

LEX와 YACC

작성자 : americanoJH

E-MAIL : americano@korea.ac.kr

1. LEX

가. LEx 소개

컴파일러 구성을 도와주는 대표적인 소프트웨어 도구로써 일정한 구조에 따라 입력된 내용을 변환하는 프로그램을 만드는 데 유용하게 쓰인다. 입력 파일에서 일정한 패턴을 찾아내는 간단한 텍스트검색 프로그램에서 소스 프로그램을 최적화된 목적 코드(object code)로 변형하는 C 컴파일러까지 다양한 프로그램을 만드는 데 도움을 준다. 원래 UNIX의 산실인 AT&T Bell Laboratories에서 UNIX 시스템의 유틸리티로서 개발되었다. Lex의 개발자는 Mark Lesk 이다.

이후 이 도구들의 유용성이 확장되면서 Sun의 Solaris나 Linux와 같은 UNIX 계열의 운영체제는 물론이고, Microsoft Windows 등의 상이한 운영체제에서도 이식되어서 널리 사용되고 있다.

일정한 구조를 가진 입력을 받아들이는 프로그램에서 지속적으로 반복하는 작업은 입력 내용을 의미 있는 단위(unit)로 분해하여 그들 사이의 관련성을 파악하는 일이다. 입력을 토큰이라고 하는 단위로 나누는 작업을 대개 어휘 분석(lexical analysis) 혹은 줄여서 렉싱(lexing)이라고 한다.

렉스는 몇개의 토큰을 받아들여 주어진 토큰을 식별할 수 있는 C 루틴(routine)을 생성한다. 이 루틴을 어휘 분석기(lexical analyzer) 또는 렉서(lexer)나 스캐너(scanner)라고 한다. 한편 사용자가 렉스에 기술하는 내용을 일컬어 렉스 명세서(lex specification)라 한다.

렉스가 사용하는 토큰 설명(token descriptions)을 보통 정규 표현식(regular expressions)이라하며, grep이나 egrep 같은 유닉스 명령어에서 사용하는 패턴을 크게 확장한 버전이다. 렉스는 찾고자 하는 표현식의 개수에 관계없이 렉서가 입력 텍스트를 최대한 빠르게 검색할 수 있는 형태로 이러한 정규 표현식을 바꾸어 준다.

나. LEX 동작방법

lex를 연동하면 구문 분석기를 만드는 C 소스 코드를 생성할 수 있다. lex는 설정 파일을 사용하여 각각 설정 파일을 받아들여 적절한 C 코드를 생성한다. lex 도구가 생성하는 C 코드는 특정한 어휘를 식별한 후 토큰을 반환한다.

LEX는 3개의 절로 나뉜다. 각 절의 끝에는 %% 기호를 넣는다. 이 세 부분들 중 꼭 있어야 하는 필수 부분은 두 번째 부분이고, 첫 번째와 세 번째 부분은 필요 없는

