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、C 定義部に次のような `#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、bison を実行するときに与えるオプションで指定した C ソースファイルの名前の .c を .h に変えたものです。ここに `%token` により宣言されたマクロの定義が書かれています。

ファイル mylexer.l

```
%{
    /* C definitions */
void yyerror(char*);

#define YY_SKIP_YYWRAP
#define YYSTYPE double
int yywrap(void) { return 1; }
#include "myparser.h" /* cf. bison's -o option */
%}
    /* definitions */
%option yylineno
%option always-interactive
%%
    /* rules */
[ \t]+          { /* do nothing */ }
[0-9]+(\.[0-9]+)?(E[+\-]?[0-9]+)?   {
   sscanf(yytext, "%lf", &yylval); return NUMBER;
                }
[+\-*/()\n]     { return yytext[0]; }
"*+"            { return FOO; }
.               {
   yyerror("Illegal character."); return '\n';
}
%%
    /* additional C code */
```

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、 1 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。

この例では [0-9]+(\.[0-9]+)?(E[+\-]?[0-9]+)? という正規表現にマッチする文字列があれば、NUMBER というトークンを yylex の戻り値として返します。そのときの属性値は yylval という大域変数に代入されています。

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。）

この例では、+, −, *, /, (,), =, \n のいずれかの文字が現れたときは、その文字コードを返します。

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 C 宣言部でプロトタイプ宣言だけしておきます。（この例では Bison のソースファイルに yylex の定義を書いてはいけません。）

ファイル myparser.y

```
%{
 /* C declarations */
#define YYSTYPE double
#include <stdio.h>
#include <stdlib.h>

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

int yylex(void); /* prototype declaration */
%}
/* Bison declarations */
%token NUMBER
%token FOO
/* %left, %right or %nonassoc */
/* lower precedence */
%left '+' '-' FOO
%left '*' '/'
/* higher precedence */
%%
input :   /* empty */
```

```
23        | input line    {}
24        ;
25 line   : '\n'       { exit(0); }  /* an empty line */
26        | expr '\n' { printf("\t%g\n", $1); }
27        ;
28 expr :  NUMBER         { $$ = $1; }
29        | expr FOO expr { $$ = $1 * 256 + $3; }
30        | expr '+' expr { $$ = $1 + $3; }
31        | expr '-' expr { $$ = $1 - $3; }
32        | expr '*' expr { $$ = $1 * $3; }
33        | expr '/' expr { $$ = $1 / $3; }
34        | '(' expr ')'  { $$ = $2; }
35        ;
36 %%
37 /* additional C code */
38 /* no yylex() function here */
39 int main(void) {
40    printf("Exit with Ctrl-c.\n");
41    yyparse();
42    return 0;
43 }
```

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（FOO）を用いることができます。

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
```

PowerShell の場合は、次のコマンドになる。

```
.\calc
```