東京家政学院筑波女子大学紀要第９集　119～134ページ　　2005年

＜研究ノート＞

正規表現について
― その５　Perl の正規表現 ―

坂本　義行・江戸　浩幸

Regular Expression on Perl

Yoshiyuki SAKAMOTO and Hiroyuki EDO

概　要

前四編までに、Perl、sed、grep、Tcl、Python、Expect、Emacs など、よく使われるツールに実装されている正規表現エンジンの長所を探ると同時に、その短所を避けるための工夫について検討をしてきた。今回はその第５段階として、Perl は、正規表現がきわめて充実しており、初心者にも始めやすく、また Amiga、DOS、MacOS、NT、OS/2、Windows、VMS や、あらゆる種類の UNIX など、さまざまなプラットフォーム上で使える。従来型の構造とオブジェクト指向の概念を用いているが、データ処理は正規表現に大きく依存している。

本編では、正規表現に関する詳細や、それを実現する演算子など、Perl の正規表現に関するあらゆる事項について見ていく。Perl 言語の正規表現を無関係なものについてはあまり詳しくは触れない。Perl 特有の正規表現について、実験を含めて詳細な検討が加えられた技術についての知見を報告する。

キーワード：CSV テキスト、正規表現演算子、動的スコープ、コンテキスト

１　Perl 流プログラミング

表１は Perl の持つ、きわめて豊富な正規表現の特徴をまとめたものである。

Perl の正規表現の特性は、今日よく用いられるツールの中で、おそらく最大の威力を備えているのではないだろうか。

Perl には、何通りものやり方がある。

１．１　言語の一部としての正規表現

Perl 言語を構成する豊富な演算子や構文とも十分な整合性をもった正規表現演算子が提供されている。**表２**は、正規表現と直接関係のある言語仕様の一部を手短にまとめたものである。

１．２　Perl の最大の威力

演算子と機能が多様でオプションが豊富であるという点が Perl の最大の特徴といえる。

１．３　Perl の最大の弱点

このように豊かな表現力が凝縮されている

表１　Perl の正規表現言語一覧

<table>
<tr><td colspan="2">.（ドット）改行を除く任意のバイト（修飾子/sを伴う場合は全てのバイト）</td></tr>
<tr><td>｜　選択</td><td>(…)　通常のグループ化および格納
(?:…)　単なるグループ化のみ</td></tr>
<tr><td>欲張りな繰り返し制御文字
★　+　?　{n}　{min,}　{min,max}</td><td>(?=…)　肯定先読み
(?!…)　否定先読み</td></tr>
<tr><td>非欲張り型繰り返し制御文字
★?　+?　??　{n}?　{min,}?　{min,max}?</td><td>アンカー
\b　\B　単語/非単語用アンカー
^　$　文字列の先頭/末尾（または論理行の先頭および末尾）
\A　\Z　文字列の先頭/末尾
\G　前回のマッチの末尾</td></tr>
<tr><td>(?#…)　コメント
#…　修飾子/xを伴う場合、改行、または正規表現末尾までのコメント</td><td></td></tr>
<tr><td>埋め込み形修飾子
(?mods)　i、x、mおよびsの修飾子</td><td></td></tr>
<tr><td colspan="2">\1、\2など　格納型丸括弧の組によって前回マッチされたテキスト</td></tr>
<tr><td colspan="2">[…]　[^…]　通常及び否定の文字クラス（文字クラス内部では下記の要素も有効）</td></tr>
<tr><td>文字の略記法
\b　\t　\n　\r　\f　\a　\e　\num　\xnum　\cchar</td><td>クラスの略記法
\w　\W　\s　\S　\d　\D</td></tr>
<tr><td colspan="2">\l　\u　\L　\U　\Q　\E　テキスト変換指示</td></tr>
</table>

表２　Perl の正規表現関連要素の一覧

<table>
<tr><td>正規表現関連の演算子
†m/regex/mods
†s/regex/subst/mods
split(…)</td><td>修飾子　及ぼす影響
/x　/o　正規表現の解釈法
/s　/m　/i　エンジンが対象テキストをどう考慮するか
/g　/e　その他</td></tr>
<tr><td>関連変数
$_　デフォルトの検査対象
$★　複数行モード（現在は使わない）</td><td>マッチ後用の変数
$1、$2など　格納されたテキスト
$+　代入された$1、$2...の中の数字が最も大きいもの
$`　$&　$’　マッチ前、マッチ後、およびマッチのテキスト</td></tr>
<tr><td colspan="2">－関連関数－
pos　study　quotemeta　lc　lcfirst uc ucfirst</td></tr>
</table>

こと自体が、同時にPerlの最大の欠点にもなる。コードを少し変更するだけで、何の前触れもなしに、突然驚くような特殊な状況、条件、コンテキストが無数に生じる。

次のような例、

```
if(m/.../g) {
    ¦
```

これを理解できるようにすることが本論文の役目である。

１．４　鶏と卵－ Perl 流

重要な例について、問題に対する Perl 流のアプローチをしめしながら、陥りやすい落とし穴を指摘する。

１．５　導入例：CSV テキストの解析

dBASE や Excel などから出力された CSV 形式のファイル（Comma Separated Values: 値がコンマで区切られていた形式）からのデータを収めた $text がある。

"earth", 1,, "moon", 9.374

このデータを配列に入れたい。クォートで囲まれたフィールドからクォートを取り除く必要がある。次のような方法がある。

@fields=split(/,/, $text);

