Lex와 Yacc 사용법

- Lex 및 Yacc의 역사
  - 1970년대 Unix 환경 및 C 언어에서 시작
  - Lex 어휘 분석기 (tokenizer, scanner)
  - Yacc 문법 분석기 (parser)

- 발전된 도구
  - Lex → flex
  - Yacc → bison

- 설치 (Linux)
  - `$ sudo apt install flex`
  - `$ sudo apt install bison`
  - `$ sudo apt install gcc`
  - `$ sudo apt install make`

Lex 실행

- lex 파일을 코딩함 (파일 확장자 .l)
- flex를 적용하면 lex.yy.c 파일이 생성됨
- 이를 C 컴파일로 컴파일하면 어휘 분석 프로그램 실행 파일 생성

file.l ⇒ **lex** ⇒ **lex.yy.c** ⇒ **gcc** ⇒ a.out

- 명령어 실행

```
$ flex file.l                    (lex.yy.c 파일 생성)
$ gcc lex.yy.c -ll               (a.out 실행 파일 생성)
$ a.out                          (표준입력 데이터)
   혹은
$ a.out < my_input_file > my_output_file
```

Lex 파일 구조

- 3개의 Section으로 구성

```
Definition Section
%%
Rules Section
%%
User subroutines
```

- 기본은 Rules Section
- 간단한 예 (오직 Rules Section 만 정의된 경우)

```
%%
username                printf("%s", getlogin());
```

– Rules 정의

pattern : `username` *action* : `printf` (탭이나 스페이스로 구분)

예 : 라인 수와 문자 수를 카운트하기

```
int num_lines = 0, num_chars = 0;   // 두 변수 선언
%%
\n              ++num_lines; ++num_chars;
.               ++num_chars;

%%
main() {
    yylex();
    printf("# of lines = %d, # of chars = %d\n", num_lines, num_chars);
}
```

- 정의 부분 : 두 변수 선언 num_lines num_chars
  - 프로그램의 다른 부분에서 사용될 수 있음
- Rule 정의 부분
  - 두 개의 패턴 : "\n" 과 "."
  - 해당 액션 : 변수 값 +1 씩 증가
- 사용자 코드 부분
  - 위에서 계산 된 변수 값을 사용함

- 예 : 프로그램의 토큰 인식

```
/* scanner for a toy Pascal-like language */
%{
/* need this for the call to atf() below */
#include <math.h>
%}
DIGIT                    [0-9]
ID                       [a-z][a-z0-9]*
%%
{DIGIT}+        { printf ("An integer : %s (%d)\n", yytext, atoi(yytext)); }

{DIGIT}+"."{DIGIT}*      { printf("A float: %s (%f)\n", yytext, atof(yytext)); }

if|then|begin|end|procedure|function { printf("A keyword: %s\n", yytext ); }

{ID}                     printf("An identifier: %s\n", yytext);

"+"|"-"|"*"|"/"          printf("An operator: %s\n", yytext);

"{"[^}\n]*"}"            printf(""); /* eat up one-line comments */
[ \t\n]+                 printf("");  /* eat up white space */
.                        printf("Unrecognized character: $s\n", yytext);
%%
```

```
main(int argc, char ** argv) {
    ++argv, --argc;  /* skip over program name */
    if (argc > 0)
       yyin = fopen(argv[0], "r");
    else
        yyin = stdin;
     yylex();
}
```

- 정의 부분

Name	definition
DIGIT	[0-9]
ID	[a-z][a-z0-9]*

  - DIGIT : 0 에서 9 까지 수 중에 하나
  - ID : 변수 이름 정의 (영문 소문자로 시작하여, 소문자와 숫자가 반복하여 나옴)

  - 실수에 대한 어휘 정의

    {DIGIT}+"."{DIGIT}* = ([0-9])+"."([0-9])*

- Rule 정의 부분

  pattern action

- 사용자 코드
  - `lex.yy.c` 에 copy 되어 들어 감.

