東京家政学院筑波女子大学紀要第８集　187～194ページ　　2004年

＜研究ノート＞

正規表現について
― その４ ツールエンジンの効率化 ―

坂本　義行・江戸　浩幸

Tool Efficiency on Regular Expression

Yoshiyuki SAKAMOTO and Hiroyuki EDO

概　要

前三編において、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４段階として、さらに「詳説正規表現」を読み進め、ツールエンジンが従来型 NFA か POSIX NFA を調べるテスト、「ループ展開」の技術について、どのような正規表現を構築すれば、検索において効率化が図られるのか。C 言語で用いられるコメント文中で最高効率の検索を行うため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：正規表現、ツールエンジン、ループ展開、C コメント、特殊要素

１　エンジンの種類のテスト

ツールの正規表現エンジンをテストする作業には、いくつかの手順がある。

１．１　NFA と DFA の基本テスト

理論的には、エンジンの基本種類を調べるためには、永久に終わらないマッチについて調べればよい。

awx についてのテスト

echo　=XX==...===　|　awx　"/X(.＋)＊X/{print}"

もし =XX==... の行が即座に出力されれば、その awx は（大半のものと同じく）DFA を使っていることになる。もし手持ちの awx が mawx や Mortice Kern Systems 社の awx であれば、NFA が使われている。

１．２　従来型 NFA か POSIX NFA かを調べるテスト

従来型 NFA はマッチを見つけた瞬間に止まるため、永遠に続くような正規表現も、マッチがあれば通常すぐに終了する。

従来型か POSIX かのテストは、これまでに述べた最適化を考えると難しくなる場合が多い。POSIX で即座に成功マッチを返すの

は、これが必要性認識の最適化を採用しているからである。つまり正規表現を「マッチするかしないか」のように二者択一的に使うと、マッチの範囲は重要ではなくなり、わざわざ最長マッチ（この場合は特定マッチ）を見つける必要もなくなる。

２　ループ展開

「ループ展開」と呼ぶ技術がある。これはある一般的な表現の速度を上げるのに力を発揮する。対象となるループは「(*this*|*that*|...)＊」のようなパターンの表現で使われるスターによるものである。前出の永久マッチ「"(\\.|[^"\\]＋)＊"」がこのパターンに当てはまる。マッチ不成立を報告するのに半永久的な時間がかかることを考えれば、これは速度向上を試すのに適した例といえる。

２つの異なる方法で、この技術に到達することができる。

１．さまざまなマッチテスト例の中で、実際に「(\\.|[^"\\]＋)＊」のどの部分が成功したかを検討し、使われた部分パターンの流れを記録しておく。そしてそこから浮かび上がったパターンに基づいて効率的な表現を再構築する。

２．もう一つの方法は、マッチしたい構文を、さらに高い次元から見る方法である。

２．１　方法１：これまでの経験に基づいて正規表現を構築する

「"(\\.|[^"\\]＋)＊"」を分析する上で、いくつかのマッチ対象文字列を調べて、全体マッチの中で厳密にどの部分パターンが使われているかを知る。

次の場合は、「"[^"\\]＋\\.[^"\\]＋\\.[^"\\]＋"」となる。

"he said \"hi there\" and left"

この例と表１では、パターンが分かりやすいように表現をハイライトしてある。

最初の４つの例を考えてみよう。クォートの間の表現が「[^"\\]＋」で始まり、次にいくつかの「\\.|[^"\\]＋」という並びが続く。これ全体を正規表現として表現し直すと、「[^"\\]＋(\\.[^"\\]＋)*」になる。

◆汎用「ループ」展開パターンの構築

ダブルクォート文字列をマッチする場合、「通常要素＋(特殊要素　通常要素＋)＊」という汎用パターンに当てはまる。これに開きクォートと閉じクォートを加えると、

「"[^"\\]＋(\\.[^"\\]＋)＊"」ができる。この表現の中にある２つの「[^"\\]」が文字列の先頭、および特殊な文字の後ろで通常の文字を要求している点にある。

