Lex & Yacc(소개와 예제 중심으로)

박 성구 byosmosis@yahoo.com 1997년 11월 1일

이 글은... 언젠가 누군가를 위해 리포트를 대신 해주느라고 작성했었던 문서입니다. ^^; 텍스트에서 볼 수 있도록 바꾸어서 게시합니다. lex & yacc을 배워보세요. :) Lex와 Yacc는 입력구문의 검색과 분석을 용이하게 하기 위하여 개발되었다. 입력에 대한 어떤 패턴을 갖어야 하는 응용 프로그램을 만들 때 좀더 빠르고 정확하게 그리고 약간의 수정으로 유지보수가 쉽게 되는 것이 lex와 yacc의 특징이다. Lex와 Yacc가 쓰이는 예로는 숫자와 연산자를 구별해줘야 하는 계산기, 컴파일러나 인터프리터 언어의 문장 검색, 문법 분석을 위한 것이나, 데이타 베이스에서의 SQL언어의 문법을 체크하는 것, lex만을 이용한 구분 분석 등의 여러가지 경우에 이용된다. 여기서는 lex와 yacc의 전반적인 소개와 lex와 yacc를 이용하는 법 그리고 약간의 lex와 yacc의 응용 등에 대해서 소개하고자 한다.

목 록

1 Lex & Yacc

lex와 yacc는 벨연구소(Bell Laboratories)에서 70년대에 개발되었고, 후에 이것들은 유닉스의 표준 유틸리티로 선정이 되어 System V와 초기 BSD 버전은 원래의 AT&T 버전이 이용되고 있으며 새로운 BSD버전으로는 flex나 Berkely yacc이 나와 이용되고 있다. 또 FSF(Free Software Foundation)의 GNU 프로젝트의 bison이라 불리는 것은 yacc에 비해 좀더 향상된 기능을 가졌으며 BSD와 GNU 프로젝트의 flex(Fast Lexical Analyzer Generator)외에도 여러 운영체제하에서 여러 가지 다른 이름으로 lex와 yacc의 변형이 지원되고 있다.

※ MS-DOS나 OS/2 등의 OS상에서도 이용할 수 있는 lex와 yacc를 만든 곳으로 MKS(Mortice Kern Systems Inc.)가 있다.

1.1 Lex의 사용(Lex를 이용한 간단한 프로그램)

예제) Lex는 아래와 같이 %%와 %%사이에는 lex가 기술되고 그 이후에는 C가 기술된다.

```
%%
.|\n    ECHO;
%%

main()
```

```
{
        yylex();
}
```

위는 표준입력을 표준출력으로 내보내는 가장 간단한 lex 프로그램으로 정규 표현 .와 \n으로 표시되는 문자가 들어왔을 시 그대로 에코해준다. (실제로 lex에서 위의 내용은 default이다.)

유용한 다른 간단한 예로 다음과 같은

```
 [\t ]+          ;       # whitespace를 무시한다
```

를 %%와 %%사이에 포함시키면 탭이라든가 하나 이상의 공백이 있는 경우 C에서와 마찬가지로 아무 처리도 안해주는 ';'를 사용하였기 때문에 아무 처리도 하지 않는다.(화면에 같은 메시지를 에코조차도 하지 않는다.)

위의 소스를 simple.l이라고 하면(밑줄 표시는 사용자 입력)

```
[ post ] % lex simple.l
                           # lex를 C언어로, lex.yy.c파일 생성
[ post ] % cc lex.yy.c -o simple -ll
                           # lex라이브러리 포함 컴파일
[ post ] % ./simple < simple.l
                           # simple.l을 표준입력으로 하는 프로그램 실행
%%
.|\n ECHO;
%%

main()
{
        yylex();
}
```

위와 같이 되어 UNIX의 cat과 유사한 프로그램이 완성된다.(인자가 없는 cat 명령의 사용시 실제로 위의 프로그램과 같다)