- 룰 패턴 정의
  - 룰 패턴의 정의는 정규식 (regular expression) 및 확장
  - 확장을 위한 특수 오퍼레이터

    `" \ [] ^ - ? . * + | ( ) $ / { } % < >`

- 패턴 매칭되는 문자

x	한 문자 x
\n	newline
.	모든 문자 매칭 (newline 제외)
\.	문자 .
[abc]	a 이거나 b 이거나 c
[a-z]	영문 소문자 중에 하나
“a”	문자 a
[abj-oZ]	a, b, j, k, l, m, n, o, Z 중에 한 문자
[^A-Z]	(negation) 영문 대문자를 제외한 모든 문자
^r	문자 r 이 라인의 맨 앞에 나오는 경우
r$	문자 r 이 라인의 맨 뒤에 나오는 경우 = “r\n”

– 연속된 문자

fred	fred
[Ff]red	fred 혹은 Fred

– 선택

aa | bb	aa 혹은 bb

– 그룹

(ab|c)d	abd 혹은 cd

– 옵션

a?b	ab 혹은 b

– 반복(1번 이상)

a+b	ab, aab, aaab, ….

– 반복 (0번 이상)

a*b	b or ab or aab or …

패턴 정의 예

- 예

```
nat           [0-9]+
signedNat     (+|-)?{nat}
foo|bar*   = (foo)|(ba(r*))
```

- Lex에서 사용되는 변수 및 함수
  - `yylex` Lex를 실행하는 함수
  - `yytext` 패턴에 매칭된 스트링을 담고 있은 변수
  - `yyin` Lex input file (default: `stdin`)
  - `yyout` Lex output file (default: `stdout`)
  - `ECHO` `print yytext` 혹은 `yyout`
  - `yyleng` 매칭된 스트링의 길이를 담고 있는 변수

YACC (Bison) 사용법

- 파일 이름 확장자 .y
- 사용되는 명령어

```
$ bison mylang.y                    (mylang.tab.c 파일 생성)
$ cc -o parser mylang.tab.c –ll     (실행 파일 parser 생성됨)
```

- Bison 구조

```
%{
%}      C declaration

        Bison declaration
%%
        Grammar rules
%%
        Additional codes
```

- 파일: yacc.y

```
%token NUMBER
%%
statement:    expression                          {printf("=%d\n", $1);}
;

expression:   NUMBER                              {$$=$1;}
              | expression '+' NUMBER             {$$=$1+$3;}
              | expression '-' NUMBER             {$$=$1-$3;}
;

%%
main ()
{
  yyparse ();
}
yyerror (s)    /* called by yyparse on error */
  char *s;
{
  printf ("%s\n", s);
}
```

- 파일: lex.l

```
%{
#include "yacc.tab.h"
extern yylval;
%}

%%
[0-9]+        {yylval = atoi(yytext); return NUMBER;}
[ \t]         ;
\n            return 0;
.             return yytext[0];
```

- 파서 만들기

```
$ bison -d yacc.y                (yacc.tab.h와 yacc.tab.c 생성)
$ flex lex.l                     (lex.yy.c 생성)
$ gcc -o parser yacc.tab.c lex.yy.c -ll (parser 생성)
$ parser
```

Makefile

- 여러 개의 파일들로 구성된 코드들을 관리함
  - 각 파일들 간의 종속성 (dependency)를 표현하고,
  - 한 파일이 수정된 경우, 그에 종속된 관련된 파일을 recompile 등의 과정을 거쳐 생성

- Make를 위한 파일 만들기 Makefile 혹은 makefile
  - 파일 간의 종속성 표현
  - 시간적으로 한 파일이 update된 경우 다른 파일도 update되어야 함.
  - 이때 어떤 명령어를 적용해야 하는 지를 표현함