これは $text の中で「,」がマッチする箇所を見つけ、これらのマッチによって区切られる部分（これらのマッチしない部分）を @fields に格納する。

ここで著者が見つけたものが紹介されている。

■正規表現演算子のコンテキスト

「"([^"\\]*(\\.[^"\\]*)*)", ?|([^,]+), ?|,」

マッチ演算子が使われ方と使う箇所によって異なった振る舞いをするという点である。

■正規表現自身の詳細

３つの選択肢からなっている。

「"(?:[^"\\]*(\\.[^"\\]*)*)", ?」

ダブルクォート文字列をマッチするもので、印を付けた丸括弧は、単にテキストを $1に格納するためだけに用いられている。

「([^,]+), ?」

コンマ以外の文字列からなる、空ではない文字列、あるいはその後にコンマが１つ続くものをマッチする。」

「,」

コンマ１個をマッチする。

■実際に表現を適用する

Perl の従来型 NFA では、１番目の選択肢は対象がマッチ可能であれば必ずマッチし、２番目の選択肢は対象がマッチ可能できるようにクォート付フィールドを残すようなことは絶対にないからである。

修飾子 /g によって設定される「現在位置」はいつも、次のフィールドの先頭に置かれる。「同期取り」の概念 j は、修飾子 /g が使われる多くの状況において、とても重要になる。

最初の選択肢はクォート内のテキストを $1の中に格納する。

Push(@fields, defined($1) ? $1: $3);

下線の部分は、「$1が定義されていれば $1 を使い、そうでなければ $3を使う」

while ループと修飾子 /g、および同期機構を組み合わせることで、すべてのフィールドの処理が可能となる。

１．６　正規表現と Perl 流プログラミング

●正規表現の使い方が、他の部分の処理と緊密に結びつきすぎている場合、例えば、これらの３つの選択肢については、それぞれ独立させて考えたいところだが、１番目の選択肢が丸括弧の組を２つ持っているため、２番目の選択肢の丸括弧を参照するのに $3を使わなければならない。

● Perl（バージョン５）標準ライブラリモ

```
@fields = ( );  # @fields を初期化して空にする
while  ($text =~  m/ "([^ "\\]*(\\.[^ "\\]*)*) ",?|([^,]+),?|,/g) {
          push(@fields, defined($1)  ?  $1  :  $3);  # マッチしたフィールドを加える
          }
          push(@fields, undef) if $text =~ m/,$/;  # 最後の空フィールドを処理する
          # これで@fields を使ってデータにアクセスできる
```

```
Push(@fields, $+) while $text =~ m{
    "([^\ "\\]*(?:\\.[^\ "\\]*)*)",?  #標準クォート文字列（コンマを伴う場合を含む）
  | ([^,]+),?                         #あるいは次のコンマまで（コンマを伴う場合を含む）
  |  ,                                #あるいはコンマのみ
}gx;
```

ジュール Text::ParseWords に含まれる quotewords ルーチンを次のように使うと望みは果たされる。

Use Text::ParseWords;

　　┆

@fields=quotewords(',', 0, $text); Perl の標準ライブラリは広い範囲を網羅しているので、これを知っていれば、高水準のものから低水準まできわめて多様な機能が手軽に利用できる。

１．７　Perl の出現

■ Perl4 VS Perl5

最新版Perlでは、CSVの例の本体部分を次のように書くことができる。

例の中のコメントは実際に正規表現の一部である。

ここでは Perl5を中心に書き進める。

２．正規表現関連の Perl 作法

●コンテキスト　Perl の重要な概念の一つは、多くの関数や演算子が、自分が置かれているコンテキストに対応できるという点である。

●動的スコープ　動的スコープはグローバル変数を「保護」するもので、コピーを保存しておき、これを後で自動的に復元する。

●文字列処理　Perl の文字列「定数」は、きわめて機能的かつ動的な演算子である。なんと文字列の中からでも関数を呼び出せるのである。

２．１　式コンテキスト

リストコンテキストでは、リストの値が要求され、一方のスカラーコンテキストでは、単一の値が要求される。

２つの代入式を考えてみよう。

$s=*expression one;*

@a=*expression two;*

$s は単純なスカラー変数なので、単なるスカラー値１つを要求する。そのため最初の式は、スカラーコンテキストに置かれることになる。同様に、@a は配列変数であり、値のリストを要求するため、２つ目の式はリストコンテキストに置かれることになる。

■式コンテキストに対応させる

リストコンテキストでは、ファイルから（残っている）すべての行のリストを返す。スカラーコンテキストでは、単に次の１行を返す。

■式を変換する

リストコンテキストにスカラー値が与えられると、単一の値を要素として持つリストがその場で生成される。これにより、@a =42 は @a=(42)と同じになる。

２．２　動的スコープと正規表現マッチの効果

■グローバル変数とプライベート変数

Perl はグローバル変数とプライベート変数という２種類の変数を提供している。プライベート変数は、my(...)を使って宣言する。グローバル変数はプログラム内でどこからでも参照できる。一方、プライベート変数は、それを包含するブロックの終わりまでの間でのみ参照できる。

■動的スコープを持つ変数

変更したいグローバル変数のコピーを Perl に記憶させ、包含ブロックが終了した時に元の値を自動的に復元する。

●意図した変数のコピーを Perl に保存・復元させることで、一時的な目的にグローバル変数の変更を隔離することができる。

●例えば、file-include という命令を処理する場合、処理が終わるまで新しいファイルを「カレントのファイル名」に反映させておき、命令の処理が終了したら元の値に戻したい。