では「"[^"\\]＊(\\.[^"\\]＊)＊"」をよく見てみよう。必要条件である開きクォートとその後に続くかもしれない通常文字をマッチする。

表１　ループ展開表現例

対象文字列	実際に使われる表現
“hi there”	” [^” \ \]+”
“just one \” here”	” [^” \ \]+\ \.[^” \ \]+”
“some \” quoted \” things”	” [^” \ \]+\ \.[^” \ \]+\ \.[^” \ \]+”
“with \” a\” and \” b\” .”	” [^” \ \]+\ \.[^” \ \]+\ \.[^” \ \]+\ \.[^” \ \]+\ \.[^” \ \]+”
“ \” ok \” \n”	” \ \.[^” \ \]+\ \. \ \.”
“empty \” \” quote”	” [^” \ \]+\ \. \ \.[^” \ \]+”

２．２　本格的な「ループ展開」パターン

この種類の表現に用いる汎用パターン：「開始　通常要素＊(特殊要素　通常要素＊)＊ 終了」

◆永久マッチの回避

「"[^"\\]＊(\\.[^"\\]＊)＊"」が永久マッチにならないのは、非常に重要な以下の３つの点が守られているからである。

特殊要素と通常要素の開始位置を決して交差させない

特殊要素と通常要素が絶対に同じ位置でマッチしないようにすれば、対象文字列にマッチできる特殊要素の「並び」が必ず一つだけ存在するはずである。この一つの並びをテストする方が、何億もの並びテストするよりはるかに速く、永久マッチを回避することができる。通常要素を「[^"\\]」とし、特殊要素「\\.」とするここでの例は、この条件を満たしている。これらの要素は決して同じ文字でマッチを開始することはない。つまり、後者は先頭にバックスラッシュを要求するのに対し、前者は明示的にバックスラッシュを許していない。

特殊要素は無とマッチしてはいけない

特殊要素はマッチするときには必ず最低１個の文字とマッチしなければならない。

一つの特殊要素でマッチできるテストは、複数の特殊要素でマッチできてはならない。

Pascal の {...} 型コメントとスペースからなる文字列をマッチする場合、コメント部分とマッチする正規表現は「\{[\}]＊\}」であり、全体表現は「(\{[^}]＊\}|_＋)＊」となる。

特殊要素	通常要素
「_＋」	「\ {[^}]＊\}」

これを今まで作っておいた「通常要素＊(特殊要素　通常要素＊)＊」というパターンに組み込むと、「(\{[^}]＊\})＊(_＋(\{[^}]＊\})＊)＊」になる。

注意すべき一般事項

「(...＊)＊」のような繰り返し制御文字が複数レベルで存在しているのは、重要な危険信号であるが、こうした表現であってもまったく問題ないものも多い。例えば次のようなものである。

・「(Re:_＊)」＊」　任意数の 'Re:' をもつシーケンスとマッチする。

・「(_＊\$[0-9]＋)＊」（スペースによって区切られた）ドル金額とマッチする。

・「(.＊\n)＋」　一つ以上の行とマッチする。

２．３　方法２：高次元の視点で見る方法

「\\.」は、時々現れるエスケープされた要素の処理に用いられるだけである。いずれかを許す選択を用いれば便利な正規表現ができるが、エスケープされた数少ない（一つもない場合の方が多い）要素のために、マッチ全体の効率性を考えると無駄である。

「[^"\\]＋」が終わる度に、閉じクォートが別のエスケープのマッチかという前と同じ状況がやってくるのである。

最終的に、「"[^"\\]＋(\\.[^"\\]＋)＊"」となる。

２．４　方法３：クォート付きのインターネットホスト名

ループ展開の技術を実現する方法を２つ述べたが、３つ目ともいえる似た方法を紹介する。

これは、prez.whitehouse.gov や www.yahoo.com といったドメイン名をマッチする正規表現を構築するときである。これを「[a-z]＋(\.[a-z]＋)＊」と書くと、文字列が通常の「[^"\\]の並びで、これが「\\.」で区切られ、全体が '"..."' で囲まれていると考えれば、ループ展開パターン「"[^"\\]＋(\\.[^"\\]＋)＊"」を作ることができる。

