

YACC 응용 예

Desktop Calculator

Lex 입력

□ 수식문법을 위한 lex 입력: calc.l

```
%{
#include "calc.tab.h"
}%
%%
[0-9]+    return (NUMBER) ;
[ \t]     ;
\n        return (0) ;
\+        return ('+') ;
\*        return ('*') ;
.         { printf("'%c': illegal character\n",
                yytext[0]); exit(-1); }
```

Yacc 입력

□ 수식문법을 위한 yacc 입력: calc.y

```
%{
#include <stdio.h>
%}
%token NUMBER
%%
Exp : Exp '+' Term      { printf("rule 1\n"); }
    | Term              { printf("rule 2\n"); }
    ;

Term : Term '*' Num     { printf("rule 3\n"); }
    | Num               { printf("rule 4\n"); }
    ;

Num : NUMBER            { printf("rule 5\n"); }
    ;
```

실행

□ UNIX 명령어

```
% gedit calc.l
% gedit calc.y
% flex calc.l
% bison -d calc.y
% gcc -o calc lex.yy.c calc.tab.c -ly -ll
% gedit input
% ./calc < input
```

확장 - 0

□ 입력

- ❖ 하나의 식
- ❖ 입력의 끝은 '\$' 문자로 표현
- ❖ 빼기(-), 나누기(/), 그리고 괄호 연산 지원

□ 출력

- ❖ 입력된 식의 계산결과를 출력

Lex 입력

```
%{  
#include <stdlib.h>  
#include "calc.tab.h"  
extern int yylval;  
%}  
%%  
[0-9]+      {yylval = atoi(yytext); return(NUMBER);}  
[ \t]       ;  
"$"         {return 0; /* end of input */ }  
\n         |  
.           {return(yytext[0]); }
```

Yacc 입력

%{

```
%{  
#include <stdio.h>  
%}
```

```
%token NUMBER
```

```
%%
```

```
Goal: Exp                                {printf("=%d\n", $1);}
```

```
;
```

```
Exp : Exp '+' Term                       {$$=$1+$3;}
```

```
    | Exp '-' Term                       {$$=$1-$3;}
```

```
    | Term                               {$$=$1;}
```

```
;
```

```
Term : Term '*' Fact                     {$$=$1*$3;}
```

```
    | Term '/' Fact                       {$$=$1/$3;}
```

```
    | Fact                               {$$=$1;}
```

```
;
```

```
Fact: NUMBER                             {$$=$1;}
```

```
    | '('Exp')'                           {$$=$2;}
```

```
;
```

확장 - 1

□ 입력

- ❖ 하나의 식에서 여러 개의 식으로 확장
- ❖ 한 줄에 하나의 식

□ 출력

- ❖ 입력된 식의 계산결과를 출력

Lex 입력

```
%{  
#include <stdlib.h>  
#include "calc.tab.h"  
extern int yylval;  
%}  
%%  
[0-9]+      {yylval = atoi(yytext); return(NUMBER);}  
[ \t]       ;  
"$"         {return 0; /* end of input */ }  
\\n         |  
.           {return(yytext[0]); }
```

Yacc 입력

```
%{  
#include <stdio.h>  
%}  
  
%token NUMBER  
  
%%  
  
ExpList: ExpList Exp'\n'      {printf("=%d\n", $2);}  
       | Exp'\n'              {printf("=%d\n", $1);}  
  
;
```

Yacc 입력

Exp : Exp '+' Term	{ \$\$=\$1+\$3; }
 Exp '-' Term	{ \$\$=\$1-\$3; }
 Term	{ \$\$=\$1; }
;	
Term : Term '*' Fact	{ \$\$=\$1*\$3; }
 Term '/' Fact	{ \$\$=\$1/\$3; }
 Fact	{ \$\$=\$1; }
;	
Fact: NUMBER	{ \$\$=\$1; }
 '('Exp')'	{ \$\$=\$2; }
;	

확장 - 2

□ 입력

- ❖ 변수 지원, 단 변수 이름은 하나의 소문자
- ❖ 입력되는 식에 변수에 대한 배정도 포함
- ❖ 피연산자는 실수도 가능

□ 출력

- ❖ 입력된 식들의 계산결과를 출력
- ❖ 배정문일 경우, 배정되는 값은 출력하지 않는다.