与えられたグローバル変数に対して、local は３つのことを行う。

１．変数の値の内部コピーを保存し、
２．新しい値を変数にコピーし（undef, または local に代入された値)、
３．実行処理が local を含むブロックを抜けた時点で、変数を書き換えて元の値を復元する。

その機能は**表３**の左側に示した。

２．３　わかりやすく例えると：透明シート

見る側（サブルーチンや割り込みハンドラなど、参照を行うもの）には新しい値が認識される。ブロックが最終的に終了する時点まで、以前の値は隠されるわけである。終了する時には、この透明シートは自動的に取り除かれ、実質上、local 宣言以降の変更が解除されるのである。

２．４　動的スコープのより高度な応用例

Perl 流の動的スコープの応用を示すより高度の例として、その中心となる関数は ProcessFile である。

■正規表現の副作用と動的スコープ

スコープは、正規表現とどの程度関係しているのだろうか？実は大きく関係している。成功マッチの副作用として、いくつかの変数が自動的に設定される。

次の例を見よう。

```
If (m/(...)/)
{
   &do_some_other_stuff( );
print "the matched text was $1.\n";
}
```

関数が行う $1の値の変更は、その関数が定義されるブロック内、または関数の部分ブロック内に納められている。このため、関数から戻った後に、このプログラムの中の print 文の認識する値が影響を受けることはない。

自動的な動的スコープは、明らかにそれとは分からない場合にも役に立っている。

```
If ($result = ~m/ERROR =(.*)/) {
   warn "Hey, tell $Config{perladmin}
      about $1!\n";
   }
```

表３　local の意味

通常のPerl	対応する意味

{
local($SomeVar);　# コピーを保存する
$SomeVar = ´My Value´;
⋮
⋮
⋮
⋮
}　　# $SomeVarの値を自動的に復元する
	{
	my $TempCopy = $SomeVar;
	$SomeVar = undef;
	$SomeVar = ´My Value´;
	⋮
	$SomeVar = $TempCopy;
	}

■動的スコープ対レキシカルスコープ

動的スコープは、効果的に用いれば多くの恩恵が得られる。

my(...)を宣言すると、レキシカルスコープを持ったプライベート変数が作成される。

２．５　マッチによって影響を受ける特殊変数

$&　正規表現によって成功マッチされたテキストのコピー。この変数は（下記の$` および$' とともに）使用しない方がよい

$`　マッチ開始位置より前（つまり左側）にある対象テキストのコピー。

$'　成功マッチしたテキストの後（つまり右側）にある対象テキストのコピー。

$1、$2、$3など

「(\w+)」と「(\w)+」の違い。

tubby という文字列に対してマッチを行うと、前者は $1の中に tubby を格納するが、後者は y のみを格納する。

「(x)?」と「(x?)」の違いも知っておくべきである。前者の場合、$1は x か未定義値になる。しかし「(x?)」の場合、「(x?)」を使うと$1の値は x か空文字列となる。

$+　マッチの際に明示的に設定された$1、$2などの中で最も大きな数字を持つもの。

■正規表現の中で $1を使う

正規表現に関する限り、$1の値はその時点でマッチは無関係であり、以前に別の場所で行われたマッチによって残された値を参照する。

２．６「ダブルクォート風処理」と変数展開

■演算子としての文字列

文字列を定数と考える。

```
$month = "January";
```

Perl はダブルクォート文字列内の変数を展開することができる。

```
$message="Report for $month:";
```

ダブルクォート文字列 "Report for $month:" は、次とまったく同じものである。

```
'Repor for ' . $month . ':'
```

ダブルクォート文字列内部から実際に関数を呼び出すことができるである。これはダブルクォートが演算子だからである。

qq/…/ という表記は、"…" と同じ機能を提供しているので、qq/Report for $month:/ はダブルクォート文字列となる。またユーザが好きなデリミタを選ぶこともできる。

シングルクォート文字列では、qq/.../ ではなく q/.../ を使う。

■文字列としての正規表現 / 正規表現としての文字列

Perl がプログラムを解析する順序を追ってみる。

例として、$dir には‘~/.bin’が格納されているものとする。

図 1 は、解析前のスクリプトから、正規表現オペランド、実際の正規表現、そして検索での使用に至るまでの段階を示したものである。スクリプトが読み込まれると、字句解析

```
Warn  qq{ "$ARGV " line $.: $ErrorMessage\n};
```

```
$header =~ m/^\Q$dir\E  # ディレクトリ
           \/           # 区切りのスラッシュ
            (.*)        # 残りのファイル名
            /xgm;
```

```
「^name=(?:' (...) '|"(...) ")$」
```