２．５　考察

このダブルクォートの文字列の例についてまとめると、「"[^"\\]＊(\\.[^"\\]＊)＊"」の表現には利点が多く、しかもほとんど欠点がないことがわかる。

欠点

・効率のために読みやすさを多少犠牲にしている（「"([^"\\]|\\.)＊")」）と比較
・効率のために維持性を多少犠牲にしている

利点

・速度の点で、新しい正規表現は、マッチが不可能な場合でも破綻することはない

３　C コメントの展開

C 言語では、コメントは/* で始まり、*/ で終了し、複数行にまたがることもある。

３．１　複雑な正規表現

単に「/\＊[^＊]＊\＊/」とするだけではうまくいかない。これでは内部に有効な '＊' を含む/＊＊コメント部分＊＊/などをマッチできない。対象となるコメントは/＊...＊/ ではなく、/x...x/ として考える。「/\＊[^＊]＊\＊/」は「/x[^x]＊x/」のように、ずっと読みやすくなる。

３．２　厳密でない考え方

1．開きデリミタをマッチする
2．テキスト本体をマッチする：実際には「終了デリミタでない任意のもの」をマッチする
3．閉じデリミタをマッチする

この場合、終了デリミタが単独の文字でない場合、成功しない。これを解決する一つの方法は、x を終了デリミタの始まりとして解釈する方法である。「終了デリミタでない任意のもの」は、

・x ではない任意のもの、すなわち「[^x]」
・x で、その後ろにスラッシュが続かないもの、すなわち「x[^/]」

のいずれかになる。すなわち、「/x([^x]|x[^/])＊x/」ができる。

もう一つの方法は、x が先行する場合に限り、スラッシュを終了デリミタとみなす方法である。「終了デリミタでない任意のもの」は、

・スラッシュでない任意のもの、すなわち「[^/]」
・前に x を伴わないスラッシュ１個、すなわち「[^x]/」

これより、「/x([^/]|[^x]/)＊x/」ができる。しかし、いずれの方法もうまくいかない。前の方法は、'/xx_foo_xx/' の例で、後の例は、'/x/_foo_/x/' の例でうまくいかないことが分かる。

◆正しい表現に修正する

解決策は、「本当に必要なものを書く」という視点に立ち返ることである。

すなわち、「/x([^x]|x＋[^/x])＊x＋/」が得られる。

実際のコメント（x の代わりに＊が使われる）では次のようにしなければならない。

「/\＊([^＊]|\＊＋[^/＊])＊\＊＋/」

となる。正規表現を読むことは簡単なことではない。肝心なのは、十分に気を配りながら、複雑な表現を頭の中で慎重に区切って考えることだ。

３．３　C コメント表現のループ展開

効率の点から、正規表現の展開について考えてみよう。表２は、ループ展開のパターンの中に挿入できる表現を示したものである。通常要素の並びが、常に終了デリミタの最初の文字で終わり、次に続く文字が末尾を構成しない場合に限り、特殊要素として取り扱いたい。

これらの要素を汎用展開パターンに挿入すると次のようになる。

表２　C 言語コメント用のループ展開要素

「開き 通常*(特殊 通常*)* 閉じ」		
要素	**対象**	**正規表現**
開き	コメント冒頭	/x
通常*	コメントテキストから1個以上の'x'を含むところまで	[^x]*x+
特殊	終了スラッシュ以外（かつ'x'ではない）のもの	[^/x]
閉じ	後続のスラッシュ	/

「/x[^x]＊x ＋([^/x][^x]＊x＋)＊/」

◆現実を考える

「/x[^x]＊x ＋([^/x][^x]＊x＋)＊/」は、すぐにそのまま使えるというわけではない。当然ながら、コメントは /x...x/ ではなく /＊... ＊/ である。これは各 x を \＊と置き換えれば簡単に直る（文字クラスであれば各 x は＊に置き換わる）。