확장 - 2 의 예

입력

a = 10.0

a+1

b = a+10

b+a

출력

11.0

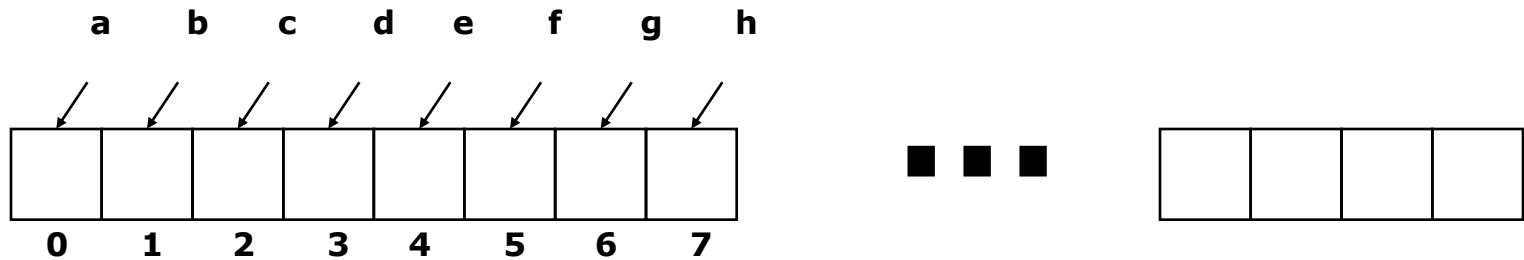
30.0

변수를 어떻게 처리?

□ 가능한 변수의 수는 26개임

- ❖ 크기 26인 배열을 사용
- ❖ 인덱스 0 에는 변수 a의 값, 인덱스 1 에는 변수 b의 값,
- ❖ 어휘분석 단계에서 변수를 인식하면, 토큰 Name 의 값으로 변수에 대한 인덱스를 전달

double vbltables[26];



토큰의 값

□ 토큰의 값

- ❖ yylval을 통하여 파서에게 전달

□ 토큰 값의 타입

- ❖ yylval의 타입
- ❖ YYSTYPE

□ 여러 타입의 값을 허용하려면

- ❖ C: union을 사용
- ❖ Yacc: %union을 사용

%union

□ 토큰 값의 종류

- ❖ 정수, 실수, 스트링, ...
- ❖ 여러 타입의 토큰을 파서에게 전달

□ Yacc에서 토큰의 타입을 정의

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

□ 각 토큰의 타입을 지정

```
%token <vblno> NAME  
%token <dval> NUMBER
```


문법 기호의 타입

□ 문법 기호의 타입

- ❖ %union에서 정의
- ❖ %type을 이용하여 지정

□ 예

```
%union {  
    double dval;  
    struct symtab *symp;  
}  
%token <symp> NAME  
%token <dval> NUMBER  
%type <dval> expression
```

Lex 입력

```
%{
#include <stdlib.h>
#include "calc.tab.h"
extern double vbltables[26];
}%
%%
([0-9]+ | [0-9]*\.[0-9]+)
        {yyval.dval = atof(yytext); return(NUMBER);}
[ \t]
        ;
[a-z]
        {yyval.vblno = yytext[0]-'a'; return NAME;}
"$"
        {return 0; /* end of input */ }
\n
        |
.
        {return(yytext[0]); }
```

Yacc 입력

```
%{  
#include <stdio.h>  
double vbltables[26];  
%}  
%union {  
    double dval;  
    int vblno;  
}  
%token <vblno> NAME  
%token <dval> NUMBER  
%type <dval> Fact Term Exp  
%%
```

Yacc 입력

StmtList: StmtList Stmt '\n'

| Stmt '\n'

;

Stmt : NAME '=' Exp {vbtables[\$1] = \$3;}

| Exp {printf("=%f\n", \$1);}

;

Exp : Exp '+' Term {\$\$=\$1+\$3;}

| Exp '-' Term {\$\$=\$1-\$3;}

| Term {\$\$=\$1;}

;

Term : Term '*' Fact {\$\$=\$1*\$3;}

| Term '/' Fact {\$\$=\$1/\$3;}

| Fact {\$\$=\$1;}

;

Fact : NUMBER {\$\$=\$1;}

| NAME {\$\$= vbtables[\$1];}

| '('Exp')' {\$\$=\$2;}

;