（スクリプトを調べ、どれが文で、どれが文字列で、どれが正規表現オペランドなのかといったことを判断する）が一度だけ行われる。

■段階Ａ －マッチオペランドの認識
■段階Ｂ －ダブルクォート風処理
■段階Ｃ － x/ 処理

段階Ｃは、/x が付加された正規表現オペラ

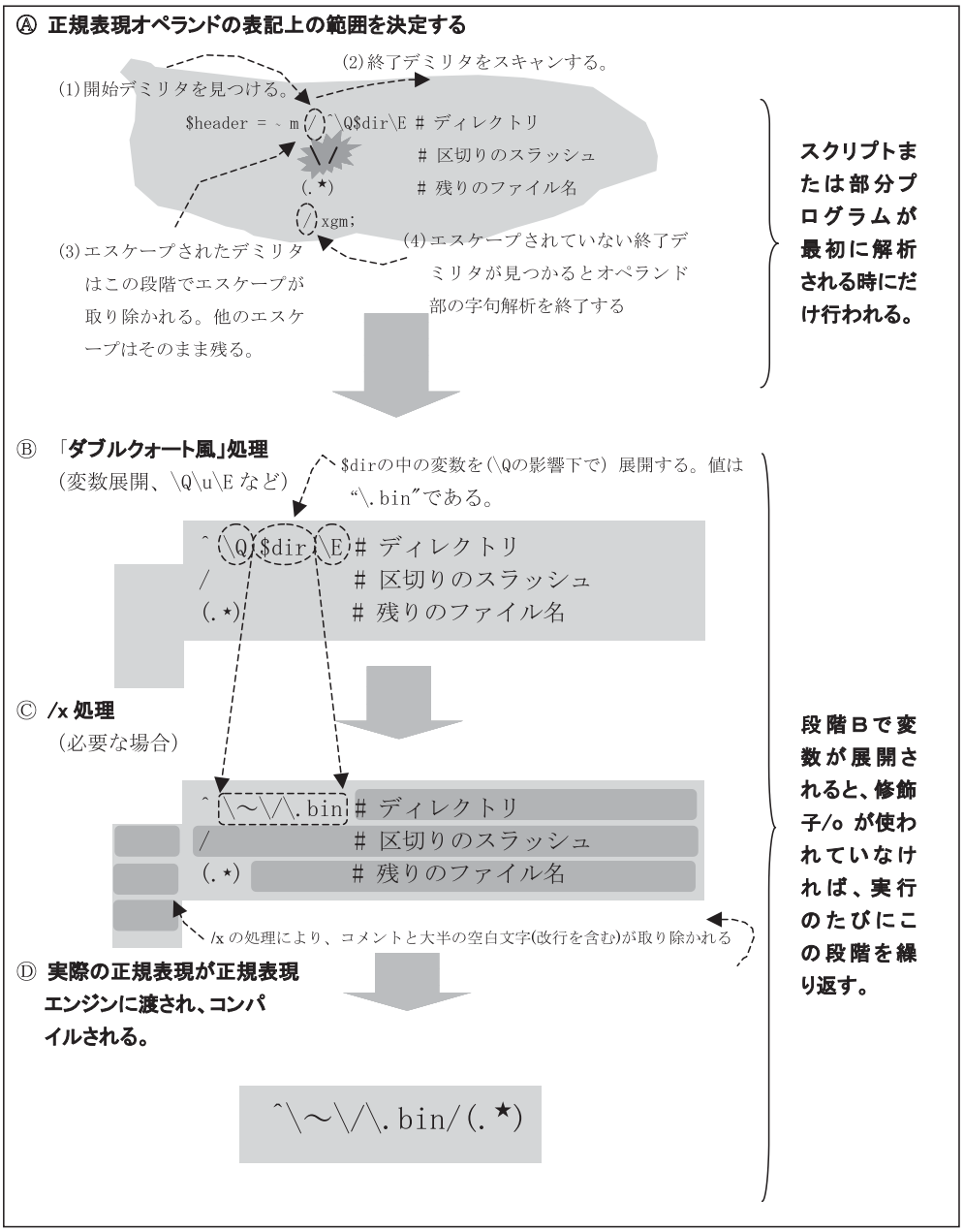

図１　プログラムテキストが正規表現エンジンに渡されるまでの解析処理

ンドのみが対象になる。

本当のコメントは下線で示した部分である。

■段階Ｄ－正規表現のコンパイル

３　Perl 正規表現の特性

PerlがNFAを使っていること以外に、便利な略記法やその他の基本機能を提供する多くの追加のメタ文字を持っていることに由来する。

３．１　繰り返し制御文字－欲張り型と怠け者型

Perl は通常の欲張り型繰り返し制御文字を提供しており、Perl5では２編「処理メカニズム」で解説した非欲張り型が追加されている。欲張り型繰り返し制御文字は「最大マッチ」とも呼ばれ、一方非欲張り型は「怠け者」、「非欲張り型」、「最小マッチ」などと呼ばれている。**表４**にこれをまとめた。

■非欲張り型の効率

■非欲張り型構文対否定文字クラス

否定文字クラスよりも入力が容易であるという理由で、非欲張り型の構文を使う人がいる。

３．２　グループ化

丸括弧には、グループ化と、マッチしたテキストを $1や $2の仲間に格納するという２つの機能がある。

Perl だけの特徴として２種類の丸括弧のサポートがある。グループ化と格納に使われる従来の(…)、ならびに Perl5で新しく加えられたグループ化専用(?:...)である。

■格納型対非格納型丸括弧

グループ化専用で、非格納型の構文には、次の利点がある。

●効率的なマッチが行える

●正規表現エンジンの内部最適化によって、

表４　Perl の繰り返し制御文字（欲張り型と怠け者型）

マッチ数	従来の貪欲型（最大マッチ）	怠け者型（非欲張り型）（最小マッチ）
任意の回数（ゼロ回、１回、またはそれ以上）	★	★?
１回以上	+	+?
省略可能（ゼロ回または１回）	?	??
範囲指定（*min*回以上;*max*回以下）	{*min*,*max*}	{*min*,*max*}?
下限（最低*min*回）	{*min*,}	{*min*,}?
回数指定	{*num*}	{*num*}?

```
S<
        (\d{1,3})       # コンマの前に：1 桁から 3 桁の数字で
        (?=             # 次に以下の（マッチには含まれない）パターンが含まれるもの
         (?:\d\d\d)+    #  3 桁の数字の組が続いて
         (?!\d)         #  後に数字が続かない
         )              #  （つまり、数字の終わり）
    >,$1,>gx;
```

