東京家政学院筑波女子大学紀要第６集　153～167ページ　　2002年

＜研究ノート＞

正規表現について

－その２　処理メカニズム－

坂本　義行・江戸　浩幸

Processing Mechanism on Regular Expressions Ⅱ

Yoshiyuki SAKAMOTO and Hiroyuki EDO

概　要

前編において、正規表現の概要と基本的な記号およびメタ記号について学び、それには厳密な表記規則（文法）があり、また、プログラミング言語固有の解釈あるいは処理がなされることを学んだ。その応用例として、awkエンジンを用いて、電子メール自動集計処理システムの開発をすすめ、簡潔で理解し易いプログラムの作成と使い勝手の良いシステムの構築を目指した。今回はその第２段階として、「詳説正規表現」を読みすすめ、正規表現エンジンに使われている非決定性有限オートマトン（NFA）と決定性有限オートマトン（DFA）の違いと、そこに記述されているエンジンによる動作の違いを代表的な３種、動作型DFA、従来型NFA、POSIX NFAに分類し、電子メールで用いられる例をもとに、厳密な解釈と動作状態を把握することに努めた。その経過を報告する。

キーワード：正規表現、非決定性有限オートマトン、決定性有限オートマトン

１．正規表現エンジンの種類

正規表現は、基本的に二つの異なる種類に分類される。一つは『DFA』と呼ばれ、もう一つは『NFA』と呼ばれる。

NFAの方がよく使われている。

NFAエンジンは、Tcl, Perl, Python, GNU Emacs, ed, sed, vi, grepの大部分の版と、egrep, awkの一部の版で使われている。DFAエンジンは、egrep, awkの大部分の版とlex, flexの一部の版で使われている。

すなわち、大きく分けて次の３種類に分類できる。

＊DFA（POSIX対応型および非対応型）
＊従来型NFA
＊POSIX NFA

表１　各プログラム言語と正規表現エンジン

プログラム名	（原）著作者	版	正規表現エンジン
awk	Aho,Weinberger,Kernighan	総称	DFA
新awk	Brian Kernighan	総称	DFA
GNU awk	Arnold Robbins		大半はDFAだが一部はNFA
MKS awk	Mortice Kern Systems		POSIX NFA
mawk	Mike Brennan	すべて	POSIX NFA
egrep	Alfred Aho	総称	DFA
MKS egrep	Mortice Kern Systems		POSIX NFA
GNU Emacs	Richard Stallman	すべて	従来型NFA(POSIX NFAもあり)
Expect	Don Libes	すべて	従来型NFA
expr	Dick Haight	総称	従来型NFA
grep	Ken Thompson	総称	従来型NFA
GNU grep	Mike Haertel	ﾊﾞｰｼﾞｮﾝ2.0	大半はDFAだが一部はNFA
GNU find	GNU		従来型NFA
lex	Mike Lesk	総称	DFA
flex	Vern Pazon	すべて	DFA
lex	Mortice Kern Systems		POSIX NFA
more	Eric Schienbrood	総称	従来型NFA
less	Mark Nudelman		不定(通常は従来型NFA)
Perl	Larry Wall	すべて	従来型NFA
Python	Guido van Rossum	すべて	従来型NFA
sed	Lee McMahon	総称	従来型NFA
Tcl	John Ousterhout	すべて	従来型NFA
vi	Bill Joy	総称	従来型NFA

２．マッチの基本原則

2.1　例題について

本章では主に、機能が完備した典型的な正規表現エンジンに焦点を当てる。そのためツール（プログラム言語）によっては示したものすべてがサポートされていないものもある。

大半の例では引き続きPerlの記法を用いる。本編では、マッチを実行した際の結果について詳しく説明する。

１．最初のマッチが優先される

２．繰り返し制御文字は可能な限りのマッチを行う

2.2　第一原則：最初のマッチが優先される

文字列中で最も早く始まったマッチが、それより後に始まるどのマッチより常に優先される。マッチはまず検索対象文字列の先頭（最初の文字の直前）でテストされる。

例えば、次の文字列に対して「cat」をマッチさせた場合はどうなるだろうか？

The dragging belly indicates your cat is too fat

このマッチは行の後ろにあるcatという単語ではなく、indicatesの中で起こる。

「fat|cat|belly|our」の場合、「fat」が選択肢の中で最初に置かれているにもかかわらず、‘ belly ’とマッチする。すなわち下線の引かれた部分、