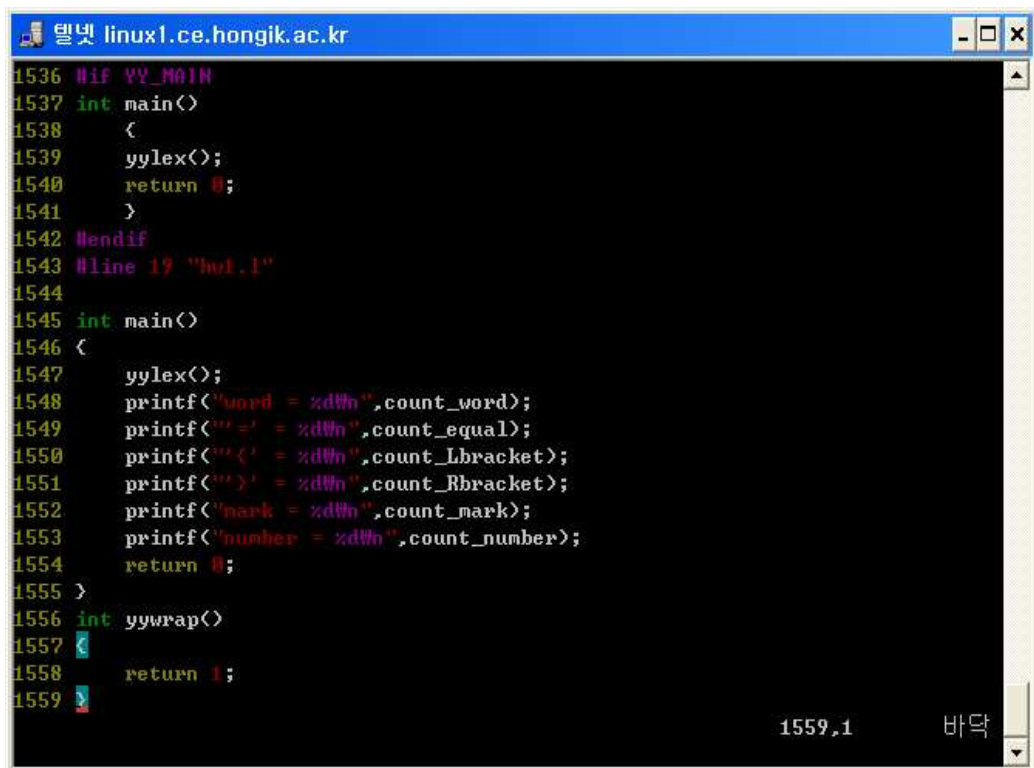
경우 생략이 가능하다

첫번째 절은 정의절(definition section)이라 하며 최종 프로그램에 포함하고자 하는 C 프로그램의 내용을 삽입하는 기능을 담당한다. 선언(declaration)이나 정의(definition)가 포함된다. 그러나 최종 파일 안에 반드시 포함되어야하는 헤더 파일이 존재하는 경우에는 이 절이 단순한 삽입 이상의 중요한 기능을 담당한다. 이 절에 존재하는 C 코드는 "%{"과 "%}"라는 특별한 분리자(delimiter)로 둘러싸여 있다. 렉스는 이 분리자 사이에 있는 내용을 C파일에 그대로 복사한다.

두 번째 절은 규칙절(rules section)이다. 본론에 해당하는 것으로서 각 프로그램이 수행하는 일에 대한 규칙(보통 번역 규칙(translation rule)이라고 부른다)을 기술하고 각각의 규칙은 두 부분, 즉 공백으로 구분된 패턴(pattern)과 동작(action)으로 이루어진다. 렉스가 생성하는 렉서는 패턴을 인식했을 때 주어진 동작을 수행한다. 이러한 패턴은 유닉스에서 흔히 사용되는 정규 표현식(regular expressions)과 비슷하지만, grep, sed, ed 등에서 사용하는 표현식보다는 좀더 확장된 내용을 담고 있다. 패턴이 중복될 수 있으므로 렉스에는 모호함을 제거하는 규칙이 있다, 렉서가 제대로 동작하도록 하는 두 가지 규칙은 다음과 같다.

1. 렉스 패턴은 주어진 문자나 문자열에 오직 한 번만 매치한다.
2. 렉스는 현재 주어진 입력에서 패턴에 매치하는 문자 또는 문자열 중 가장 길게 매치할 수 있는 것(longest possible match)에 대한 동작을 수행한다.

세 번째 절은 사용자 서브루틴절(user subroutine section)이며, 보조 프로시저(auxiliary procedure) 또는 지원 프로그램(supporting routines)을 담고 있다. 일반적인 C 코드가 작성될 수 있는 곳이다. 렉스는 생성한 C코드 뒤에 이 절에 있는 내용을 복사한다.



```
1536 #if YY_MAIN
1537 int main()
1538 {
1539     yylex();
1540     return 0;
1541 }
1542 #endif
1543 #line 19 "hwl.1"
1544
1545 int main()
1546 {
1547     yylex();
1548     printf("word = %d\n", count_word);
1549     printf("'=' = %d\n", count_equal);
1550     printf("<' = %d\n", count_Lbracket);
1551     printf(">' = %d\n", count_Rbracket);
1552     printf("mark = %d\n", count_mark);
1553     printf("number = %d\n", count_number);
1554     return 0;
1555 }
1556 int yywrap()
1557 {
1558     return 1;
1559 }
```

1559,1 바닥

위의 그림은 렉스가 생성한 C 코드인 lex.yy.c의 내부 코드이다. 그리고 이 장면은 마지막에 사용자 서브루틴절이 추가된 화면이다.

2. 코드 설명

가. 정의절 (definition section)

```
%{
#include<stdio.h>
int count_word = 0;
int count_number = 0;
int count_equal = 0;
int count_Lbracket = 0;
int count_Rbracket = 0;
int count_mark = 0;
}%
%%
```

정의절에는 최종 프로그램에 포함하고자 하는 C 프로그램의 내용을 삽입하는 기능을 담당한다. 삽입되는 내용은 최종파일에서 출력을 위해 필요한 헤더 파일과 분석된 코드의 카운트를 담당하는 변수가 포함된다. 이 변수들은 그 다음의 규칙절에서 사용되며 해당되는 패턴이 나올 경우 값을 1씩 증가시킨다. 최종적으로 각 토큰의 개수를 가지게 된다.

나. 규칙절(rules section)

<code>([a-zA-Z])+</code>	<code>{count_word++;}</code>
<code>([0-9])+</code>	<code>{count_number++;}</code>
<code>"=</code>	<code>{count_equal++;}</code>
<code>"{"</code>	<code>{count_Lbracket++;}</code>
<code>"}"</code>	<code>{count_Rbracket++;}</code>
<code>\n</code>	<code>{count_mark++;}</code>
<code>.</code>	<code>{count_mark++;}</code>
<code>%%</code>	

규칙절에서는 입력된 문자에서 매칭되는 문자열의 패턴과 그 패턴이 나타났을 때 해당하는 동작으로 이루어졌다.

