름대로 정리하여 원하는 계산 혹은 동작을 수행해야 한다. 이러기 위해서는 앞에서 언급했듯이 많은 노력이 필요로 하다. 이러한 노력을 줄이기 위하여 yacc를 사용하는 것이다.

yacc는 토큰을 계속 읽어서 사용자가 정의한 문법과 맞춰가며 구문 분석을 하도록 되어있다. 읽어들인 토큰이 문법을 온전히 만족하지 못하고 다른 토큰이 더 필요하다면 그 토큰을 스택에 쌓아 두는데 이것을 스택에 shift하며, 이를 shift라고 칭한다.

계속 토큰을 읽어서 스택에 있는 토큰들과 문법을 비교했을 때 문법을 만족하면 스택에 있던 토큰을 꺼내고(pop) 문법의 왼쪽(LHS) 심볼로 대치하는데, 이것을 reduce라 한다.

하지만 아무리 yacc가 구문분석을 하는데에 도움을 주는 툴이라 하더라도 하나의 파스 트리가 생성되지 못하는 모호한 문법 혹은 한 개를 초과하는 갯수의 토큰을 가져와야 분석을 할 수 있는 문법 등을 구문분석할 수는 없다.

1.4 Lex 코드 분석

```
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)*

%{
#include <stdio.h>
#include "y.tab.h"
void count();
%}

%%
"/*"                    { comment(); }
```

```
"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); }
"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('?'); }
"#"                     { count(); return('#'); }

[ \t\v\n\f]             { count(); }
.                       { count(); return('.'); }
%%


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++;

        //ECHO;
}

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);
}
```

제 2 절 Yacc

2.1 yacc 전체 코드

```
%{
#include <stdio.h>
int yylex();
int func_count = 0;
int exp_count = 0;
int int_count = 0;
int char_count = 0;
int pointer_count = 0;
int array_count = 0;
int select_count = 0;
int loop_count = 0;
int return_count = 0;
int num = 0;
int check_int = 0;
int check_char = 0;
int check_pointer = 0;
int check_array = 0;
%}

%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 SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%start start_state
%%

```

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

postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
        | postfix_expression '(' ')' {func_count++;}
        | postfix_expression '(' argument_expression_list ')' {func_count++;}
        | postfix_expression '.' IDENTIFIER
        | postfix_expression PTR_OP IDENTIFIER
        | postfix_expression INC_OP {exp_count++;}
        | postfix_expression DEC_OP {exp_count++;}
        ;

argument_expression_list
        : assignment_expression
        | argument_expression_list ',' assignment_expression
        ;

unary_expression
        : postfix_expression
        | INC_OP unary_expression {exp_count++;}
        | DEC_OP unary_expression {exp_count++;}
        | unary_operator cast_expression
        | SIZEOF unary_expression
        | SIZEOF '(' type_name ')'
        ;

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

```

```
cast_expression
        : unary_expression
        | '(' type_name ')' cast_expression
        ;

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

additive_expression
        : multiplicative_expression
        | additive_expression '+' multiplicative_expression {exp_count++;}
        | additive_expression '-' multiplicative_expression {exp_count++;}
        ;

shift_expression
        : additive_expression
        | shift_expression LEFT_OP additive_expression {exp_count++;}
        | shift_expression RIGHT_OP additive_expression {exp_count++;}
        ;

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

equality_expression
        : relational_expression
        | equality_expression EQ_OP relational_expression {exp_count++;}
        | equality_expression NE_OP relational_expression {exp_count++;}
        ;