※ lex의 구성은 다음 세가지 definitions, rules, user code section으로 이루어진다.

```
definitions
  # rules에 pattern을 간단하게 하기 위한 선언과 초기 조건
  # C코드 삽입시 %{, }% 기호를 표시하고 사이에 쓸 수 있음
%%
rules
  # pattern과 action으로 이루어짐
  # C코드는 {, }로 감싸서 표시함
%%
user code
  # yylex()함수와 사용자가 원하는 C 루틴으로 이루어짐
```

예제)

또 %%와 %%사이에 lex를 기술하기 이전에 C를 기술하기를 원할시에는 %{와 %}로 C의 기술이라는 표시를 하고 그 사이에 기술할 수 있다.

```
%{
        /*
         * My favorite...
         */
```

```
%}
%%

[\t ]+  ;
rain |
rose            { printf("%s!! That is my favorite.\n", yytext); }
love |
story           { printf("%s!! I like a love story. ^_~\n", yytext); }
"potato chip" |
potato |
chip  { printf("%s...!! now and then, I have a potato chip.\n", yytext); }

[A-Za-z]+ { printf("%s..., what is that?\n", yytext); }

.|\n    { ECHO; }

%%

main()
{
        yylex();
}
```

yytext는 lex에서 사용되는 입력 텍스트에 대한 global character pointer이다. 위의 소스를 simple.l과 같은 방법으로 컴파일하여 실행시켜 보면

```
[ post ] % ./favor
potato chip
potato chip...!! now and then, I have a potato chip.

love7
love!! I like a love story. ^_~
7
distress
distress...,  what is that?
위와 같이 된다.

lovestory
lovestory...,  what is that?

love story
love!! I like a love story. ^_~
story!! I like a love story. ^_~
```

위에 나타난 수행결과의 출력과 같이 처리하기를 원하는 패턴에 대해서 처리가 이루어진다. love7의 경우에는 love에 대한 처리를 해주고 7은 알파벳이 아니므로 다음줄에 에코가 된다. 또 결과중 바로 위의 'love story'는 love를 인식하고 whitespace로 빈칸을 인식하고 다음에 story를 인식한 것이다. 따라서 love에 대한 처리와 빈칸에 대한 아무 처리없음과 story에 대한 처리가 결과 내용이 될 것이다. 위의 예로서 loveyou라고 입력을 했다면 love 다음에 you를 따로 보는 것이 아니라 하나의 패턴으로 보아 distress와 같은 결과가 나올 것임을 예상할 수 있다.

예제) 다음은 전의 예제와 같은 간단한 형식으로 다만 숫자에 대한 C의 enum형식으로 선언하여 심볼 테이블의 값을 이용하는 예를 보여준다.

```
[ post ] % cat number.l
```

```
%{
enum {
        NONE = 0,
        ONE,
        TWO,
        THREE,
        FOUR,
        FIVE,
        SIX,
 SEVEN
};

int number;
void action();
%}

%%
oneone  { number = SEVEN; action(); }
one |
ONE  { number = ONE; action(); }
two |
TWO  { number = TWO; action(); }
three |
THREE  { number = THREE; action(); }
four |
FOUR  { number = FOUR; action(); }
five |
FIVE  { number = FIVE; action(); }
six |
SIX  { number = SIX;  action(); }
[a-zA-Z]+ { number = NONE; action(); }
%%

void action()
{
        if (number != NONE)
                printf("%s is %d\n", yytext, number);
        else
                printf("%s?? I don't learn it.\n", yytext);
}

main()
{
        yylex();
}

[ post ] % ./number
tow
tow?? I don't learn it.

five
five is 5

four7
four is 4
```

```
7
SIX
SIX is 6

ten
ten?? I don't learn it.

oneone
oneone is 7

twotwo
twotwo?? I don't learn it.

two two
two is 2
 two is 2
```

위의 마지막 'two two'는 two에 대한 처리를 해주고 ' '의 빈칸은 다음 줄에 단순히 에코해주고 다음줄로 넘어가지 않고 다음의 two가 처리되어 나타난 결과이다. 따라서 두 번째의 two에는 앞에 빈칸이 하나 더 있다. 그리고 oneone이라는 패턴이 존재할 때에는 one의 패턴이 아닌 oneone의 패턴을 따르는 것은 긴 문자열이 우선순위를 가지기 때문이다.(정규 표현식 참조)