より効率的な表現に変換できる可能性がある。

●文字列内に正規表現を構築するのが容易になる。

■先読み

●否定先読みは、何ともマッチできない場合に成功となる。

●先読みは（肯定でも否定でも）どんな複雑な正規表現でもテストすることができる。

先読み例をいくつか挙げる。

「Bill(? ＝ _The_Cat?_Clinton)」

Bill にマッチするが、後に ‘_The_Cat’ か ‘_Clinton’ が続く場合に限られる。

「\d+(?!\.)」

後ろにピリオドが続かない数値とマッチする。

「数字にコンマを追加する」から、特に興味深い例を引用する。

■正規表現内のコメント

「(?#...)」の構文はコメントとして解釈され、無視される。

「(?#...)」は、改行あるいは正規表現の終わりまでコメントとして扱うことができる。

CSV の例にあった文字列を、次のようにもっとわかりやすく書くことができる。

■他の（?…）構文

３．３　文字列アンカー

アンカーは、水も漏らさぬ表現を作る上で不可欠なものだ。Perl は数種類の機能を提供している。

■論理行対生のテキスト

Perl ではどちらも可能である。実際には４種類のモードがる。これを**表５**にまとめた。

対象文字列内に改行が埋め込まれていなければ、すべてのモードの動作は同じである。

■キャレット、ドル記号、およびドットのデフォルト動作

Perl のデフォルトでは、キャレットは文字列の先頭でのみマッチする。ドル記号は文字列末尾、もしくは文字列末尾にある改行の直前にマッチする。

入力が行ごとに読まれ、末尾の改行はデー

```
$text =~ m/ ″([^ ″\\]*(\\.[^  ″\\]*)*)  ″,? | ([^,]+),? | ,/g
```

```
$text =~ m/ ″([^ ″\\]*(\\.[^ ″\\]*)*) ″,?|([^,]+),?|,/g
を、次のようにもっと分かりやすく書くことができる。
  $text =~ m/                              # フィールドは３種類のうちいずれかである
                                           #    1）ダブルクォート文字列
     ″([^ ″\\]*(\\.[^ ″\\]*)*) ″           #    -標準ダブルクォート文字列（$1 にぶち
                                                込む）
     ,?                                    #    -後続のコマンドを取り込む
       |                                   #    または
                                           #   2)通常フィールド
   ([^,]+)                                 #    -次のコマンドを（$3 に）格納する
   ,?                                      #    -(そしてコンマがあればそれを含める）
             |                             #    または
                                           #   3）空フィルド
   ,                                       #    ただコマンドをマッチする
   /gx;
```

タの一部として保持される。

■ /m および不適切な名前の「複数行モード」

複数行モードのマッチを有効にするには m/ を使う。

修飾子 m/ は、「^」および「$」が改行を処理する方法にだけ影響する。

m/ は正規表現のマッチのみに影響する。

■単独行モード

/s を使うと、ドットはどんな文字ともマッチする。

■純粋な複数行モード

/m と /s の修飾子を併用することで、「純粋」な複数行モードを実現することができる。

ドットが改行とマッチしない特殊ケースを排除することで、より純粋で単純な動作を実現できるのでこう呼ぶことにする。

■明示的な文字列の先頭および末尾

Perl5では、文字列の先頭とマッチする「\A」および末尾とマッチする「\Z」が提供されている。

■改行は常に特殊扱いになる

改行が必ず特殊扱いになる状況が一つある。モードの如何にかかわらず、「$」および「\Z」は、必ずテキスト末尾の改行の前でマッチすることができる。

■ $* の警告を解除する

{local($^W)=0; eval '$*= 1'}

こうすると、$* に対する操作を行っている間に警告モードが解除され、元々警告がオンであれば、その状態は保存される。

■ /m 対(?m)、/s 対 /m

/m か（正規表現のどこであっても）「(?m)」があれば、全体のマッチに対して複数行モードが設定される。

３．４　複数マッチアンカー

Perl5には「\G」アンカーが追加されている。

■ \G を使った一例

５桁からなる米国郵便番号（ZIP コード）がひとつながりになったデータがある。

0382453144941161522134418295035442727520102174432 35

@zip=m/\d\d\d\d\d/g; を使って、各要素が１つの ZIP コードを含むリストを作成しよう。

■マッチの同期をとる

下記のような例を正規表現の先頭に挿入すると、いずれも期待通りの結果を得ることができる。

「(?:[^4]\d\d\d\d[^4]\d\d\d)*…」

「(?:(?!44)d\d\d\d\d)*…」

「(?:\d\d\d\d\d)*?...」

@zip=m/(?:\d\d\d\d\d)*?(44\d\d\d)/g;

■マッチ不成立後の同期も維持する

■ \g の位置付け

上に示した２つの例の方が、読みやすさや

表５　改行に関係したマッチモードの概要

モード	^と$の行アンカーによる対照テキストの認識	ドット
デフォルトモード	改行文字の有無とは関係なく、単独文字列としてみなす	改行とマッチしない
単独行モード	改行文字の有無とは関係なく、単独文字列としてみなす	すべての文字とマッチ
複数行モード	改行によって区切られた複数論理行としてみなす	（デフォルトと同じ）
純粋な複数行	改行によって区切られた複数論理行としてみなす	すべての文字とマッチ

```
@zips = grep {defined} m/(44\d\d\d|\d\d\d\d\d/g;
@zips = grep {m/^44/} m/\d\d\d\d\d/g;
```