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

exclusive_or_expression
        : and_expression
        | exclusive_or_expression '^' and_expression {exp_count++;}
        ;
```

2.2 yacc 요구사항과 미진한 부분 및 개선방안

1. 함수 카운팅
   전반적인 함수 카운팅은 구현하였으나, 포인터함수의 경우에서 함수로서 카운팅되지 않고 포인터변수로서 카운팅이 되며, 일반 함수 호출의 경우에 함수로서 카운팅이 되는 상황이다. 전자의 경우, checkpointer 변수와 checkfunc 변수가 둘 다 1을 만족하는 경우를 따져 이에 걸맞은 코드를 작성해야 했으나 아이디어가 부족했다. 후자의 경우, 함수가 안에 코드를 품고있지 않은채 바로 ; 으로 끝나는 경우에 countfunc 변수를 0으로 하면 되지 않았나 싶다.

2. 수식 카운팅
   연산, 비트연산, 비교, 포인터기호 등과 같은 operator 카운팅을 구현하였다.

3. int 변수 카운팅
   int 변수의 전반적인 카운팅은 구현하였으나, int main() 과 같은 int형 함수 혹은 int형 포인터함수 등의 경우는 int형 변수로 카운팅이 들어가지 않아야 하는데, 이러한 경우까지 고려하는 코드를 작성 하는 아이디어가 부족했다.
   이 부분을 개선하기 위해 함수가 카운팅 되는 구간에 intcount 변수를 0으로 만들거나 감소시켜야 한다고 생각한다.

4. char 변수 카운팅
   char 변수의 전반적인 카운팅은 구현하였으나, int 변수 카운팅의 상황과 마찬가지의 오류를 범할 수 있다.

5. pointer 변수 카운팅
   위에도 잠깐 언급하였듯이, 포인터함수의 경우에는 pointer 변수 카운팅이 들어가면 안 되는데, 이 오류를 바로잡는 데에 실패하였다.
   이 부분을 개선하기 위해서는 int *a() 와 같이 뒤에 '(', ')', '', ''이 오는 경우의 함수 조건에 pointercount 변수를 0으로 설정하거나 감소시켜야 한다고 생각한다.

6. 배열 변수 카운팅
   일차원 이상의 차원 배열 변수는 모두 배열변수가 1개인 경우이므로 이 조건을 잘 고려하여 구현하였다.

7. 선택문 카운팅
   선택문이 카운팅 되도록 잘 구현하였으며, 포함시키지 않아도 되는 조건의 경우는 제거하였다.

8. 반복문 카운팅
   반복문이 카운팅 되도록 잘 구현하였다.

9. 리턴문 카운팅
   리턴문이 카운팅 되도록 잘 구현하였다.

10. 주의사항
    include와 같은 헤더파일이나 define문을 갖는 c 코드에서 오류가 생기지 않도록 yacc 문법을 잘 수정하였다.
    테스트 케이스에 있을 수 있다고 언급된 타입들을 토큰화 시켰다.
    주의사항 6번부터 10번까지의 경우에 대해서는 혼자서 c 코드 경우를 구현하지 못했기 때문에 깊이 생각해보지 못했다.

11. 기타사항
    추가 공지사항에 의하면 printf(" ' ");, printf(' " '); 와 같은 경우도 생각해줘야 한다고 했는데, c 코드를 돌려볼 때 이에 관련된 오류를 발견하지 못했고 그 밖의 설명이나 문제거리를 접한 바가 없어서 무슨 경우를 뜻하는 것인지 확신하지 못한 상황이다.
    또한, LaTeX에서의 Lex 코드 설명란에서 주석을 LaTeX로 구현하는 것을 숙지하지 못하였다.

제 3 절 Reference

1. https://stackoverflow.com/questions/41900370/writing-lex-and-yacc-rules-to-detect-include-in-c-language

2. https://kgon.tistory.com/90

3. https://github.com/SilverScar/C-Language-Parser/commit/f27512f6c789481ad87aaf0c32db6422c90c0c2b8118486a76df593ce14R13

4. https://stackoverflow.com/questions/56087456/is-there-any-c-library-for-drawing-pictures-and-making-jpg-files

5. https://www.lysator.liu.se/c/ANSI-C-grammar-l.html

6. https://www.lysator.liu.se/c/ANSI-C-grammar-y.html

7. https://stackoverflow.com/questions/41900370/writing-lex-and-yacc-rules-to-detect-include-in-c-language

8. https://www.joinc.co.kr/w/Site/Development/Env/Yacc