예제) 다음은 lex의 위에 제시한 것보다 약간 복잡한 숫자와 연산자와 변수를 인식하는 예제로서 조금만 개선시키면 완전한 계산기 프로그램이 될 수 있다. 이 예제는 단순히 숫자인지, 연산자인지, 예약어 또는 변수로서 사용될 값인지를 출력하여 나타내도록 하는 예제이다.

```
%{
/* need this for the call to atof() below */
#include <math.h>
%}

DIGIT    [0-9]  /* 매크로를 정의한다 */
ID       [a-z][a-z0-9]*

%%
{DIGIT}+  { printf( "An integer: %s (%d)\n", yytext, atoi( yytext ) ); }
{DIGIT}+"."{DIGIT}* { printf( "A float: %s (%g)\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]*"}"  ; /* { } 안의 내용은 주석으로 무시한다. */
[ \t\n]+  ; /* whitespace를 무시한다. */
.   printf( "Unrecognized character: %s\n", yytext );
%%

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

```
        yylex();
    }
```

위의 예에서 yyin에 대한 지정은 입력을 커맨드 라인상의 첫인자가 가리키는 파일의 내용을 처리할 것이지 화면에서 표준입력을 받아 처리할 것인지에 대한 처리를 할 수 있게 한다.

1.2 Lex와 Yacc의 관계(Lex scanner, Yacc parser)

Lex scanner와 Yacc parser는 같이 구현하는 경우가 대부분인데, Yacc가 Lex의 상위에서 구현된다. Lex는 입력 문자열에 대한 일차적인 검색을 하고 실제적인 분석은 Yacc가 하는 것이다. Yacc는 입력에 대한 토큰(token)이 필요하면, Lex에서 제공하는 yylex()함수를 호출하여, 입력된 토큰들의 배열이 주어진 문법에 맞는지를 체크하면서 그 조건에 맞는 실행을 하게 된다. 다르게 표현하면, Lex와 Yacc을 사용 동시에 사용할 시에는 yacc 기술파일의 main()함수에서 yyparse()함수라는 yacc에 의해 만들어지는 구문분석기를 부르고, yyparse()함수는 yylex()라는 lex가 만들어 주는 해석기(lexer)를 이용해서 입력열에서 처리단위의 토큰을 뽑아오게 된다. 예를 들어 lex에서 '[\t]+'로 표시되었던 공백이나 탭가 같은 whitespace문자는 대부분의 프로그램에서 실제로 처리하기를 원치 않는다. 따라서 위와 같은 의미 없는 문자나 처리할 필요가 없는 문자열에 대해서는 그 내용을 parser에게 넘겨줄 필요가 없다. 그리고 다음에 오는 처리할 문자가 있을 경우 그 내용(토큰)을 yacc parser에게 넘기면 그 처리를 할 것이다. 이런 경우가 lex와 yacc를 같이 쓰게 되는 간단한 예로 생각될 수 있을 것이다.

※ MS-DOS나 OS/2 등의 OS상에서도 이용할 수 있는 lex와 yacc를 개발했던 MKS(Mortice Kern Systems Inc.)가 WWW과 관련하여 Netscape의 Fasttrack이나 Microsoft의 IIS에 대한 검색 엔진 등에 이용할 수 있게도 하였다.

1.3 Yacc의 사용

Yacc는 Yet Another Compiler Compiler의 약자로, 컴파일러 생성을 위한 프로그램을 compiler-generator 또는 compiler-compiler라고 부르는데, Yacc가 BNF와 같은 형식의 rules의 항목들로부터 parser를 만들어내는 프로그램이다.(Yacc 입력은 BNF를 간단하게 만든 버전이다.) Yacc를 이용한 프로그램들은 portable C 컴파일러, APL, Pascal, Ratfor, tbl, eqn 등이 있다.

다음은 Yacc의 구조이며, Yacc는 Lex와 비슷한 구조를 가지고 있는데 실제로는 Lex가 Yacc의 구조를 본따 쓴 것이다.