- 명령어 실행하기 make

  make –k : error 발생시에도 계속 수행

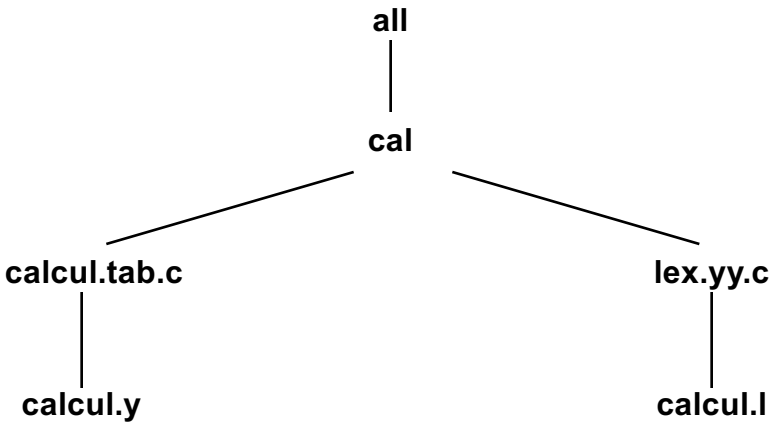
  make –n : 실제 수행되지 않고, Makefile의 shell이 어떻게 수행되는 지 만을 점검함

  make –f <filename> : Makefile이 아닌 다른 이름으로 작성된 파일을 이용하는 경우

예: Makefile (Lex, Yacc, C in Linux)

```
all: cal

cal: lex.yy.c calcul.tab.c
        gcc -o cal lex.yy.c calcul.tab.c -ll

lex.yy.c: calcul.l
        flex calcul.l

calcul.tab.c: calcul.y
        bison -d calcul.y

clean:
        rm -f lex.yy.c calcul.tab.c calcul.tab.h cal
```

Dependency Graph



`all` : make shell이 제일 처음 수행되는 곳

`make clean` : `clean` 부분을 수행시킴

Lex & Yacc 기호 관례

- 넌터미널은 영문 소문자
  - E.g. **`expr, stmnt`** …
- 터미널은 대문자
  - E.g. **NUMBER, NAME, …**
- token type
  - represents a class of syntactically equivalent tokens.
  - **yylex** 가 token 타입 생성
  - 모든 수는 동일한 토큰 타입 NUM 등으로 표현
  - 모든 변수 및 함수는 토큰 타입 ID 혹은 NAME 등으로 표현

```
%token <dval> NUM
```

  - Character token type (한 개의 문자로 구성된 토큰 인 경우)

```
'+'
```

yacc and lex의 interface

- Yacc이 yylex()를 호출하여 토큰을 생성시킴
- yylex 수행 결과 (토큰 타입, 관련된 값)을 yacc으로 전달
  - 관련된 값은 전역변수 yylval로 저장됨
  - 예: 토큰 타입 NUM 인 경우, 값 1234 을 yylval로 전달함.

- 변수 및 수에 대한 (토큰 타입, 값)

```
%union{
  double dval;
  int vblno;
}

%token <vblno> NAME          // 변수 이름에 대한 토큰 타입 및 값
%token <dval> NUMBER         // 수에 대한 토큰  타입 및 값
%type <dval> expression      // 수식에 대한 값
```

예제 (할당문을 갖는 계산기)

- Makefile 파일

```
LEX = flex
YACC = bison
CC = gcc
LIB = -lfl -lm

all: cal

cal: lex.yy.c calcul.tab.c
  $(CC) -o cal lex.yy.c calcul.tab.c $(LIB)

lex.yy.c: calcul.l
  $(LEX) calcul.l

calcul.tab.c: calcul.y
  $(YACC) -d calcul.y

clean:
  rm -f lex.yy.c calcul.tab.c calcul.tab.h *.exe *~
```