保守性において優れている。

■ /g マッチのプライミング

/G を使わないとしても、Perlがどのように「前回マッチの終端」を覚えているかは重要である。

データ内の全数字のリストを @nums に返す場合を考えてみる。

@nums=$data=～ m/\d+/g;

特定の文字列 <xx> が行内にある場合、その後に数字のみを手に入れたいとする。

$data=˜m/<xx>/g; #/g が開始する位置を合わせる。pos($data)は現在 <xx> の直後を示している

@nums=$data=～m/?d+/g;

筆者はこの技術を「/g のプライミング」と呼ぶ。

３．５　単語アンカー

Perl には「\b」および「\B」という単語境界アンカーがある。

単語境界とは、片側にある文字が「\w」とマッチし、その反対側の文字が「\W」とマッチする位置である。

■「\b」の注意点

次のような表現が単純かつ効果的である。

「(?:\W|ˆ) \Q$item\E(?!\w)」

■個別の単語先頭アンカーおよび単語末尾アンカーがないことへの対応

s/\b(?!\w), *\b(?=\w)//

例えば上の文字列は、文字列内の最初と最後の単語の間にあるものをすべて取り除く。

３．６　便利な略記法とその他の記法

Perl の便利な略記法を数多く見てきた。**表７**はその完全なリストである。

表６　アンカーおよびドットのモードのまとめ

モード	指定方法	^	$	\A、\Z	ドットによる改行マッチ
デフォルト	/s も/m なし、$ *偽	文字列	文字列	文字列	しない
単独行	/s（$*は関係なし）	文字列	文字列	文字列	する
複数行	/m（$*は関係なし）	行	行	文字列	しない（デフォルトと変らず）
純粋な複数行	/m と/s($*は関係なし)	行	行	文字列	する
旧式の複数行	/s も/m もなし、$*は真	行	行	文字列	しない（デフォルトと変らず）
文字列―埋め込まれた改行を意識しない 行―埋め込まれた改行を意識する					

表７　正規表現の略記法と特殊文字エンコーディング

バイト記法	機種依存の制御文字
num　8 進表記の文字	\a　アラーム(ベル)
xnum　16 進表記の文字	\f　改頁
ochar　制御文字	\e　エスケープ
一般クラスの略記法	\n　改行
\d　[0-9]の数字	\r　復帰
\s　空白文字。通常は[\f\n\r\t]	\t　タブ
\w　単語構成文字。通常は[a-zA-Z0-9…]	\b　バックスペース
\D、\S、\W、…\d、\s、\wの反対	（文字クラス内のみ）

■ Perl と POSIX ロケール

Perl における POSIX ロケールのサポートはきわめて限定的である。

■関連する標準モジュール

関連する Perl の標準モジュールには、POSIX モジュールや Jarkko Hietaniemi の I18N::Collate モジュールがある。

■バイト記法

Perl では、数値的な値を使ってバイトを正規表現内に簡単に挿入する方法が提供されている。２桁または３桁の８進値で「\33」や「\177」あるいは、１桁または２桁の16進値で、「\xA」や「\xFF」のように表される。

■バイト対文字、改行対ラインフィード

\n やその仲間で定義される値は、Perl によって決定されるのでなく、システムに依存する。

３．７　文字クラス

Perl の文字クラス内文法は、バックスラッシュエスケープを完全にサポートいているという点で、多くの正規表現文法の中でも独特である。

スター、プラス記号、丸括弧、ドット、選択、アンカーなどは、どれも文字クラスの中では意味をもたない。

■文字クラスと非 ASCII データ

８進および16進エスケープは、文字クラスの内部、特に領域指定の場合にきわめて便利である。

次に簡単な実装例を紹介する。

３．８　\Q とその仲間を用いた修飾：本当の正体

\L、\E、\u といった表８にまとめたような

```
Foreach $item (@Items) {
  $key = lc $item;                          # $item をコピーし、ASCII を小文字に変換する。
  $key =~ s/[\x d9-\xdc\xf9-\xfcl/u/g;      # アクセント記号の付いたすべての u を、単なる u 変換する他のアクセント記号の付いた文字にも同様の処理…

  $pair{$item} = $key;                      # $item->$key の関連付けを覚えておく
    }

    # 要素をそれぞれのキーによってソートする。
    @SortedItems = sort { $pair{$a} cmp $pair{$b} } @Items;
```

表８　文字列および正規表現オペランドの文字ケース変更構文

文字列内構文	意味	埋め込み関数
\L　\U	\E*までテキストの文字を小文字あるいは大文字にする	lc(...)、uc(...)
\l、\u	次の文字を小文字あるいは大文字にする†	lcfirst(...) ucfirst(...)
\Q	\E まで、すべての非アルファベットの前にエスケープを加える	quotemeta(...)
特別な組み合わせ		
\u\L	最初の文字を大文字にし、\E またはテキスト末尾まで残りの文字を小文字にする	
\l\U	最初の文字を小文字にし、\E またはテキスト末尾まで残りの文字を大文字にする	