```
definitions
   # C코드 삽입시 %{, }% 기호로 표시하고 사이에 쓸 수 있음
%%
rules
   # 각 rule은 "LHS: RHS;"와 같은 형식으로 이루어짐
%%
user code
   # lex와의 결합시 yylex()를 이용한다.

※ LHS는 left-hand symbol의 약자로 ':'의 왼쪽에 오는 statement,
   expression과 같이 왼쪽에 오는 기호를 말하며,  RHS는 right-hand
   symbol의 약자로 ':'의 오른쪽에 오는 기호를 말한다.
   각 rule의 끝은 ';'으로 표시한다.
```

예제) yacc의 이해를 위한 간단한 단지 덧셈과 뺄셈만이 가능한 아주 간단한 기능을 가진 계산기를 만든다고 할 때 그것의 실제 실행 예를 보이면 아래와 같다.

```
[ post ] % cat y_calc.y
```

```
%token NAME NUMBER
%%
statement:    NAME '=' expression
            | expression  { printf("= %d\n", $1); }
            ;
expression:   expression '+' NUMBER { $$ = $1 + $3; }
            |   expression '-' NUMBER { $$ = $1 - $3; }
            |   NUMBER  { $$ = $1; }
            ;
[ post ] % cat y_calc.l
%{
#include  "y.tab.h"
extern int  yylval;
%}
%%
[0-9]+ { yylval = atoi(yytext); return NUMBER;  }
[\t ]+ ;
\n return 0;    /* 논리적인 입력처리열의 끝 */
. return  yytext[0];
%%
```

yacc가 lex를 호출하면(=yylex()를 호출하면) NAME, NUMBER, 그리고 +, -, =의 형태로 토큰을 넘겨 받는다.(정규식 표현에 의해서 ' '안에 놓여진 문자는 그 문자 그 자체로 본다) 또 y_calc.y를 보면, $1, $2, $3는 각각 입력받은 토큰의 위치에 해당하는 값을 가지고 있고, 이 값은 yacc에서 yylval가 type int, 정수로 처리가 되어 %d로 정수를 출력한다.(y_calc.l에서 사용하고 있는 yylval이라는 변수는 union으로 선언되어 있다. 이것을 lex에서 외부변수로 만들어 이용하여 yyparse()함수를 호출하는 yacc가 처리하도록 값을 전달하게 되는 것임) 그리고 $2의 경우에 해당하는 것은 연산자에 해당하는 문자가 들어가기 때문에 expression의 RHS의 { } 안에서 +, 와 -로 이용하고 있다. 그리고 $$는 LHS의 값, expression의 값이 된다.

yacc를 실제로 사용하기 위해서 example.y라는 yacc파일이 작성되면 yacc를 이용하여, yyparse()라는 구문분석기 함수를 가진 y.tab.c라는 C언어 화일을 만들기 위해서 'yacc example.y'를 실행하고 C컴파일러를 이용해서, y.tab.o라는 오브젝트 화일을 만든 후에, 코드생성기나 다른 프로그램과 링크하여 원하는 결과를 만든다. 위의 프로그램(y_calc.y와 y_calc.l)을 컴파일하여 실행한 결과는 아래와 같다.

```
[ post ] % yacc -d y_calc.y /* y.tab.h와 y.tab.c가 만들어진다 */
[ post ] % lex y_calc.l
[ post ] % cc -o y_calc y.tab.c lex.yy.c -ly -ll
[ post ] % ./y_calc
1+101
= 102
[ post ] % ./y_calc
1000-300+200+100
= 1000
```

여기서는 y.tab.h를 lexer에서 include하기 때문에, yacc에서 만들어주기 위해 -d 스위치를 이용하였다. lex를 이용하여, lex.yy.c를 만들고, yacc을 이용하여, y.tab.c를 만들어서 liby.a와 libl.a를 링크하여 원하는 실행화일을 만들게 된다.

※ Yacc에서의 모호성 - 다음과 같은 예제를 생각해보자.