- Lex 파일

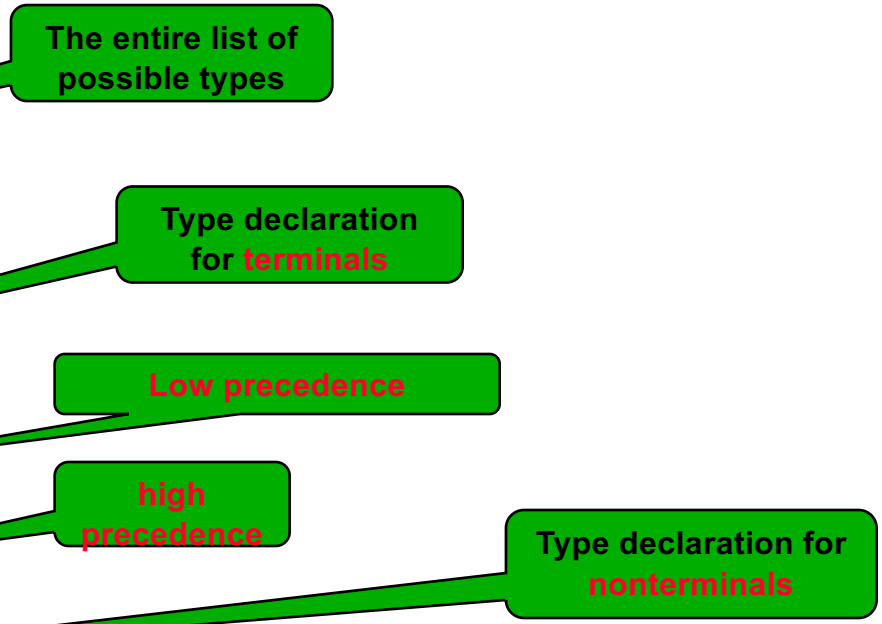
```
%{
#include "calcul.tab.h"
#include "hdr.h"
#include <math.h>
%}

%%
([0-9]+|([0-9]*\.[0-9]+)([eE][+-]?[0-9]+)?)   {    /* 정수 및 실수 */
      yylval.dval = atof(yytext); return NUMBER; }
[ \t]              ;                              /* whitespace */
[A-Za-z][A-Za-z0-9]*     {                        /* 변수를 테이블에 등록 */
    yylval.symp = symlook(yytext);
    return NAME; }
"$"                {return 0; }                   /* 라인 끝*/
\n       |
.                  return yytext[0];
```

- 헤드파일: hdr.h (Symbol Table)

```
#define NSYMS 20      /* 최대 심볼 수*/

struct symtab {
    char *name;
    double value;
} symtab[NSYMS];

struct symtab *symlook();
```

- Yacc 파일

```
%{
#include "hdr.h"
#include <string.h>
#include <stdio.h>
int yylex(void);
void yyerror(char *s);
%}

%union {
   double dval;
   struct symtab *symp;
}

%token <symp> NAME
%token <dval> NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%type <dval> expression
```



```
%%
statement_list:  statement '\n'
        | statement_list statement '\n'
;

statement:  NAME '=' expression          {$1->value = $3;}
            | expression                 {printf(" = %g\n", $1);}
;

expression: expression '+' expression    {$$=$1+$3;}
        | expression '-' expression      {$$=$1-$3;}
        | expression '*' expression      {$$=$1*$3;}
        | expression '/' expression
          {if ($3 == 0.0) yyerror ("divide by zero");
           else $$=$1/$3;}
        | '-' expression %prec UMINUS    {$$=-$2;}
        | '(' expression ')'             {$$=$2;}
        | NUMBER
        | NAME                           {$$=$1->value;}
;
```

```
%%
main(){
    yyparse();
}
```

// 주어진 변수를 테이블에서 찾아 해당 index를 return. 없으면 새로 등록

```
look up a symbol table entry, add if not present
  struct symtab *symlook(char *s) {
    char *p; struct symtab *sp;

  for (sp = symtab; sp < &symtab[NSYMS]; sp++) {
    // 이미 등록되어 있는가? 있으면 해당 index (sp) 를 return
    if(sp->name && !strcmp(sp->name, s))
       return sp;
    // 등록이 되지 않았으므로, 빈 공간에 등록
    if(!sp->name) {
      sp->name = strdup(s);
      return sp;
    }
  }
  yyerror ("Too many symbols");
}
```