패턴	동작
<code>([a-zA-Z])+</code>	a부터 z까지 혹은 A부터 Z까지의 문자로만 구성되는 문자열이 나타났을 때 <code>count_word</code> 의 변수값을 1증가시킨다.
<code>([0-9])+</code>	0부터 9까지의 숫자로 구성되는 수, 즉 10진 수가 나타났을 때 <code>count_number</code> 의 변수값을 1증가시킨다.
<code>"=</code>	'='의 기호가 나타나면 <code>count_equal</code> 의 변수값을 1 증가시킨다.
<code>"{"</code>	'{'의 기호가 나타나면 <code>count_Lbracket</code> 의 변수값을 1증가시킨다.
<code>"}"</code>	'}'의 기호가 나타나면 <code>count_Rbracket</code> 의 변수값을 1증가시킨다.
<code>\n</code>	한줄 개행이 나타나면 <code>count_mark</code> 의 변수값을 1증가시킨다.
<code>.</code>	위의 패턴에 해당되지 않는 코드가 나타나면 <code>count_mark</code> 의 변수값을 1증가시킨다.

다. 사용자 서브루틴절(user subroutine section)

```
int main()
{
    yylex();
    printf("word = %d\n",count_word);
    printf("'=' = %d\n",count_equal);
    printf("'{' = %d\n",countLbracket);
    printf("'}' = %d\n",countRbracket);
    printf("mark = %d\n",count_mark);
    printf("number = %d\n", count_number);
    return 0;
}
int yywrap()
{
    return 1;
}
```

사용자 서브루틴절 에는 각 토큰들의 개수를 출력하는 내용의 `main` 함수와 `Lex`

Functions 중 하나인 yywrap()가 정의되어 있다.

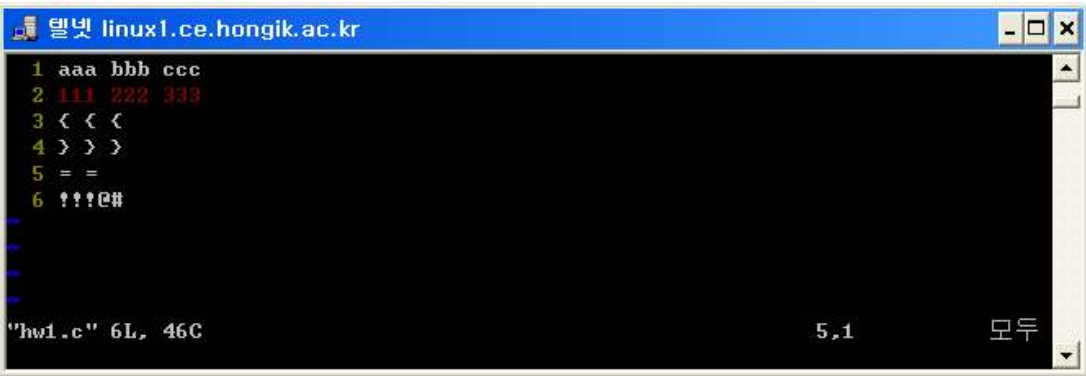
main 함수를 보면 먼저 yylex() 함수가 있다. 이 함수는 yywrap()와 마찬가지로 Lex Functions 중 하나로서 실행되면 입력파일의 분석을 시작한다.

그 다음부터는 정의절에서 선언되었던 변수들의 값들을 출력하는 함수들이다. 이 출력을 위해서 정의절에서 각각의 변수들과 헤더파일이 포함되었다.

마지막으로 yywrap 함수는 파일의 EOF가 나타나면 불러지는 함수이다. 이 함수가 1을 리턴하면 파싱의 종료를 지시한다.

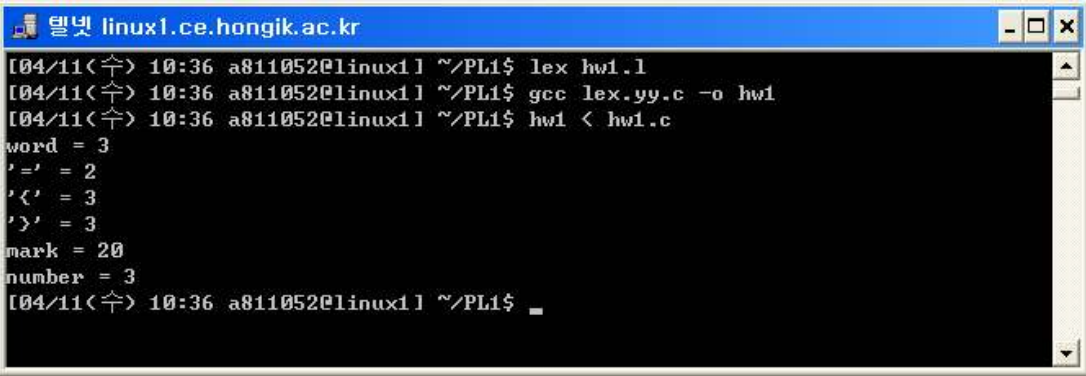
3. 출력화면

입력파일 : hw1.c



word	3개	aaa bbb ccc
number	3개	111 222 333
'{'	3개	{ { {
'}'	3개	} } }
'='	2개	= =
mark	20개	!!!@t#(5개), \n(5개), \t(9개), EOF(1개)