```
expression : expression '+' expression
    |        expression '-' expression
    |        expression '*' expression
    |        expression '/' expression
    |        '-' expression
```

```
        |       '(' expression ')'
        |       NUMBER
```

위의 yacc 문법을 가지고, 2+3*4를 파싱하려고 하면, 그림과 같이, (2+3)*4로 파스트리를 구성할 수도 있고, 2+(3*4)로 처리될 수도 있게 되어, 문제가 생기게 된다. 또한 2-3-4-5의 경우에도, 2-(3-(4-5))인지, (2-3)-(4-5)인지 문제가 생기게 된다 따라서 애매하지 않은 문법으로 바꿀 수가 있는데, 우선순위와 결합법칙을 문법에 감안하는 방법으로 한가지는 위의 문제가 생기지 않게 쓰는 것이고, 다른 한가지는 이런 문제를 해결하기 위하여 yacc에서 지정되어 있는 사용하는 명령을 이용하여(%prec, %left, %right, ...) 우선순위와 결합법칙을 지정하는 것이다. 이를 해결하는 방법에는 문법을 바꾸는 경우의 기본적인 아이디어는 우선순위를 높게 가지는 연산자에 대해서는 LHS에 오게 만들어서, 미리 reduce되도록 하는 것이며, 결합법칙에 대해서도 마찬가지로 문제를 해결할 수 있다. 앞의 문법을 바꾸어서 shift/reduce conflict를 해결한 것은 아래와 같다.

```
expression: expression '+' mulexp
  |   expression '-' mulexp
  |   mulexp
  ;
mulexp : mulexp '*' primary
  |   mulexp '/' primary
  |   primary
  ;
primary:  '(' expression ')'
  |   '-' primary
  |   NUMBER
  ;
```

문제를 해결하는 또다른 방법으로 yacc에서 제공하는 명령어를 이용하는 방법으로 이 명령어를 이용해서, 우선순위 및 결합법칙을 제공할 수 있게 된다. 몇가지 유용한 명령의 예를 들어보면 다음과 같다.

```
%left  '+' '-'
%left  '*' '/'
%right '='
%nonassoc  UMINUS /* unary operator의 정의로 결합법칙이 적용이 안된다.  */
```

위와 같이 기술하게 되면, + - * / = 의 우선순위는 + - 가 가장 낮고, * /가 그것보다 높고, 대입연산자 = 이 높고, 부호연산자인 -가 가장 높은 우선순위를 가진다는 것을 보여주고 있다. 또한 +나 -, *, /는 왼쪽으로 결합법칙을 가지고, 대입연산자 =는 우측으로 결합법칙을 따른다.

```
%token NAME NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS
expression:     expression '+' expression
        |       expression '-' expression
        |       expression '*' expression
        |       expression '/' expression
        |       '-' expression %prec UMINUS
        |       '(' expression ')'
        |       NUMBER
        ;
```

위에서 중간에 보이는 %prec UMINUS는 '-' expression과 같은 문자열을 파싱하게 되면 -에 대해서는 UMINUS로 지정된 우선순위를 사용하라는 의미이다. 따라서, 모호한 문법에서 보이는 shift/reduce conflict가 생길때, 우선순위와 결합법칙을 적용함으로써, 쉽게 해결할 수가 있게 된다.

※ 또다른 Yacc의 모호성, LALR(1)이 아닌 문법의 예 다음과 같은 문법을 살펴보자.

