Flex について

Flex は正規表現を記述したソースファイルから、 C 言語の字句解析プログラム（スキャナー）を自動生成するプログラムです。 Flex のソースファイル（通常 `.l` か `.lex` という拡張子をつける）は、次のようなものです。

Flex は構文解析系（パーサー）から利用するための関数 `yylex` を生成します。 `yylex` 関数は、与えられたファイルを最初から順に読み、呼ばれるたびに次のトークンをリターンするのが本来の使用法ですが、この例では標準出力に結果を出力しています。

例 1

ファイル lexer0.l

```
%{
   /* C definitions */
#define YY_SKIP_YYWRAP
int yywrap(void) { return 1; }
%}
   /* definitions */
%option always-interactive

%%
   /* rules */
[hH]ello        { printf("Bonjour"); }
.|\n            { ECHO; }

%%
   /* user code */
int main (void) {
  return yylex();
}
```

これはおそらく一番簡単な例で、ファイル中の `hello` を `Bonjour` に書き換えます。

「`%{`」から「`%}`」の間（C definitions というコメントの部分）は C 定義部です。後の動作記述の中で用いる関数の定義や宣言をここに書きます。この例の 2 行:

```
#define YY_SKIP_YYWRAP
int yywrap(void) { return 1; }
```

は決まり文句なので、以下の例でもこのまま使用します。

定義部（definitions というコメントの部分）は最初の「`%%`」までで、 Flex へのオプションの指定や正規表現の定義などを書きます。この例では `always-`

interactive というオプションを指定しています。

最初の「%%」から 2 つめの「%%」まで（rules というコメントの部分）が Flex の核心部分で動作記述を書く部分です。左側に正規表現を、空白で区切って右側に正規表現が出現したときの動作（C のプログラム）を書きます。ここで、ECHO は正規表現にマッチした文字列をそのまま出力するマクロです。

2 つめの「%%」からファイルの最後まで（user code というコメントの部分）はユーザーコード部です。その他の関数の定義をここに書きます。この部分は、出力の C プログラムにそのままコピーされます。

この例では lexer 単独で動作させるために yylex 関数を呼び出すだけの main 関数を定義しています。ここで yylex は上の動作記述から Flex が生成する関数です。

なお、この例の動作記述では return していませんが、動作記述の中で return を書くと、その式が yylex 関数の戻り値になります。（これが通常の使い方です。）

例 2

ファイル lexer1.l

```
%{
   /* C definitions */

#define YY_SKIP_YYWRAP
int yywrap(void) { return 1; }
%}
   /* definitions */
%option always-interactive

%%
   /* rules */
[ \t]+                                  { putchar('_'); }
[0-9]+(\.[0-9]+)?(E[+\-]?[0-9]+)?    {
    printf("<b>"); ECHO; printf("</b>");
}
[A-Za-z]([A-Za-z0-9])*                  {
    printf("<i>"); ECHO; printf("</i>");
}
"."                                     { ECHO; exit(1); }

.|\n                                    { ECHO; }

%%
   /* user code */
int main (void) {
  return yylex();
}
```

これはもう少しだけ複雑な例です。標準入力から文字を読み込み、連続した空白文字をひとつのアンダースコア（_）に置き換え、数と識別子のまわりにそれぞれ、 〜, 〜 というタグを挿入します。

ファイル lexer2.l

```
%{
   /* C definitions */

#define YY_SKIP_YYWRAP
int yywrap(void) { return 1; }
%}
   /* definitions */
%option always-interactive
delim    [ \t]
ws       {delim}+
letter   [A-Za-z]
digit    [0-9]
ident    {letter}({letter}|{digit})*
number   {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
%%
   /* rules */
{ws}          { putchar('_'); }
{number}      { printf("<b>"); ECHO; printf("</b>"); }
{ident}       { printf("<i>"); ECHO; printf("</i>"); }
"."           { ECHO; exit(1); }
.|\n          { ECHO; }
%%
   /* user code */
int main (void) {
  return yylex();
}
```

次はさっきと同じ動作ですが、正規表現に名前をつけた例です。定義部（/* definitions */）に左側に正規表現の名前、空白で区切って右側に正規表現を書きます。この定義の行頭に空白を入れないようにしてください。（コメントと解釈されます。）