실행화면 : hw1.l



2. YACC

가. YACC 소개

본래 YACC는 유닉스 환경에서 실행되는 구문분석 테이블 생성기를 말하는 것이었으나 오늘날에는 다른 환경에서 사용되는 제품도 개발되어 있다. 구문분석 테이블 생성을 위하여 사용되는 시스템 소프트웨어로 입력된 프로그램 명령어의 구문을 분석하여 구문분석 트리를 자동적으로 생성하는 작업을 수행한다.

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)이라고 하며, 그런 일을 하는 것을 어휘분석기(Lexical Analyzer 또는 lexer · scanner)라고 부른다. 입력값이 의미단위로 나뉘게 되면 프로그램은 그것들간의 관계를 따지게 된다. 예를 들어 C 컴파일러는 토큰들이 수식인지, 문장인지, 선언문인지, 블록인지, 프로시저인지를 판별해낸다. 이런 작업이 구문분석이고 그 관계들을 정해 놓은 규칙을 문법(Grammar)이라고 한다.

YACC는 구문분석기(Parser)를 생성해 주는 도구이다. 구문분석기는 어휘분석기로부터 받은 의미단위를 문법에 맞는지 검사하는 일을 한다. 사용자가 문법을 정의하고 그 문법에 맞을 경우 취할 행동을 C 언어로 기술해 주면 YACC가 알아서 구문분석기를 생성해준다. 구문분석기는 의미단위를 가져와서 구문분석을 하는 프로그램이므로 어휘분석을 하는 프로그램은 따로 구현이 되어야 한다.

[출처] YACC [yet another compiler compiler] | 네이버 백과사전

나. 야크 문법

Yacc의 Grammar 화일은 Lex Specification과 비슷하다.

Yacc Grammar
%{ <u>C 선언 부분(C Definition Section)</u> %} <u>Yacc 선언 부분(Yacc Definition Section)</u> %% <u>문법 부분(Grammar Rules Section)</u> %% <u>사용자 정의 함수(User Subroutines)</u>

1) C 선언 부분(C Definition Section)

이곳에 쓰여진 것들은 그대로 y.tab.c에 복사가 된다. Lex에서와 마찬가지로 다른 부분에서 사용할 변수가 있다면 이곳에서 미리 정의를 하고 헤더화일을 include해야 한다.

2) Yacc 선언 부분(Yacc Definition Section)

문법 부분에서 사용하는 토큰, 결합법칙, 변수나 토큰들의 타입 등을 선언한다.

3) 문법부분(Grammar Rules Section)

인식할 문법과 그에 따라 취할 C 언어로 된 행동 정의한다. -> 표시를 할수 없으므로 :으로 대신하며, 문법의 마지막은 ;으로 끝나야한다.

4) 사용자 정의 함수(User Subroutines)

사용자가 만들어서 사용해야 할 함수가 있으면 이곳에 정의 한다.

다. 동작 방식

Yacc는 토큰을 계속 읽어서 사용자가 정의한 문법과 맞춰가며 구문 분석을 하게 된다. 읽어 들인 토큰이 문법을 완전히 만족하지 않고 다른 토큰이 더 필요하다면 그 토큰을 스택에 쌓아 두는데 이것을 스택에 shift 한다고 해서 shift라고 한다.

계속 토큰을 읽어서 스택에 있는 토큰들과 함께 문법과 비교를 했더니 문법에 만족을 하면 스택에 있던 토큰을 꺼내고(pop) 문법의 왼쪽(LHS) 심볼로 대치한다. 이것을 reduce라고 한다.

Yacc가 구문분석을 하는데 도움을 주는 툴이지만 Yacc도 구문분석을 할 수 없는 문법이 있다. 하나의 토큰이 여러개의 문법에 적용되어 하나의 파스 트리가 생성되지 않고 여러개의 트리가 생성되는 모호한 문법의 경우나, Yacc는 다음 토큰 하나를 가져와서 살펴보는데, 두개 이상의 토큰을 가져와야 분석을 할 수 있는 문법등을 구문분석 할 수 없다.