확장 - 3

□ 변수이름

❖ Single character → multiple characters

□ 예

입력

input = 10.0

result = input+1

result

result-input

출력

11.0

1.0

Symbol table

- 심볼테이블의 구조

```
//symbol.h
#define NSYMS 20 /* maximum number of symbols */
struct symtab {
    char      *name;
    double    value;
} symtab[NSYMS];

struct symtab *symlook();
```

- 어휘분석

- 변수를 인식하면 심볼테이블을 검색하여, 심볼테이블 항목의 주소를 리턴
- `symlook()` 함수를 이용

symlook() 함수

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

    for(sp=symtab; sp < &symtab[NSYMS]; sp++) {
        /* is it already here ? */
        if(sp->name && !strcmp(sp->name, s))
            return sp;
        /* is it free ? */
        if(!sp->name) {
            sp->name=strdup(s);
            return sp;
        }
        /* otherwise continue to next */
    }
    yyerror("Too many symbols");
    exit(1);
}
```

```
%{
#include <stdlib.h>
#include "symbol.h"
#include "calc.tab.h"
}%
%%
([0-9]+ | [0-9]*\.[0-9]+)
    {yylval.dval = atof(yytext); return(NUMBER);}
[ \t]
    ;
[A-Za-z][A-Za-z0-9]*
    {yylval.symp = symlook(yytext); return NAME;}
"$"
    {return 0; /* end of input */ }
\n
    |
.
    {return(yytext[0]); }
```


Yacc 입력

```
%{  
#include "symbol.h"  
#include <stdio.h>  
#include <string.h>  
%}  
%union {  
    double dval;  
    struct symtab *symp;  
}  
%token <symp> NAME  
%token <dval> NUMBER  
%type <dval> Fact Term Exp
```

Yacc 입력

```
%o%  
  
StmtList: StmtList Stmt '\n'  
        | Stmt '\n'  
        ;  
  
Stmt : NAME '=' Exp      { $1->value = $3;}  
      | Exp              { printf("=%f\n", $1);}  
      ;  
  
Exp  : Exp '+' Term      { $$=$1+$3;}  
      | Exp '-' Term     { $$=$1-$3;}  
      | Term             { $$=$1;}  
      ;  
  
Term : Term '*' Fact     { $$=$1*$3;}  
      | Term '/' Fact    { $$=$1/$3;}  
      | Fact            { $$=$1;}  
      ;  
  
Fact : NUMBER { $$=$1;}  
      | NAME   { $$=$1->value;}  
      | '('Exp')' { $$=$2;}  
      ;
```

확장 - 4

□ Allow mathematical functions

❖ Such as `sqrt()`, `exp()`, `log()`, ...

□ 예

입력

`s2 = sqrt(2)`

`s2`

`s2*s2`

출력

1.41421

2

Naïve approach

❑ In Yacc source

```
%token Sqrt LOG EXP
...
%%
Term: ...
    | Sqrt '(' Exp ')' { $$ = sqrt($3); }
    | EXP '(' Exp ')' { $$ = exp($3); }
    | LOG '(' Exp ')' { $$ = log($3); }
```

❑ In Lex source

```
...
sqrt    return Sqrt
log     return LOG
exp     return EXP
...
```

•You must hard-code functions into parser and lexer

•Function names are *reserved words*

Reserved Words in Symbol Table

□ 심볼 테이블 구조

```
struct symtab {  
    char    *name;  
    double (*funcptr) ();  
    double  value;  
} symtab[NSYMS];
```

□ 심볼테이블에 예약어 저장

```
main() {  
    extern double sqrt(), exp(), log();  
    addfunc("sqrt", sqrt);  
    addfunc("sqrt", sqrt);  
    addfunc("sqrt", sqrt);  
  
    yyparse();  
}
```

```
void addfunc (char *name,  
              double(*func)( ))  
{  
    struct symtab *sp = symlook(name);  
    sp->funcptr = func;  
}
```

Reserved Words in Symbol Table

❑ In Yacc source

```
%token <symp> NAME FUNC
...
%%
Term: ...
    | FUNC '(' Exp' )' { $$ = sqrt($3); }
```

❑ In Lex source

```
[A-Za-z][A-Za-z0-9]* {
    struct symtab *sp = symlook(yytext);

    yylval.symp = sp;
    if(sp->funcptr) /* is it a function? */
        return FUNC
    else
        return NAME;
}
```