「/\＊[^＊]＊\＊＋([^/＊][^＊]＊\＊＋)＊/」

これを用いる際に関係してくる問題は、コメントがしばしば複数行にまたがることである。

４　スムーズに流れる正規表現

文字列定数内のコメントに似た要素がマッチされないようにするにはどうしたらよいかという問題が残されている。

４．１　マッチするように導くには

Tcl の例：

```
set COMMENT {/\＊[^＊]＊\＊＋([^/＊][^＊]＊\＊＋)＊/}
# コメントをマッチする正規表現
set DOUBLE {"(\\.|[^"\\])＊"}
＃ダブルクォート文字列をマッチする正規表現
regsub -all "$DOUBLE|$COMMENT" $text {} text
```

これでは、不十分で、次のように変更すべきである。

```
set COMMENT {/\＊[^＊]＊\＊＋([^/＊][^＊]＊\＊＋)＊/}
＃コメントをマッチする正規表現
set DOUBLE {"(\\.|[^"\\])＊"}
＃ダブルクォート文字列をマッチする正規表現
regsub -all "($DOUBLE)|$COMMENT" $text {\1} text
```

この変更点は

・マッチが文字列選択肢であった場合、\1（Perl の $1に対応する Tcl の置換文字列）を埋める丸括弧を加えた点。マッチがコメント選択肢の場合には \1は空のままになる。

・置換文字列を上に出てきた \1にした点。これにより、ダブルクォート文字列がマッチされると、置換は同一のダブルクォート文字列になる。

最後に、'\t' のようにシングルクォートで囲まれた C の文字定数を処理する必要がある。これには、単に丸括弧の中にもう一つの選択肢を加えれば済む。

```
set COMMENT
{/\＊[^＊]＊\＊＋([^/＊][^＊]＊\＊＋)＊/}
＃コメントをマッチする正規表現
set COMMENT2 "*//\[^\n]*"
＃ C++ の // コメントをマッチする正規表現
set DOUBLE {\\.|[^"\\])*"}
＃ダブルクォート文字列をマッチする正規表現
set SINGLE {(\\.|[^'\\])＊'}
＃シングルクォート文字列をマッチする正規
```

表現

regsub -all
"($DOUBLE)|$SINGLE)|$COMMENT|
$COMMENT2" $text {\1} text

この基本的な構造はかなり単純である。これにより効率化が図られた。

４．２　よく制御された正規表現は速い

正規表現エンジンが進む流れを制御して、もっと速くマッチさせることができる。

例えば、選択肢がマッチするためには、先頭の文字はスラッシュ、シングルクォートまたはダブルクォートでなければならない。時間と労力のかかる方法でエンジンに模索させるのでなく、「[^'"/]」を選択肢として加えて、エンジンに直接指示を出す。実際には、これらが連続している場合は一気に捕まえられるので、代わりに「[^'"/]＋」使うことにする。実験ではこれにより、１桁も短縮できた。

もっとすばやくマッチさせる方法がある。

・一般的な選択肢は「$OTHER＋」なので、これを丸括弧の中で最初に持ってくる

・人間がより上位の視点に立って局所的な最適化を行うことで、正規表現エンジンをマッチに導いているからである。

４．３　まとめ

２つの文字列部分パターンを次のように置き換える。

set DOUBLE　{"[^"\\]＊(\\.[^"\\]＊)＊"}

set SINGLE　{'[^'\\]＊(\\.[^'\\])＊)＊'}

さらに、25%短縮できた。図１はこれまでの短縮時間の成果をすべて示したものである。

５　エンジンの最高効率化

汎用の最適化よりも重要なこととして、NFA エンジンから最高効率を引き出すには、自分の目標を達成するために必要なものと不要なものを見極めると同時に、それを実現するために、NFA エンジンがどう動けばよいのかを把握することが鍵となる。

５．１　最適化の紆余曲折

「\b**char**\b|\b**const**\b|\b**ouble**\b...
\b**singned**\b|\b**unsigned**\b|\b**while**\b」