```
phrase: cart_animal AND CART
       | work_animal  AND PLOW
cart_animal: HORSE | GOAT
work_animal: HORSE | OX
```

위의 예는 들어, HORSE AND CART 라는 입력이 들어왔을 경우에, HORSE를 cart_animal로 reduce해야 하는지, work_animal로 reduce해야 하는지 알려면, HORSE를 보고, AND를 보고, CART를 보고 나서야 알 수가 있게 된다. Yacc가 처리할 수 없는 이 문제는, yacc가 구문 분석을 하는데 사용하는 기술은 LALR(1)인데 LALR은 LookAhead Left Recursive, 즉, 한 개의 단어를 더 보고 맞는 문법을 결정할 수 있다는 것이기 때문이다. 따라서, HORSE 다음에 2개의 단어를 더 읽어야 하는 경우인데, 이 경우 LALR(2)문법이며, yacc는 한 개의 lookahead만을 볼 수 있는 문법인 LALR(1)에 해당하므로, 위의 문법은 처리할 수 없다. 따라서 위의 문법은 다음과 같이 LALR(1)문법으로 바꾸어 주어야 한다.

```
phrase -> cart_animal  CART
        | work_animal  PLOW
cart_animal  -> HORSE | GOAT
work_animal -> HORSE | OX
```

위의 문법은 HORSE를 보고, 바로 뒤의 한 단어인 CART만을 보면 cart_animal로 reduce할지, work_animal로 reduce해야할지를 알게 된다.

예제) 다음과 같이 변수 처리가 가능한 탁상 계산기를 구현하려고 한다.

```
a = 2.3 + 3.5
b = 3.5 + 5.2
c = a + b - 1
c
= 13.5
```

위와 같은 기능을 수행하려면 기본적으로 기억장소로 사용될 수 있는 심볼테이블을 유지하고, 심볼테이블에 변수의 이름과 값을 유지해야 한다. 다음은 변수이름이 영문의 알파벳만을 가진다고 가정한 그러한 예를 보인 것이다.

```
[ post ] % cat variable.y
%{
double vbltable[26];  /* double형의 기억장소 배열 */
%}
%union  {
                double dval;
                int vblno;
        }
%token    <vblno> NAME
%token    <dval> NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS
%type <dval> expression
%%
statement_list: statement '\n'
          |         statement_list statement '\n'
          ;
statement:         NAME '=' expression  { vbltable[$1] = $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       { $$ = vbltable[$1]; }
           ;
%%
main()
{
    yyparse();
}
```

변수이름이 영문의 알파벳만을 가진 26개의 원소를 가진 배열인 vbltable[26]을 선언해서 변수의 값을 유지하도록 한다. 위에서 %union은 lexer에서 받는 token인 NAME과 NUMBER를 NAME은 vbltable[]의 한 인덱스를 지칭하게 하여, 0부터 25의 값을 가지게 하는 반면에, NUMBER는 실수의 값을 가지도록 한다. 따라서, 같은 token이라고 하더라도, 가지는 값이 다르게 되는데, 가지는 값은 달라도, 한 순간에는 오직 한 개의 token만이 인식되기 때문에, 같은 기억장소를 유지하도록 위와 같이 %union으로 선언하게 된다. 그리고 %union문은 실제, y.tab.c안에서 C의 union으로 변환되어 처리된다. “%type <dval> expression” 문장은 역시, token뿐이 아니라 token으로 정해지지 않은 LHS의 경우에도 알맞은 타입으로 설정이 가능하다는 것을 보여준다. “NAME '=' expression { vbltable[$1] = $3; }”의 문장은 왼편의 문법으로 reduce될 때 오른편의 action부를 실행하게 되는데, 의미는 $1 즉 NAME의 값(0부터 25사이의 값을 yylex()로부터 받았다)으로 vbltable의 한 원소에 $3 즉 expression의 값을 넣는 것이다. 예를 들어 x = 2.0 + 3.2이었으면 x에 대한 $1은 23이 되며 vbltable[23] = 5.2의 의미를 가진다. “'-' expression %prec UMINUS { $$ = -$2; }”의 문장은 -2와 같은 단어가 들어왔을 때 -는 2-3에서의 -가 아니라 %prec에 의해 부호연산자인 -로 우선 순위를 두어 연산이 이루어짐을 의미한다. “NAME { $$ = vbltable[$1]; }”은 만약 y라는 변수이름이 주어지면 $1의 값이 0부터 25사이의 값 중에서 24번째 해당되므로 vbltable[24]의 값을 스택에 저장하게 된다.