```
123
124 inclusive_or_expression
125         : exclusive_or_expression
126         | inclusive_or_expression '|' exclusive_or_expression {exp_count++;}
127         ;
128
129 logical_and_expression
130         : inclusive_or_expression
131         | logical_and_expression AND_OP inclusive_or_expression {exp_count++;}
132         ;
133
134 logical_or_expression
135         : logical_and_expression
136         | logical_or_expression OR_OP logical_and_expression {exp_count++;}
137         ;
138
139 conditional_expression
140         : logical_or_expression
141         ;
142
143 assignment_expression
144         : conditional_expression
145         | unary_expression assignment_operator assignment_expression {exp_count++;}
146         ;
147
148 assignment_operator
149         : '='
150         | MUL_ASSIGN
151         | DIV_ASSIGN
152         | MOD_ASSIGN
153         | ADD_ASSIGN
154         | SUB_ASSIGN
155         | LEFT_ASSIGN
156         | RIGHT_ASSIGN
157         | AND_ASSIGN
158         | XOR_ASSIGN
159         | OR_ASSIGN
160         ;
161
162 expression
163         : assignment_expression
164         | expression ',' assignment_expression
165         ;
166
167 constant_expression
168         : conditional_expression
169         ;
```

```
171 declaration
172         : declaration_specifiers ';'
173         | declaration_specifiers init_declarator_list ';'
174         {
175                 if(check_int == 1)
176                         int_count += num;
177                 if(check_char == 1)
178                         char_count += num;
179                 if(check_pointer == 1)
180                         pointer_count += num;
181                 if(check_array == 1)
182                         array_count += num;
183                 if(check_pointer == 1)
184                         pointer_count += num;
185                 check_array = 0;
186                 check_int = 0;
187                 check_char = 0;
188                 check_pointer = 0;
189                 num = 0;
190         }
191         ;
192
193 declaration_specifiers
194         : storage_class_specifier
195         | storage_class_specifier declaration_specifiers
196         | type_specifier
197         | type_specifier declaration_specifiers
198         | type_qualifier
199         | type_qualifier declaration_specifiers
200         ;
201
202 init_declarator_list
203         : init_declarator {num++;}
204         | init_declarator_list ',' init_declarator {num++;}
205         ;
206
207 init_declarator
208         : declarator
209         | declarator '=' initializer {exp_count++;}
210         ;
211
212 storage_class_specifier
213         : TYPEDEF
214         | EXTERN
```

```
type_specifier
        : VOID {check_int = 0; check_char = 0;}
        | CHAR {check_int = 0; check_char = 1;}
        | SHORT {check_int = 0; check_char = 0;}
        | INT {check_int = 1; check_char = 0;}
        | LONG {check_int = 0; check_char = 0;}
        | FLOAT {check_int = 0; check_char = 0;}
        | DOUBLE {check_int = 0; check_char = 0;}
        | SIGNED
        | UNSIGNED
        | struct_or_union_specifier
        | enum_specifier
        | TYPE_NAME
        ;

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 {exp_count++;}
        ;

type_qualifier
        : CONST
        | VOLATILE
        ;

declarator
        : pointer direct_declarator {pointer_count++;}
        | direct_declarator
        ;

```

```
direct_declarator
        : IDENTIFIER
        {
                if(check_pointer == 1)
                        pointer_count++;
                check_pointer = 0;
        }
        | '(' declarator ')'
        | direct_declarator '[' constant_expression ']'
        {
                check_array = 1;
        }
        | direct_declarator '[' ']'
        {
                check_array = 1;
        }
        | direct_declarator '(' parameter_type_list ')'
        {

        }
        | direct_declarator '(' identifier_list ')'
        | direct_declarator '(' ')'
        ;

pointer
        : '*'
        | '*' type_qualifier_list
        | '*' pointer
        | '*' type_qualifier_list pointer
        ;