上の文字列を使ってＣ言語の予約語を含む行を見つける場合を考える。

具体的な例として、Perl のソースプログラムからこれらの単語を含む行を数える短い

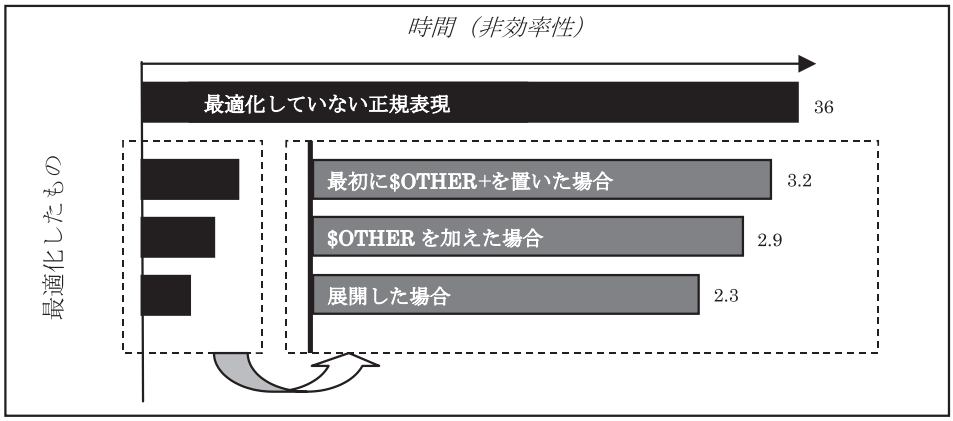

図１　十分に制御された最適化による短縮時間効果

Perl スクリプトを作った。その一つを次に示す。

```
$count = 0;
while（<>）
{
  $count++ if m<   \bchar\b
                 | \bconst\b
                 | \bdoube\b
    .
    .
    .
                 | \bsign"ed\b
                 | \bunsigned\b
                 | \bwhile\b  >x;
}
```

ここではPerlの/x機能が使われている。これは正規表現におけるクラスの部分で自由形式のスペースを許す。長く複数行にわたる「m <...> x」の内部では、空白文字を除くすべてが（この例では多くの選択肢を含む）一つの正規表現となる。次のような別のスクリプトも試してみた。

```
$count = 0;
while（<>） {
    $count++ next if m/ \bchar\b/;
    $count++ next if m \bconst\b/;
    $count++ next if m \bdoube\b/;
      .
      .
      .
    $count++ next if m \bsigned\b/;
    $count++ next if m \bunsigned\b/;
    $count++ next if m \bwhile\b/;
}
```

どちらからも同じ結果が得られたが、２番目の方が６倍以上も速い。それはバックトラックにかかる処理負担の多くが取り除かれたからである。正規表現が「.＊」など、無差別の文字で始まるマッチを許してしまう構造で始まる場合は、最適化が無効になるが、この例では適用可能である。

◆他の最適化を考える

これをさらに推し進め、手作業で各選択肢に共通した先頭要素を選択の冒頭に持ってくることもできる。この作業は、
「this|that|the_other」を
「th(is|at|e_other)」に変更するのと似ている。こうした変更を行うことで、比較的速度の遅かった選択を、頭文字識別を使わなくても「th」がマッチした後でのみ実行することができる。

総じて、プログラムの効率化は、アイデアとロジックを用いることで完全ではないにせよ大方実現することができる。筆者の責任はこうした高速化方式の中で最速の方法を教えることである。Perl をはじめ、筆者がこれらのテストに用いたツールにしても、次のバー上では内部最適化速度の順位が変わるかもしれない。

あとがき

今回も前回に引き続き、各エンジンによって、実際の検索効率が大きく異なることに着目して、文字列により、エンジンの種類でどのような検索が行われるかについて詳細な検討を加えた。とくに、「ループ展開」のパターンで効率化を図るにはどのような正規表現が有効であるかを検討した。さらにエンジンの種類によってCコメント表現のループ展開において、スムーズに流れる正規表現の記述の仕方、実際のエンジンでの表現の違いによる速度差を実験により確かめられている。次回はPerlについて、その機能の実態を詳細に検討し、実際の応用プログラムに適用した場合の効率的な正規表現について検証を加えていく予定である。

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ー」、東京家政学院筑波女子大学紀要、第７集、2003.