3. 코드 분석

가. 함수 선언

1) 일반형식

[type] 함수명 ([전달인자 선언]) ;

2) 관련 YACC 문법

[type] 함수명	([전달인자 선언]) ;
direct_declarator	
: IDENTIFIER	
direct_declarator '(' parameter_type_list '	
direct_declarator '(' identifier_list '	
direct_declarator '(' ')'	

[type] 함수명
declaration
declaration_specifiers init_declarator_list ;'

[type]
declaration_specifiers
: storage_class_specifier
storage_class_specifier declaration_specifiers
type_specifier
type_specifier declaration_specifiers
type_qualifier
type_qualifier declaration_specifiers

함수명
init_declarator_list -> init_declarator -> declarator
declarator
: pointer direct_declarator
direct_declarator

3) 관련 카운팅 코드

```

direct_declarator
{
    direct_declarator '(' parameter_type_list ')'
    {
        def func++;
        if(check==1)
            ptr_iden--;
        else
            var_iden--;
        check = 0;
    }
    | direct_declarator '(' identifier_list ')'
    {
        def func++;
        if(check==1)
            ptr_iden--;
        else
            var_iden--;
        check = 0;
    }
    | direct_declarator '(' ')'
    {
        def func++;
        if(check==1)
            ptr_iden--;
        else
            var_iden--;
        check = 0;
    }
}
;

```

4) 카운팅 코드 분석

관련 야크 문법에 함수선언 카운팅 변수를 1씩 증가시킵니다.

단, 이 YACC 문법에 들어오기 전에 포인터 변수 선언 카운팅과 일반변수선언카운팅과 중복되므로 그들의 카운팅을 1씩 빼주어야 합니다.

나. 함수 정의

1) 일반형식

```

[type] 함수명 ([전달인자 선언])
{
    //몸체시작
    내부변수 선언;
    수행 문장들;
    return 반환데이터타입 ;
}
// 함수 몸체 종료

```

2) 관련 YACC 문법

[type]	함수명	{ 내용 }
function_definition		: declaration_specifiers declarator declaration_list compound_statement declaration_specifiers declarator compound_statement declarator declaration_list compound_statement declarator compound_statement

함수명
postfix_expression -> primary_expression
primary_expression : IDENTIFIER '(' expression ')'

([전달인자])
argument_expression_list : assignment_expression argument_expression_list ',' assignment_expression

3) 관련 카운팅 코드

postfix_expression postfix_expression '(' ')' {cal_func++;} postfix_expression '(' argument_expression_list ')' {cal_func++;}

4) 카운팅 코드 분석

관련 야크 문법에 함수정의 카운팅 변수를 1씩 증가시킵니다.

중복되는 것은 없으므로 따로 카운팅 변수를 빼주지 않아도 괜찮습니다.

라. 일반 변수 선언문

1) 일반형식

[type] 변수명 ;

2) 관련 YACC 문법

[type]	변수명
declaration declaration_specifiers init_declarator_list ';' ;	

[type]
declaration_specifiers : storage_class_specifier storage_class_specifier declaration_specifiers type_specifier type_specifier declaration_specifiers type_qualifier type_qualifier declaration_specifiers

변수명
init_declarator_list -> init_declarator -> declarator init_declarator_list -> init_declarator -> declarator '=' initializer
declarator -> direct_declarator
direct_declarator : IDENTIFIER
initializer -> assignment_expression
assignment_expression : conditional_expression unary_expression assignment_operator assignment_expression

3) 관련 카운팅 코드

declarator : pointer direct_declarator {check = 1;} direct_declarator ;
direct_declarator : IDENTIFIER { if(check==1) ptr_iden++; else var_iden++; check = 0; }

4) 카운팅 코드 분석

변수명은 오로지 IDENTIFIER만 가능합니다.

하지만 이 direct_declarator를 통한 IDENTIFIER의 접근은 일반변수뿐만 아니라 포인터 변수명, 그리고 함수명에서도 접근이 가능합니다. 함수명은 좀 더 복잡한 문법이 되므로 함수 카운팅 하는 곳에서 따로 카운팅문을 만들어주고 여기에서는 if문을 사용하여 포인터변수선언이 아닐 경우에만 일반변수선언 카운터를 증가시켜주었습니다.

마. 포인터 변수 선언

1) 일반형식

[type] 포인터 변수명 ;

2) 관련 YACC 문법

[type]	포인터 변수명
declaration declaration_specifiers init_declarator_list ;	

[type]
<pre> declaration_specifiers : storage_class_specifier storage_class_specifier declaration_specifiers type_specifier type_specifier declaration_specifiers type_qualifier type_qualifier declaration_specifiers </pre>

포인터	변수명
init_declarator_list -> init_declarator -> declarator	
<pre> declarator : pointer direct_declarator </pre>	

포인터
<pre> pointer : '*' '*' type_qualifier_list '*' pointer '*' type_qualifier_list pointer ; </pre>

변수명
init_declarator_list -> init_declarator -> declarator
init_declarator_list -> init_declarator -> declarator '=' initializer
declarator -> direct_declarator
<pre> direct_declarator : IDENTIFIER </pre>
initializer -> assignment_expression
<pre> assignment_expression : conditional_expression unary_expression assignment_operator assignment_expression </pre>

3) 관련 카운팅 코드

<pre> declarator : pointer direct_declarator {check = 1;} direct_declarator ; </pre>
<pre> direct_declarator : IDENTIFIER { if(check==1) ptr_iden++; else var_iden++; check = 0; } </pre>

4) 카운팅 코드 분석

일반변수 선언문과 비슷합니다. 단지 포인터 변수의 접근은 꼭 pointer direct_declarator를 통해 접근하므로 이 때 포인터변수임을 체크를 해주어서 카운팅 시 일반변수와 구분합니다.