```
[ post ] % cat variable.l
%{
#include "y.tab.h"
#include <math.h>

extern double vbltable[26];
%}
%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)  {
                 yylval.dval = atof(yytext); return NUMBER;
                }
[\t]   ;        /* ignore white space */
[a-z]    {      yylval.vblno = yytext[0] - 'a'; return NAME; }
"$"      { return 0; }    /*  end of input  */
\n       |
.                return yytext[0];
%%
```

위의 lex화일을 보면, yylval.dval로 표현한 것이 있는 데, 이것은 yylval가 union타입으로 선언이 되어있으므로, NUMBER형이면, 값을 넣을때에도, yylval.dval식으로 하여, 넣으라는 것을 나타낸다. 또한 변수가 영문 알파

벳만을 가정하기 때문에, 단순히 0부터 25사이의 값으로 vbltable을 인덱싱하게 되는데, lex에서의 yylval.vblno = yytext[0] - 'a'는 yylval에 0부터 25사이의 값을 넣게 된다. 위의 프로그램(variable.l 과 variable.y)을 컴파일하여 실행한 결과는 아래와 같다.

```
[ post ] % yacc -d variable.y
[ post ] % lex variable.l
[ post ] % cc -o variable y.tab.c lex.yy.c -ll
[ post ] % ./variable
7.5+3.5
= 10
200/20+20/2
= 20
x=20/2
y=x+1
y
= 11
```

1.4 Lex와 Yacc의 이용

다음은 실제로 Lex와 Yacc를 동시에 결합하여 사용한 예이다.

```
[ post ] % cat example.h
#define NOUN 257
#define PRONOUN 258
#define  VERB  259
#define  ADVERB  260
#define  ADJECTIVE  261
#define  PREPOSITION   262
#define  CONJUNCTION 263
[ post ] % cat example.l
%{
/*
 * build a lexical analyzer to be used by a high-level parser.
 */
#include "example.h" /* token codes from the parser */
#define LOOKUP 0 /* default - not a defined word type.  */
int state;

%}
%%

\n { state = LOOKUP; }
\.\n { state = LOOKUP; return 0; }  /*  논리적인 처리단위의 끝을 의미 */
^verb { state = VERB;  }
^adj { state = ADJECTIVE; }
^adv { state = ADVERB; }
^noun { state = NOUN; }
^prep { state = PREPOSITION; }
^pron { state = PRONOUN; }
^conj { state = CONJUNCTION; }
[a-zA-Z]+ {
            if (state != LOOKUP) {
                  add_word(state, yytext);
            } else {
```

```
                  switch(lookup_word(yytext))
                  {
                      case VERB  : return(VERB);
                      case ADJECTIVE : return(ADJECTIVE);
                      case ADVERB  : return(ADVERB);
                      case NOUN  : return(NOUN);
                      case PREPOSITION : return(PREPOSITION);
                      case PRONOUN : return(PRONOUN);
                      case CONJUNCTION : return(CONJUNCTION);
                      default    : printf("%s: don't reconize\n", yytext);
                                           /*  don't return just ignore it  */
                  }
            }
          }
.   ;

%%
/* 한 단어 타이프에 대하여 단어 여러개를 갖기 위한 구조체 선언 */
struct word {
        char *word_name;
        int  word_type;
        struct word *next;
};
struct word *word_list;
extern void *malloc();

int add_word(type, word)
int type; char *word;
{
        struct word *wp;
        if (lookup_word(word) != LOOKUP) {
                printf("!!! warning: word %s already defined \n",word);
                return 0;
        }