The dragging belly indicates your cat is too fat

となる。

正規表現は検索文字列中のある位置で完全にテストされてから、文字列に沿って次の位置に移動し、そこで再びテストされる。

2.3　エンジン部分

リテラル文字

「usa」のように正規表現がリテラルテキストだけの場合、まず「u」、次に「s」、そして「a」という形でマッチが行われる。

文字クラス、ドット、および同種の記号

文字クラスがどのように長いものでも、１文字のみとマッチする。文字クラスはマッチ可能な文字の集合を表す。含まれる文字は明示的に指定され、否定クラスでは明示的に除外される。

「\w」、「\W」、「\d」、「\D」、「\s」、「\S」といった略記法などにも当てはまる。

アンカー

これらは検索対象文字列中の位置とマッチする。

2.4　第二原則：一部のメタ文字は可能な限りのマッチを行う

スター、プラス記号、選択といったさらに強力なメタ文字を使わなければ、有効で効率的な処理は行えない。

繰り返し制御文字（?、*、+および{*min*, *max*}）、すなわち、

不特定回数のマッチが許されているアイテムは、常に可能な限り何回でもマッチを行う。

簡単な例は正規表現「\<\w+s\>」を用いて、regexesのように「s」で終わる単語をマッチさせる場合である。「\w+」だけでも問題なく単語全体とマッチするが、そうすると「s」の部分にマッチできない。この場合は「\w+」はregexesの部分でマッチし、その後の「s\>」のマッチを可能にすれば、正規表現全体がマッチに成功する。繰り返し制御文字は必ず最低必要数以上で、できるだけ多くの回数を繰り返す

「[0-9]+」がMarch_1998の数字全体と一致するのかが説明できる。

この記号は最大回数のマッチを行なおうとするため、文字列末尾によって強制的に止められるまで引き続き998とマッチを行うのである。

Subjectの例

電子メールのヘッダーから一行を抜き出し、それがSubject行かどうかをチェックしたい場合について考える。単に「^Subject:_」を使えば目的は果たせる。だが、「^Subject:_(.*)」を使えば、ツールがサポートする丸括弧の事後参照用メモリ（例えばPerlの$1）によって、後の処理でSubject行の中からテキスト部分を取り出すことができる。

「.*」の場合、［マッチなし］という最悪のケースでも、スタートにとっては成功と見なされるので、絶対に失敗することがない。

ここで丸括弧を使うのは、単に「.*」でマッチしたテキストを格納するためである。

変数$lineが次のような文字列を保持している場合、

```
Subject: Re: happy birthday
```

下記のPerlのコードは、

```
if ($line =~m/^Subject: (.*)/ )
  { print "The subject is: $1\n";
  }
```

の処理結果として次ぎのような結果をだす。

```
The subject is: Re: happy birthday
```

返信について

「^(Re:_)?」が（.*）の前に加えられている。「^(Re:_)?」の方が先に 'Re:_' とマッチし、その後に「.*」が残りの部分をマッチする。実は「^(Re:_)*」も使うことができる。これは返信のやり取りの間にたまったRe:をすべて削除してくれる。

```
if ($line =~m/^Subject: (Re: )?(.*)/ )
  {print "the subject is: $2\n";
  }
```

を実行すると、

'The subject is: happy birthday'

という結果が得られる。

最後の比較例として、同じ表現で丸括弧を一つ動かした例

「^Subject: ((Re:_)? .*)」

では、'Re:_' の付いた題名をそっくりそのまま検索し、しかも返信かどうかを簡単に見分ける方法も欲しい場合には役に立つ。

余分な正規表現

「^Subject:_(.*).*」のように。「.*」をもう一つ加えると、マッチおよび結果はどのように変わるだろうか？答えは『何も変わらない』である。

次の例では、

「^.*([0-9][0-9])」を'about_24_characters_long'に適用した結果として、$1が '24' を取り込む。

先のマッチが優先

正規表現を「^.*([0-9]+)」に変えてみると、「[0-9]+」は「[0-9]*」とマッチ１文字分しか変わらないことに注意しよう。「[0-9]*」は「.*」と同類である。

３．正規表現制御型とテキスト制御型

NFAエンジンを『正規表現制御型（regex directed)』と呼び、DFAエンジンを『テキスト制御型（text-directed)』と呼ぶ。

3.1　NFAエンジン：正規表現制御型

エンジンが正規表現「to(nite|knight|night)」を '…tonight…' というテキストに対してマッチさせる場合、「t」からはじまって、正規表現は一度に１要素ずつ調べられる。

「to(nite|knight|night)の例では、最初の要素は「t」であるが、これがマッチしたら「o」が次の文字に対してチェックされ、もしマッチしたら制御が次の要素に移る。『次の要素』は、『「nite」、「knight」または「night」』を意味する「(nite|knight|night)」だ。これら３つの要素に対し、エンジンは一つずつ順番に見ていく。