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
        : declaration_specifiers declarator
        {
                if(check_int == 1)
                        int_count++;
                check_int = 0;
                if(check_char == 1)
                        char_count++;
                check_char = 0;
        }
        | declaration_specifiers abstract_declarator
        | 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 '(' ')'
        | direct_abstract_declarator '(' parameter_type_list ')'
        ;
```

```
388 initializer
389         : assignment_expression
390         | '{' initializer_list '}'
391         | '{' initializer_list ',' '}'
392         ;
393
394 initializer_list
395         : initializer
396         | initializer_list ',' initializer
397         ;
398
399 statement
400         : labeled_statement
401         | compound_statement
402         | expression_statement
403         | selection_statement
404         | iteration_statement
405         | jump_statement
406         ;
407
408 labeled_statement
409         : IDENTIFIER ':' statement
410         | CASE constant_expression ':' statement
411         | DEFAULT ':' statement
412         ;
413
414 compound_statement
415         : '{' '}'
416         | '{' statement_list '}'
417         | '{' declaration_list '}'
418         | '{' declaration_list statement_list '}'
419         ;
420
421 declaration_list
422         : declaration
423         | declaration_list declaration
424         ;
425
426 statement_list
427         : statement
428         | statement_list statement
429         ;
430
431 expression_statement
432         : ';'
433         | expression ';'
434         ;
```

```
436 selection_statement
437         : IF '(' expression ')' statement {select_count++;}
438         | SWITCH '(' expression ')' statement {select_count++;}
439         ;
440
441 iteration_statement
442         : WHILE '(' expression ')' statement {loop_count++;}
443         | DO statement WHILE '(' expression ')' ';' {loop_count++;}
444         | FOR '(' expression_statement expression_statement ')' statement {loop_count++;}
445         | FOR '(' expression_statement expression_statement expression ')' statement {loo
    p_count++;}
446         ;
447
448 jump_statement
449         : GOTO IDENTIFIER ';'
450         | CONTINUE ';'
451         | BREAK ';'
452         | RETURN ';' {return_count++;}
453         | RETURN expression ';' {return_count++;}
454         ;
455
456 external_declaration
457         : '#' IDENTIFIER IDENTIFIER CONSTANT
458         | '#' IDENTIFIER '<' IDENTIFIER '.' IDENTIFIER '>'
459         | function_definition
460         | declaration
461         ;
462
463 start_state
464         : external_declaration
465         | start_state external_declaration
466         ;
467
468 function_definition
469         : declaration_specifiers declarator declaration_list compound_statement
470         {
471                 func_count++;
472                 if(check_pointer == 1)
473                         pointer_count--;
474                 check_pointer = 0;
475         }
476         | declaration_specifiers declarator compound_statement
477         {
478                 func_count++;
```

```
483         | declarator declaration_list compound_statement
484         {
485                 func_count++;
486                 if(check_pointer == 1)
487                         pointer_count--;
488                 check_pointer = 0;
489         }
490         | declarator compound_statement
491         {
492                 func_count++;
493                 if(check_pointer == 1)
494                         pointer_count--;
495                 check_pointer = 0;
496         }
497         ;
498
499 %%
500 #include <stdio.h>
501
502 extern char yytext[];
503 extern int column;
504
505 yyerror(s)
506 char *s;
507 {
508         fflush(stdout);
509         printf("\n%*s\n%*s\n", column, "^", column, s);
510 }
511
512 int main(void)
513 {
514         yyparse();
515
516         printf("함수  = %d\n", func_count);
517         printf("수식  = %d\n", exp_count);
518         printf("int 변수  선언  =  %d\n", int_count);
519         printf("char 변수  선언  = %d\n", char_count);
520         printf("pointer 변수  선언  = %d\n", pointer_count);
521         printf("배열  변수  선언  = %d\n", array_count);
522         printf("선택문  = %d\n", select_count);
523         printf("반복문  = %d\n", loop_count);
524         printf("리턴문  = %d\n", return_count);
525
526         return 0;
527 }
```