* これら\E では、いずれも\E が指定されなければ、「文字列または正規表現の末尾まで」となる

† Perl5 の場合、\L...\E および\U...\E の中では、\L または\U の直後でない限り無視される。

項目が見つかるだろう。

３．８　第２クラスメタ文字

●オペランドに対するごく初期の検査の間のみ有効となる

●正規表現にコンパイルされる文字のみに影響を与える

４　マッチ演算子

基本的な文字列マッチは、Perl 正規表現の使用目的の中核をなす。

「分割･統治方法」を用いながら次の点を見ていくことにしよう。

●マッチ演算子自体の指定（正規表現オペランドに対して）

●マッチする対象文字列の指定（対象オペランドに対して）

●マッチによる副作用

●マッチによって返される値

●マッチに対する外部的影響

４．１　マッチオペランドデリミタ

マッチオペランドデリミタとしてはスラッシュを使うのが一般的だが、好きな記号を選ぶこともできる。

バックスラッシュで煩わしい思いをするよりも、違うデリミタを使えば可読性がさらに高まる。m!^/(?:[^/]+/)+Perl$! と m,^/(?:[^/]+/)+Perl$, は、ともに同じ意味の正規表現を表わす例である。他には m|…|、m#…#、m%…% などのデリミタがよく使われる。

● m(...)、m{...}、m[...]、m<...> のケースは特別で、対応する開き・閉じデリミタをもつ。修飾子 /x と組み合わせて使うと次のような使い方ができる。

```
m{
    ここは          # ここは
    正規表現        # コメント
  }x;
```

●シングルクォートをデリミタとして使用した場合、図１の段階Ｂが省略され、変数展開が無効になり、**表８**に示した記法が使えなくなる。

●疑問符をデリミタとして用いた場合、最初の成功マッチに対してのみ成功値を返す。

これは、ループ内で、特定のマッチを一度だけ成功させたいと時に役立つ。例えば、電子メールのヘッダを処理する場合、次のように使うことができる。

```
$subject=$1 if m?^Subject: (.*)?;
```

■デフォルトの正規表現

もし m// のように正規表現が指定されなかった場合には、Perl はそのコードが含まれる動的スコープの中で一番最後に成功した正規表現を再使用する。

４．２　マッチ修飾子

マッチ演算子は次のような効果がある。

●正規表現オペランドがどのように解釈されるか

●正規表現エンジンが対象テキストをどうみなすか

●エンジンが正規表現をどう適用するか

■ m/.../g で無にマッチする正規表現を使う

表９に簡単な例をいくつか示した。

４．３　マッチ対象オペランドの指定

“*expr*=~ m/.../” 全体が式なので、式を置ける場所であればどこにでも置くことができる。例を示す。

```
$text=~m/.../;      # 副作用を期待して、とりあえずマッチしてみる
```

また、＝~ の代わりに !~ を使って、戻り値を論理的に否定することもある。

４．４　マッチ演算子によるその他の副作用

「目に見えない状態」を保持する２種類の状況がある。一つは、もしマッチが m?...? を使って指定されている場合、もう一つは、適

表９　何ともマッチしない正規表現を m/…/g に使った例

正規表現	「d*」 マッチ回数	「\d*」 マッチ回数	「x｜\d*」 マッチ回数	「\d*｜x」* マッチ回数
Perl4	123▲　2	123▲x▲　3	▲a123▲w▲x▲y▲z456▲　8	▲a123▲w▲x▲y▲z456▲ 8
Perl5†	123　1	123　x▲　3	▲a123　w▲x　y▲z456　5	▲a123　w▲x▲y▲z456　6

（各マッチは下線部で、またゼロ幅のマッチの場合は▲で示した）

* ちなみに「\d*」は決して失敗しないので、「｜x」は使われることはない。そのため無意味になる点に注意

† バージョン 5.001 以降

用する正規表現が、動的スコープが終わるか、別の正規表現がマッチするまで、デフォルトの正規表現になることである。

４．５　マッチ演算子の戻り値

■スカラーコンテキスト（修飾子 /g なし）

修飾子 /g のないスカラーコンテキストは、マッチが見つかると、真のブール値が返る。

■リストコンテキスト（修飾子 /g なし）

修飾子 /g のないリストコンテキストは、正規表現内の各格納用丸括弧の組に対応する要素を返す。

■リストコンテキスト（修飾子 /g あり）

/g を用いたリストコンテキストでは１回のマッチの結果であるのに対し、この場合は文字列内のすべてのマッチの結果を返す。

■スカラーコンテキスト（修飾子 /g あり）

スカラーコンテキストの m/…/g は、他の３つとはかなり異なる特殊な構文である。

これを while ループの条件式として使うと非常に便利である。次の例を見てほしい。

４．６　マッチ演算子に対する外部からの影響

- ●コンテキスト―マッチ方法、およびその戻り値や副作用に大きく影響する
- ● $* ― Perl4から受け継いだもので、キャレットおよびドル記号アンカーに影響を与える
- ●デフォルト正規表現
- ● m?...? ―演算子 m?...? が「すでにマッチしたかどうか」を示す目に見えない状態に影響する

５　置換演算子

Perl の置換演算子 s/*regex/replacement*/ は、テキストをマッチするという概念をマッチおよび置換にまで押し広げたものである。

５．１　置換オペランド

一般的な形式である s/.../.../ の場合、置換オペランドは正規表現オペランドの直後に置かれ、m/.../ のように２つではなく、合計３つのデリミタを使う。

- ●一回実行型のマッチ演算子特殊デリミタ ?...? は、置換演算子と併用する場合、特殊な意味を失う
- ●バッククォートをデリミタとして用いても、Perl4の時のように特殊な意味は持たない。

```
While ($ConfigData =~ m/^(\w+)=(.*)/mg) {
    my($key, $value) = ($1, $2);
}
```