바. 배열 변수 선언

1) 일반형식

[type] 변수명 [상수] ;

2) 관련 YACC 문법

[type]	변수명 [상수]
declaration	
declaration_specifiers init_declarator_list ';'	

[type]
declaration_specifiers
: storage_class_specifier
storage_class_specifier declaration_specifiers
type_specifier
type_specifier declaration_specifiers
type_qualifier
type_qualifier declaration_specifiers

변수명	[상수]
init_declarator_list -> init_declarator -> declarator -> direct_declarator	
direct_declarator	
: IDENTIFIER	
direct_declarator '[' constant_expression ']'	
direct_declarator '[' ']'	

변수명
direct_declarator
: IDENTIFIER

[상수]
constant_expression -> conditional_expression -> logical_or_expression ->
logical_and_expression -> inclusive_or_expression -> exclusive_or_expression ->
and_expression -> equality_expression -> relational_expression -> shift_expression ->
additive_expression -> multiplicative_expression -> cast_expression ->
unary_expression -> postfix_expression -> primary_expression
primary_expression
CONSTANT

3) 관련 카운팅 코드

```
direct_declarator
{
    direct_declarator '[' constant_expression ']'
    {
        arr_iden++;
        if(check==1)
            ptr_iden--;
        else
            var_iden--;
        check = 0;
    }

    | direct_declarator '[' ']'
    {
        arr_iden++;
        if(check==1)
            ptr_iden--;
        else
            var_iden--;
        check = 0;
    }
}
```

4) 카운팅 코드 분석

배열의 주 포인트는 '[' ']' 기호입니다. 이 기호가 있을 때만 배열이 선언 되므로 그 기호를 통해 문법이 정의되는 곳에 카운팅을 해주었습니다. 하지만 결국 배열 변수명은 IDENTIFIER 이므로 일반변수 또는 포인터 변수와 중복됩니다. 따라서 일반변수 또는 포인터변수의 값을 1씩 빼주면서 카운팅을 해주었습니다.

사. 수식

1) 일반형식

```
식
```

2) 관련 YACC 문법

식
expression : assignment_expression expression ',' assignment_expression ;

3) 관련 카운팅 코드

```
expression
: assignment_expression {exp_state++;}
| expression ',' assignment_expression {exp_state++;}
;
```

4) 카운팅 코드 분석

YACC의 모든 식은 expression으로 표현됩니다. expression을 분석하면 연산자를 이용한 수식들을 표현할 수 있는 구문이 나옵니다. expression 가 식을 표현하는 가장 제한적이면서 최상위적인 구문이므로 이 곳에 카운팅을 해주었습니다.

아. 리턴문

1) 일반형식

```
return 반환데이터타입
```

2) 관련 YACC 문법

return 반환데이터타입	
jump_statement	RETURN ';' RETURN expression ';' ;

반환데이터타입	
expression	: assignment_expression expression ',' assignment_expression ;

3) 관련 카운팅 코드

```
jump_statement  
| RETURN ';' { ret_stat++; }  
| RETURN expression ';' { ret_stat++; }
```

4) 카운팅 코드 분석

리턴문은 오로지 return 토큰을 이용해야만 합니다. return 토큰 뒤에는 아무 것도 없을 수 도 있고 수식이 존재할 수 있습니다. 따라서 만약 return 뒤에 수식이 존재한다면 리턴문 카운터 뿐만아니라 수식 카운터도 같이 증가하게 됩니다.

자. 선택문

1) 일반형식

```
switch (조건수식) {  
    case 상수1 :  
        문장들 ;  
        break;  
    case 상수2 :  
        문장들 ;  
        break;  
    :  
    default :  
        문장들;  
}
```

2) 관련 YACC 문법

switch	(조건수식)	{ 내용 }
selection_statement	SWITCH '(' expression ')' statement ;	

(조건수식)
expression : assignment_expression expression ',' assignment_expression ;
{ 내용 }
statement -> labeled_statement
labeled_statement CASE constant_expression ':' statement DEFAULT ':' statement ;

3) 관련 카운팅 코드

selection_statement SWITCH '(' expression ')' statement {swi_stat++;}
--

4) 카운팅 코드 분석

선택문은 switch라는 토큰이 있어야만 실행되는 구문이므로 그 구문이 있는 곳에 카운팅을 해주었습니다. 스위치문은 필수적으로 조건수식이 있어야 하므로 무조건 수식 카운터 또한 증가하게됩니다.