        /*  word not there, allocate a new entry and link it on the list */
        wp = (struct word *) malloc(sizeof(struct word));
        wp->next = word_list;
        /* have to copy the word itself as well */

        wp->word_name = (char *)malloc(strlen(word)+1);
        strcpy(wp->word_name, word);
        wp->word_type = type;
        word_list = wp;
        return 1;    /* it worked */
}

int lookup_word(word)
char *word;
{
        struct word *wp = word_list;
        /*  search down the list looking for the word  */
        for (; wp; wp = wp->next) {
                if (strcmp(wp->word_name, word) == 0)
                        return wp->word_type;
```

```
        }
        return LOOKUP; /*  not found  */
}
```

verb, noun 등에 대해서 state를 이용하여, 줄이 끝나기 전까지 나오는 단어들을 심볼테이블에 등록하게 되는 것이 add_word()함수이고 그 등록된 단어를 살펴보는 함수가 lookup_word()이다.

다음은 yacc화일 example.y화일의 내용이다.

```
[ post ] % cat example.y
%{
 /*
  A lexer for basic grammar to use for recognizing english sentences.
  */
#include <stdio.h>
%}
%token  NOUN PRONOUN VERB ADVERB ADJECTIVE PREPOSITION CONJUNCTION
%%
sentence: subject VERB object { printf("Sentence is valid.\n"); }
  ;
subject:  NOUN
  | PRONOUN
  ;
object:  NOUN
  ;
%%

extern FILE *yyin;
main()
{
  while(!feof(yyin)) {
     yyparse();
  }
}

yyerror(s)
char *s;
{
  fprintf(stderr,"%s\n",s);
}
```

위의 yacc화일에서 "%token NOUN PRONOUN VERB ADVERB ADJECTIVE PREPOSITION CONJUNCTION"문은 입력 기본단위인 토큰으로 처리한다는 것을 의미하고 여기서는 lexer에게서 넘겨받는 것이 된다. 즉, yyparse()가 매번 yylex()를 부르면, 그 결과는 NOUN이나 PRONOUN 등의 token으로 오게 되는 것이다. 따라서, yyparse는 그러한 토큰의 나열이 문법에 맞는지를 분석하여, 맞으면 옆에 주어진, action인 C언어 문장을 수행하게 된다. yacc의 입력은 yyin이라는 화일에서 들어오게 되고, 출력은 yyout이라는 화일로 쓰게 된다. 기본적으로는 yyin과 yyout은 stdin과 stdout, 즉 표준 입,출력으로 설정이 되어있으므로, 키보드와 화면을 통해 입출력이 이루어지게 된다. 이를 바꾸어주는 것은 앞에서 제시하였었다. yyparse()는 yacc가 만들어주는 구문분석기(parser)이고, yyerror()는 구문분석 중에 에러가 생겼을 때 부르는 함수이다. 다음은 앞에서 설명한 lex화일(example.l)과 yacc화일(example.y)을 실제 컴파일하여 수행한 결과이다.

```
[ post ] % lex example.l
[ post ] % yacc example.y
[ post ] % cc -o example lex.yy.c y.tab.c -ll
```

```
[ post ] % ./example
noun pig fish I
verb run am are
verb outrun
pig outrun fish
Sentence is valid.
noun you
you are pig
Sentence is valid.
pig are
syntax error
syntax error
pig outrun fish.
Sentence is valid.
verb am
!!! warning: word am already defined
I am fish
Sentence is valid.
i am fish
i: don't reconize
syntax error
```

1.5 Makefile의 이용

컴파일 절차를 간소화하고, 프로그램 보수를 쉽게 하기 위해서 다음과 같이 makefile를 정의하여 편리하게 컴파일을 수행할 수 있다.

```
[ post ] % cat makefile
CC = cc
LIBS = -ll
LEX = lex
YACC = yacc
CFLAGS = -DYYDEBUG=1
all: speech
speech: y.tab.c lex.yy.c
                $(CC) -o speech y.tab.c lex.yy.c $(LIBS)
y.tab.c: example.y
                $(YACC) -d example.y
lex.yy.c : exaple.l
                $(LEX) example-07.l
[ post ] % make
cc -o speech y.tab.c lex.yy.c -ll
```

2 참고도서

- lex & yacc - O'Reilly & Associates. Inc.