制御は正規表現内を要素から要素へと移るため、筆者はこれを『正規表現制御型』と呼んでいる。

NFAエンジンにおける制御上の利点

NFAエンジンは正規表現で制御されるため、正規表現を書く人間には、その動きをかなり自分の思い通りに工夫する機会が与えられている。

3.2　DFAエンジン：テキスト制御型

後続の文字がスキャンされる度に、マッチ途中の情報が更新される。

２つの候補に対してマッチ動作が行われている（もう１つの選択肢、knightは除外されている)。

しかし、次にgが出てくることによって、最後の選択肢だけが有効なものとして残る。そしてhとtがスキャンされると、エンジンは完全なマッチが修了したと判断して成功を返す。

これを『テキスト制御型』のマッチと呼ぶこととする。

NFAとの違い

２種類のエンジンを比べると、DFAエンジンの方が全体的に速度が速いという結論になるだろう。NFAのマッチでは、部分パターンが何度も適用される。一方、DFAエンジンではすべてのマッチ候補を平行して管理してい

文字列中	正規表現中
…t▲onight…の後	マッチ途中状態：t▲o(nite | knight | night)」

文字列中	正規表現中
…toni▲ght…の後	マッチ途中状態：to(ni▲te | knight | night)」

るため、たった１度のチェックで終わる。

3.3　正式な名称

正規表現エンジンに使われている２つの基本技術の名称は、非決定性有限オートマトン（Nondeterministic Finite Automaton:NFA）と決定性有限オートマトン（Deterministic Finite Automaton:DFA）である。

正規表現制御型であるNFAにとって、正規表現の書き方を変えるだけで、様々なケースを試してみることができる。tonightの例の場合、「to(ni(ght|te)|knight)」、「tonite|toknight|tonight」または「to(k?night|nite)」という違った正規表現を書けば、もっと無駄が省けたかもしれない。

DFAはまったく正反対である。上記のような表記の違いは、最終的に同じマッチを表現している限りまったく問題にならない。

この章のまとめ

・DFAのマッチは非常に速い
・DFAのマッチは一貫している
・DFAのマッチは正規表現の違いによる差が処理の差として現われない。

正規表現制御型のNFAについては、正規表現を工夫することにより、処理に差が現われる。これを理解するためには、NFAエンジンの本質、すなわちバックトラックを学ぶ必要がある。

４．バックトラック

4.1　分岐点の指標

NFAエンジンの本質は以下のようなところにある。各部分パターンや要素を順番に調べ、等しく有効な２つの候補の間で判断を下す必要がある場合は片方を選択し、同時に後で必要なときに戻れるように、もう片方を記憶しておく。

マッチの位置指標を示す例

文字列 'hot_tonic_tonight!' に対して先の正規表現「to(nite|knight|night)」をフルに使った例を見てみよう。最初の要素「t」が文字列先頭で試される。hとのマッチが失敗する。

やがてこのテストが…▲tonic…の位置で始まる。toが一度マッチすると、これら３つの選択肢がどれも有効な候補となる.正規表現はこのうち１つを選んでテストを行うが、最初の候補が失敗した場合を考えて、残りの候補も記憶しておく。エンジンは始めに「nite」を選択したとしよう。この表現は“「n」+「i」+「t」…”というように分解できるので、…toni▲c…のところまで行って失敗する。エンジンは別の候補、例えば「knight」を選ぶが、これは即座に失敗する。

エンジンが…▲tonight!の位置で始まるテストのところに来る。正規表現の末尾までマッチが成功したので、全体マッチが成功する。

4.2　バックトラックに関する２つの重要事項

バックトラックの基本的な概念だけを説明する。バックトラックせざるを得ない場合、エンジンは記憶してある選択候補のうちいずれを使うべきなのだろうか？

疑問符やスターなどに支配される要素に関して、『テストを行う』か『テストをスキップする』かを判断する状況では、エンジンは必ずテストを行う。表現全体を成功させる必要性から止むを得ない場合に限り、（その要素をスキップするために）後で戻ってくる。

局所的な失敗によってバックトラックが行われると、最も新しく保存された候補が選択される。つまりLIFO（last in first out:後入れ先出し）である。

4.3 記憶されたカレントステート

ステートは、必要に応じてテストを再開する位置を示す。正規表現内の位置、および未実行の候補が始まる文字列内の位置を反映している。

バックトラックを行わないマッチ

abcに対して「abc」をマッチする例について見てみる。「a」がマッチすると、マッチのカレントステート（現在の位置）は次ぎのように表される。

'a▲bc' の位置で	「a▲b?c」をマッチする

エンジンは次ぎの内容をそれまで空だったステートの保存リストに加える。

'a▲bc' の位置で	「ab?▲c」をマッチする

テキスト中ではbの直前（つまり現在位置）からマッチが再開できることを示している。

'ab▲c' の位置で	「ab?▲c」をマッチする

バックトラック後のマッチ

もしマッチ対象のテキストが 'ac' なら、「b」がテストされるところまではすべて同じである。エンジンはバックトラックを行う。つまり一番最後に保存されたステートを最新のカレントステートとして選ぶのである。