차. 조건문

1) 일반형식

if(조건문장) { 처리문장 ; //조건이 참일 때 수행 } 다음문장; // if문과 별개

if(조건문장) { 처리문장 1 ; // 조건이 참일 때 수행 } else { 처리문장 2; // 조건이 거짓일 때 수행 }
--

논리수식 ? 수식 : 조건수식

2) 관련 YACC 문법

if	(조건문장)	{ 내용 }
: IF '(' expression ')' statement		

if	(조건문장)	{ 내용 }	else	{ 내용 }
selection_statement IF '(' expression ')' statement ELSE statement				

논리수식 ? 수식 : 조건수식
conditional_expression logical_or_expression '?' expression ':' conditional_expression
(조건문장)
수식
expression : assignment_expression expression ',' assignment_expression ;
{ 내용 }
statement : labeled_statement compound_statement expression_statement selection_statement iteration_statement jump_statement ;
논리수식
logical_or_expression : logical_and_expression logical_or_expression OR_OP logical_and_expression ;
조건수식
conditional_expression : logical_or_expression logical_or_expression '?' expression ':' conditional_expression ;

3) 관련 카운팅 코드

selection_statement : IF '(' expression ')' statement {if_state++;} IF '(' expression ')' statement ELSE statement {if_state++;}
conditional_expression logical_or_expression '?' expression ':' conditional_expression {if_state++;}

4) 카운팅 코드 분석

조건문은 크게 2가지로 나누어 집니다.

하나는 IF와 ELSE 라는 토큰으로 이루어진 구문과 다른 하나는 삼항연산자를 이용한 구문입니다.

ELSE의 사용여부에 따라 앞의 구문 또한 작게 2가지로 나누어 집니다. 따라서 그 모든 구문에 카운팅을 해주어야 합니다. 여기서 주의 점은 문법 충돌이 발생한다는 점입니다.

IF '(' expression ')' statement ELSE statement 문장 안에는 이미

IF '(' expression ')' statement 문장이 포함되었기 때문입니다. 그 문장이 모호하기 때문에 분석 실행 시 문법 충돌이 발생합니다.

카. 선택문

1) 일반형식

```
for (초기식 ; 조건식 ; 증감식) {
    반복할 문장;
}
```

```
While(조건식) {
    반복한 문장;
}
```

```
do {
    반복한 문장;
} while(조건식);
```

2) 관련 YACC 문법

for	(초기식 ; 조건식 ; 증감식)	{ 내용 }
iteration_statement FOR '(' expression_statement expression_statement ')' statement FOR '(' expression_statement expression_statement expression ')' statement		

while	(조건식)	{ 내용 }
iteration_statement : WHILE '(' expression ')' statement		

do	{ 내용 }	while	(조건식)
iteration_statement DO statement WHILE '(' expression ')' ';' ;			

초기식
조건식(for문)
expression_statement : ',' expression ',' ;

증감식
조건식(while문)
expression : assignment_expression expression ',' assignment_expression ;

{ 내용 }
statement : labeled_statement compound_statement expression_statement selection_statement iteration_statement jump_statement ;

3) 관련 카운팅 코드

```
iteration_statement
: WHILE '(' expression ')' statement { for_stat++;}
| DO statement WHILE '(' expression ')' ';' { for_stat++;}
| FOR '(' expression_statement expression_statement ')' statement { for_stat++;}
| FOR '(' expression_statement expression_statement expression ')' statement { for_stat++;}
;
```

4) 카운팅 코드 분석

반복문은 for문 while문 do ~ while 문으로 나누어 집니다.

따라서 for , while, do 세 토큰으로 이루어진 문법에 카운팅을 해주었습니다. for문의 경우 (초기식 ; 조건식 ; 증감식) 인데 이 3개의 식은 모두 생략가능한데 증감식 뒤에는 ';' 기호가 안 붙으므로 for문은 2개의 문법이 정의 되었습니다.

4. 전체코드

```
&{
#include<stdio.h>
int def_func = 0;
int dec_func = 0;
int cal_func = 0;
int ptr_iden = 0;
int arr_iden = 0;
int var_iden = 0;
int exp_stat = 0;
int swi_stat = 0;
int if_state = 0;
int for_stat = 0;
int ret_stat = 0;
int check++;
&}

%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

%start translation_unit
%%

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

postfix_expression
: primary_expression
| postfix_expression '[' expression ']'
| postfix_expression '(' ')' {cal_func++;}
| postfix_expression '(' argument_expression_list ')' {cal_func++;}
```

```

| postfix_expression '.' IDENTIFIER
| postfix_expression PTR_OP IDENTIFIER
| postfix_expression INC_OP
| postfix_expression DEC_OP
;

argument_expression_list
: assignment_expression
| argument_expression_list ',' assignment_expression
;

unary_expression
: postfix_expression
| INC_OP unary_expression
| DEC_OP unary_expression
| 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
| multiplicative_expression '/' cast_expression
| multiplicative_expression '%' cast_expression
;

additive_expression
: multiplicative_expression
| additive_expression '+' multiplicative_expression
| additive_expression '-' multiplicative_expression