５．２　修飾子 /e

修飾子 /e を使うことができるのは置換演算子だけである。eval(…)を用いたのと同じように評価され（ロード時に構文解析される)、その結果がマッチされたテキストと置換される。

■変化する評価対象

特に修飾子 /e を使う場合、例えば、s/…/`echo $$'/e といった簡単なものでさえ、$s を解釈するのはその Perl 自身なのか、それともシェルなのかといった疑問が生じる。

■ /eieio

/e が複数回指定された場合、置換オペランドがその回数だけ評価される（出現回数が意味を持つ唯一の修飾子である)。

５．３　コンテキストと戻り値

置換演算子にはこうした複雑な問題は一切ない。これは同じ種類のデータを返す。

返される値は実行された置換の回数であるり、置換が行われなければ空文字列になる。

５．４　/g を無とマッチする正規表現と併用する

表 9 に示したものはすべて、s/…/…/g のマッチ動作にも当てはまる、

６　split 演算子

多面的特徴を持つ split 演算子はリストコンテキスト m/…/g の裏返しとしてよく用いられる。

６．１　基本知識

split は関数に似た演算子で、オペランドを最大３個とる。

split（マッチオペランド、対象文字列、分割数指定オペランド）

省略されたオペランドに対してはデフォルト値が適用される。

■基本的なマッチペランド

マッチオペランドが与えられなければ、デフォルトのマッチオペランドが使われるが、これは、この先解説する複雑な特殊ケースの１つである。

■対象文字列オペランド

■分割数指定オペランド

分割数指定オペランドの主な役割は、split が分割する個数に上限を指定することである。

６．２　split の高度な利用法

split は関数ではなく演算子なので、通常の関数呼び出し処理に縛られることなく、「魔法のよう」なやり方でそのオペランドを解釈することができる。

Split は便利だが、使いこなすのはそれほど簡単ではない。注意すべき点をいくつか挙げる。

- split のマッチオペランドは、通常のマッチオペランド m/…/ とは異なる。
- 正規表現が無をマッチできるとどうなるか？
- 正規表現に格納型丸括弧があると、split は異なった振る舞いをする。

■ split は空要素を返すことができる

split の基本前提は、マッチ間にあるテキストを返すことである。

■後続の空要素は（通常）返さない

分割数指定オペランドが指定されていないと、Perl はリストを返す前に最後の空要素を、リストから取り除いてしまう。

■分割数指定オペランドの第２の機能

分割要素の数を限定できるだけでなく、ゼロ以外の分割数指定オペランドを与えることで、後続の空要素が取り除かれるのを防ぐことができる。

６．３　split マッチオペランドの高度の利用

マッチオペランドには４種類の形式がある。

■ split のマッチ演算子型マッチオペランド

マッチオペランドと本物のマッチ演算子の間には重要な違いがある。

- ●マッチ演算子の対象文字列オペランドを指定するのは避けるべきである
- ● m/x*/ や s/x*/.../ を使う場合など、繰り返しはマッチ演算子ではなく split によって行われるので、状況はずっと単純である
- ● split に空の正規表現を使うと、「その時のデフォルトの正規表現を使え」という意味ではなく、対象文字列を文字単位に分割せよという意味になる
- ●正規表現を split と併用しても、その後のデフォルトマッチや置換演算子に影響を与えることはない
- ●修飾子 /g は split と併用しても無意味である
- ●特殊正規表現デリミタ ?...? は split と併用すると、特殊ではなくなる

■特殊マッチオペランド：シングルスペース１個の文字列

スペースただ１個からなる文字列（正規表現ではない）によるマッチオペランドは、特殊ケースとして扱われる。

■マッチオペランドとして一般のスカラー値を使う

マッチオペランドとして使われるといずれも独立して評価され、文字列として受け取られ、さらに正規表現として解釈される。

■デフォルトのマッチオペランド

デフォルトのマッチオペランド(// や " とは異なる)は、' ' を使った場合とまったく同じになる。

６．４　スカラーコンテキストでの split

Perl4はスカラーコンテキストでの split をサポートしている。この機能を使わないほうがよいとされており、将来的にはなくなる可能性が高い。

６．５　split のマッチオペランドの中で格納型丸括弧を使う

マッチオペランドの正規表現の中で格納型丸括弧を用いると、split の様相全体が一変する。この場合、返されるリストの中は、丸括弧が格納した要素も挿入されるようになる。

参考文献

１）Jeffrey E.F.Friedl 著、歌代和正監訳、1999、オライリー・ジャパン発行

２）江戸浩幸・坂本義行、「電子メール自動集計システムーI」、東京家政学院筑波女子大学紀要、第４集、2000.

３）江戸浩幸・坂本義行、「電子メール自動集計システム II」、東京家政学院筑波女子大学紀要、第５集、2001.

４）坂本義行・江戸浩幸、「正規表現についてーその１　どう読むかー」、東京家政学院筑波女子大学紀要、第５集、2001.

５）坂本義行・江戸浩幸、「正規表現についてーその２　処理メカニズムー」、東京家政学院筑波女子大学紀要、第６集、2002.

６）坂本義行・江戸浩幸、「正規表現についてーその３　正規表現を工夫する」、東京家政学院筑波女子大学紀要、第７集、2003.

７）坂本義行・江戸浩幸、「正規表現についてーその４　ツールエンジンの効率化」、東京家政学院筑波女子大学紀要、第８集、2004.