名前の定義の右側の正規表現で、それより前の行で定義している名前を使うことはできます。しかし再帰的な定義はできません。

なお bison （構文解析部生成系）で生成した構文解析部といっしょに動作させる例は、 ここにあります。

説明

Flex のソース中では次の文字は特別な意味を持ち、正規表現のために使います。（教科書や授業で説明した記法と微妙に違う場合があるので注意してください。）

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

式	意味	例
c	上記の特殊文字以外の文字は *c* という文字そのもの	a
c	\n, \t などは C 言語と同じ意味、 それ以外の文字は *c* そのもの 上記の特殊文字そのものをあらわすためには、この形を使う必要がある。	* \"
"*str*"	文字列 *str* そのもの **注:** "～" の中では " と \ は特別な意味を持つのでエスケープ （\" と \\）する必要がある。 **注:** \n, \t など C 言語で使われるエスケープシーケンスは C 言語と同じ意味になる。 **注:** その他の文字は "～" の中で特別な意味を持たない。	"**" "\"\"\"" "\\\""
.	改行以外の任意の文字	a.*b
[*str*]	文字列 *str* 中の任意の文字 **注:** [～] の中では], \, –, ^ は特別な意味を持つので エスケープ（\], \\, \–, \^）する必要がある。 **注:**], \, –, ^ 以外の文字は [～] の中で特別な意味を持たない。	[abc]
[c_1–c_2]	文字 c_1 から文字 c_2 の範囲の任意の文字 **注:** [～] の中では], \, –, ^ は特別な意味を持つので エスケープ（\], \\, \–, \^）する必要がある。 **注:**], \, –, ^ 以外の文字は [～] の中で特別な意味を持たない。	[0-9] [a-zA-Z] [a-zA-Z_\-$]
[^*str*]	文字列 *str* に含まれない任意の文字 **注:** [～] の中では], \, –, ^ は特別な意味を持つので エスケープ（\], \\, \–, \^）する必要がある。 **注:**], \, –, ^ 以外の文字は [～] の中で特別な意味を持たない。	[^abc] [^*+/\-]
*r**	正規表現 *r* が表す文字列の 0 回以上の繰り返し	a*
r+	正規表現 *r* が表す文字列の 1 回以上の繰り返し。 *rr** のこと	a+
r?	正規表現 *r* が表す文字列の 0 回か 1 回の出現。 ($r \mid \varepsilon$) に相当する	a?
r_1r_2	正規表現 r_1 が表す文字列の後に正規表現 r_2 が表す文字列が続く文字列	ab
$r_1 \mid r_2$	正規表現 r_1 が表す文字列、または 正規表現 r_2 が表す文字列	a|b
{*name*}	*name* という名前のついた正規表現の定義の展開	{ident}

この他、通常の括弧、(と)、がグループ化のために用いられます。例えば、(ab|cd)* は、ab または cd の 0 回以上の繰り返しを表わします。

生成

このようなファイル（lexer1.l とする）から C ソースファイルを生成するには

```
flex -I lexer1.l
```

というコマンドを実行します。

注: `-I`（小文字のエル）ではなくて `-I`（大文字のアイ）です。

これで `lex.yy.c` という名前の C ソースファイルが生成されます。また、

```
flex -ofoo.c -I lexer1.l
```

というように `-o` の後にファイル名を書くと、その名前（この場合 `foo.c`）の C ソースファイルが生成されます。

この例の場合は、この C ソースファイル（`foo.c`）をコンパイルします

Microsoft Visual Studio の場合は、`cl` でコンパイル

```
cl foo.c
```

（または

```
cl /Febar foo.c
```

）すると、実行可能ファイルが生成されます。

（`/Fe` は、実行可能ファイルの名前を指定するためのオプションです。後者のように `/Febar` というように使うと `bar.exe` という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。

```
foo
```

（コンパイラーのオプションで実行可能ファイルの名前を `bar` と指定している場合は、次のコマンドで実行できます。

```
bar
```

）

FAQ（よくある質問）

1. （例 1 について）作成したプログラムを実行したとき、どうやって終了すれば良いですか？

   Ctrl-c または Ctrl-z で終了できます。

2. （例 2 について）`exit(1)` って何ですか？

   `exit` はプログラムを終了させるための関数です。例 2 では「.」を入力するとプログラムを終了するようになっています。

3. （例 2 について）`\.` や `[+\-]` ってどういう意味ですか？

\. はピリオドそのものを表わします。「.」（ピリオド）は flex の規則中では特別な意味を持つので、ピリオドそのものを表わすには \ でエスケープしてやります。同様に「-」も[〜]の中では特別な意味を持つので、エスケープが必要で、[+\-] は「+ または -」の意味になります。

4. マッチする正規表現が 2 つ以上ある場合はどうなりますか？

   より長くマッチするほうが優先されます。同じ長さの場合には、先に書かれているほうが優先です。