```

```
;  
  
shift_expression  
: additive_expression  
| shift_expression LEFT_OP additive_expression  
| shift_expression RIGHT_OP additive_expression  
;  
  
relational_expression  
: shift_expression  
| relational_expression '<' shift_expression  
| relational_expression '>' shift_expression  
| relational_expression LE_OP shift_expression  
| relational_expression GE_OP shift_expression  
;  
  
equality_expression  
: relational_expression  
| equality_expression EQ_OP relational_expression  
| equality_expression NE_OP relational_expression  
;  
  
and_expression  
: equality_expression  
| and_expression '&' equality_expression  
;  
  
exclusive_or_expression  
: and_expression  
| exclusive_or_expression '^' and_expression  
;  
  
inclusive_or_expression  
: exclusive_or_expression  
| inclusive_or_expression '|' exclusive_or_expression  
;  
  
logical_and_expression  
: inclusive_or_expression  
| logical_and_expression AND_OP inclusive_or_expression  
;  
  
logical_or_expression  
: logical_and_expression  
| logical_or_expression OR_OP logical_and_expression
```



```

;

conditional_expression
: logical_or_expression
| logical_or_expression '?' expression ':' conditional_expression {if_state++;}
;

assignment_expression
: conditional_expression
| unary_expression assignment_operator assignment_expression
;

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 {exp_state++;}
| expression ',' assignment_expression{exp_state++;}
;

constant_expression
: conditional_expression
;

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

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
| init_declarator_list ';' init_declarator
;

init_declarator
: declarator
| declarator '=' initializer
;

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

type_specifier
: VOID
| CHAR
| SHORT
| INT
| LONG
| FLOAT
| DOUBLE
| 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
;

```

```

type_qualifier
: CONST
| VOLATILE
;

declarator
: pointer direct_declarator {check = 1;}
| direct_declarator
;

direct_declarator
: IDENTIFIER
{
  if(check==1)
    ptr_iden++;
  else
    var_iden++;
  check = 0;
}
| '(' declarator ')'
| direct_declarator '[' constant_expression ']'
{
  arr_iden++;
  if(check==1)
    ptr_iden--;
  else
    var_iden--;
  check = 0;
}
}
| direct_declarator '[' ']'
{
  arr_iden++;
  if(check==1)
    ptr_iden--;
  else
    var_iden--;
  check = 0;
}
| direct_declarator '(' parameter_type_list ')'
{
  def_func++;
  if(check==1)

```

```

ptr_iden--;
else
var_iden--;
check = 0;
}
| direct_declarator '(' identifier_list ')'
{
def_func++;
if(check==1)
ptr_iden--;
else
var_iden--;
check = 0;
}
| direct_declarator '(' ' ' ')'
{
def_func++;
if(check==1)
ptr_iden--;
else
var_iden--;
check = 0;
}
;ZZ

```

```

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
| 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 ')'
;

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
;

expression_statement
: ';'
| expression ';'
;

selection_statement
: IF '(' expression ')' statement {if_state++;}
| IF '(' expression ')' statement ELSE statement {if_state++;}
| SWITCH '(' expression ')' statement {swi_stat++;}
;

```

```

iteration_statement
: WHILE '(' expression ')' statement { for_stat++;}
| DO statement WHILE '(' expression ')' ';' { for_stat++;}
| FOR '(' expression_statement expression_statement ')' statement { for_stat++;}
| FOR '(' expression_statement expression_statement expression ')' statement { for_stat++;}
;

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

translation_unit
: external_declaration
| translation_unit external_declaration
;

external_declaration
: function_definition
| declaration
;

function_definition
: declaration_specifiers declarator declaration_list compound_statement (dec_func++; def_func--)
| declaration_specifiers declarator compound_statement (dec_func++; def_func--)
| declarator declaration_list compound_statement (dec_func++; def_func--)
| declarator compound_statement (dec_func++; def_func--)
;

%%
#include <stdio.h>

int main(){
    yyparse();
    printf("함수선언 = %d\n",def_func);
    printf("함수정의 = %d\n",dec_func);
    printf("함수호출 = %d\n",cal_func);
    printf("일반변수 선언문 = %d\n",ptr_iden);
}

```



```
printf("배열변수 선언문 = %d\\n",arr_iden);
printf("포인터변수 선언문 = %d\\n", var_iden);
printf("선택문 = %d\\n", swi_stat);
printf("조건문 = %d\\n", if_state);
printf("반복문 = %d\\n", for_stat);
printf("리턴문 = %d\\n", ret_stat);
printf("수식 = %d\\n", exp_stat);
return 0;
}

yyerror(s)
char *s;
{
fflush(stdout);
}
```