Flex と Bison を同時に使う

Flex と Bison を併用するとき、Flex から生成される関数 (yylex) は、トークン（終端記号） の情報を戻り値とし、それを Bison が利用します。 トークンが 1 文字の場合は、通常、戻り値は文字リテラルそのものです。トーク ンが 2 文字以上の場合は、Bison で %token により宣言されたマクロ を使用します。

ここでは 2文字以上の演算子の例として、「*+」という演算子を x *+ y が x * 256 + y を表し、「+」「-」と同じ優先順位を持つ、 左結合の演算子として導入します。仮に FOO 演算子と呼ぶことにします.

Flex のソースファイル（mylexer.l）には次のような #include 文を入れておきます。インクルードされるファイル （myparser.h） は Bison が生成するファイルで、%token に より宣言されたマクロの定義が書かれています。

```
%{
void yyerror(char*);

#define YY_SKIP_YYWRAP
int yywrap(void) { return 1; }
#include "myparser.h" /* myparser.h は bison が生成するファイル
%}
%option always-interactive
%%
[ \t]+                                  { /* ここでは何もしない */ }
[0-9]+(\.[0-9]+)?(E[+\-]?[0-9]+)? {
  /* NUMBER というトークンを返す。その値（〝属性〞）は
  yylval という大域変数に代入する。 */
                                          sscanf(yytext, "%lf", &y
                                        }
[+\-*/()\n]                             {
  /* + - * / ( )の場合は、マッチした文字をそのまま返す。*/
  /* マッチした文字は一般に yytext[0] 〜 yytext[yyleng - 1]。*/
                                          return yytext[0];
                                        }
"*+"                                    {
  /* 複数のルールにマッチする場合は，長い方が優先される */
                                          return FOO;
                                        }
.                                       { yyerror("不正な文字です。"
%%
/* なにもなし */
```

動作の中に return 文を入れておくと、 その式の値が Flex の生成する yylex 関数の戻り値になります。 yylex 関数は呼び出されるたびに、次のトークンを返します。

トークンは、 I 文字からなるトークンの場合は通常、 文字コードそのまま、 2 文字以上からなるトークンの場合は %token で宣言されたマクロです。

トークンの “種類”（NUMBER など）を yylex 関数の値として返し、値（“属性”）を yylval という大域変数に代入していることに注意します。 これが通常

の yylex 関数の書き方です。

一般に正規表現にマッチした文字は、 yytext という配列に保持されています。また yyleng という変数にマッチした文字の数が保持されています。 だからマッチした文字は一般に yytext[0] ～ yytext[yyleng - 1] ということになります。（通常の C 言語の文字列とは異なり、 最後（yytext[yyleng]） にナル文字 '\0' は入っていないので注意が必要です。）

例えば、入力が "(12*+34)*56\n" という文字列の場合、 yylex() の戻り値とそのときの yylval の値は次のようになります。

	1回目	2回目	3回目	4回目	5回目	6回目	7回目	8回目
yylex()	'('	NUMBER	FOO	NUMBER	')'	'*'	NUMBER	'\n'
yylval		12.0		34.0			56.0	

Bison のソースファイル （myparser.y）の方は、 単独で使う場合とあまり変わりませんが、 yylex 関数は Flex の方で用意するのでプロトタイプ宣言だけしておきます。

生成規則の中で、トークン（終端記号）として文字リテラル （'+', '-' など）と %token で宣言したマクロを用いることができます。

```
%{
#define YYSTYPE double  /* トークンの属性の型を宣言 */
#include <stdio.h>
#include <stdlib.h> /* exit 関数を使うため */

void yyerror(char* s) {
   printf("%s\n", s);
}

int yylex(void); /* yylex のプロトタイプ宣言 */
%}

%token NUMBER
%token FOO
  /* 演算子の優先順位 */
%left '+' '-' FOO
%left '*' '/'
%%
input :   /* 空 */
      | input line    {}
      ;
line  :  '\n'          { exit (0); } /* 空行だったら終了 */
      |  expr '\n'     { printf ("\t%g\n", $1); }
      ;
expr  :  NUMBER        { $$ = $1; }
      |  expr FOO expr { $$ = $1 * 256 + $3; }
      |  expr '+' expr { $$ = $1 + $3; }
      |  expr '-' expr { $$ = $1 - $3; }
      |  expr '*' expr { $$ = $1 * $3; }
      |  expr '/' expr { $$ = $1 / $3; }
      |  '(' expr ')'    $$ = $2; }
      ;
%%
```

```
/* yylex 関数は Flex が作成する。 */
int main(void) {
   printf("Ctrl-cで終了します。\n");
   yyparse();
   return 0;
}
```

C ソースファイルはそれぞれ次のコマンドで生成します。

```
bison -omyparser.c -d myparser.y
flex -omylexer.c -I mylexer.l
```

必ず -d オプションをつけて Bison を実行します。 このとき -o オプションで、C ファイル名 （この場合 myparser.c） を指定しておきます。 すると、拡張子を除いて同じ名前のヘッダーファイル （この場合 myparser.h） も生成されます。 （-o オプションをつけないと、 myparser.tab.c と myparser.tab.h という名前のファイルが生成されます。）

あとはこの 2 つの C ソースファイルをまとめてコンパイルします。

Microsoft Visual Studio の場合は、

```
cl /Fecalc mylexer.c myparser.c
```

/Fe は実行ファイルの名前を指定するオプションです。

これで calc.exe という名前の実行可能ファイルが生成されます。 次のコマンドで実行できます.

```
calc
```