'a▲c' の位置で	「ab?▲c」をマッチする

マッチ不成立

表現は同じだが、今度はabxに対して行われる例を見よう。

'a▲bX' の位置で	「ab?▲c」をマッチする

正規表現を再実行する。これを異種の擬似バックトラックと考えてもよい。マッチは次ぎの位置から再開される。

'a▲bX' の位置で	「▲ab? c」をマッチ

今度は新しい位置でもう1度全体のマッチが行われるが、前回と同様、すべての経路が失敗する。その後の２回のテスト（ab▲xとabx▲）が同じく失敗した後で、全体マッチが完全に失敗したことが報告される。

4.4 バックトラックと網羅性

目指す目標をすばやく達成する正規表現を書くためには、バックトラックが自分の正規表現ではどのように行われているかを理解することが鍵となる。「?」の網羅的なマッチ動作がどんなものかについては学んだ。ではスター（およびプラス記号）の動作を見よう。

スター、プラス記号、およびそのバックトラック

「[0-9]+」を 'a_1234_num' に対してマッチさせた場合、文字列の各位置に対してマッチが再開できることを示す４通りのステートが保存されている。

a_1▲234_num
a_12▲34_num
a_123▲4_num
a_1234▲_num

先に挙げた４つの文字列位置のリストには、'a_▲1234_num' が含まれていない。プラス記号を用いた最初のマッチは任意ではなく、不可欠である。

もっとも本格的な例

４ページの『余分な正規表現』の「^.*([0-9][0-9])」の例を再び調べてみる。

例として 'CA_95472,_usa' を使う。「.*」が文字列末尾までマッチを成功させると、スターに支配されたドットがマッチする（必要に応じて）省略可能な対象から、12のステートができる。

『バックトラック－テスト』というサイクルは、エンジンが２をマッチ解除するまで続けられ、その位置で最初の「0-9」がマッチする。だが２番目の「0-9」はマッチしないので、バックトラックを続けなければならない。この場合、最初の「0-9」がその前のテストでマッチしたことは関係ない。カレントステート

はバックトラックによって最初の「0-9」の前に再設定されるからだ。結果として、同じバックトラックで文字列の位置も７の前に変更されるため、最初の「0-9」が再びマッチする。今度は2番目の「0-9」も（２と）マッチする。このため ‘CA_95472,_USA’ というマッチが得られ、$1には72が取り込まれる。

スター（またはいずれかの繰り返し制御文字）に支配されるものは、正規表現内でその後に続くものとは無関係に。真っ先に、しかも可能な限りマッチを行う。という点を理解しておくことが重要である。

５．網羅性について

網羅性から生ずる多くの問題（および利点）は、NFA式とDFA式のいずれにも存在する。DFAは『網羅的』の一語にに尽きる。

NFAエンジンでは、正規表現を書いた人間がマッチの実行方法を直接管理できる。これによって多くの利得が得られるが､同時に効率面でいくつかの問題点もある。

両者のエンジンについて話を進めるが、理解が容易なNFA正規表現制御型の観点で説明する。

5.1　網羅性による問題

先の例で見たように、「.*」によるマッチは必ず行末まで進む。注１）ダブルクォートで囲まれたテキストにマッチする正規表現を考えてみよう。次ぎの例のどこでマッチするかを考えて欲しい。

The name "McDonald’s" is said "makudonarudo" in Japanese

次ぎの部分とマッチすることがわかる。

The name "McDonald’s" is said "mkudonarudo" in Japanese

明らかにこれは意図していたようなダブルクォート文字列ではない。これこそ筆者が「.*」を乱用しないよう警告した理由の一つである。「.*」の網羅性に充分な注意を払わなかったことで、予想しなかったような結果が出ることがよくあるからだ。

ではどうすれば “McDonald’s” だけをマッチさせることができるだろうか？

もし「.*」ではなく「[^"]*」を使えば、閉じクォートを飛び越すことはない。

最初のクォートがマッチすると、「[^"]*」はできる限りのマッチを試みる。McDonald’sの後のクォートのところで、「[^"]」がこのクォートとマッチできないので、最終的にこの位置でマッチが終わる。その結果、全体がマッチに成功する。

The name "McDonald’s" is said "makudonarudo" in Japanese

5.2　複数文字からなる『クォート』

シーケンス<B>...</B>とのマッチテストは、クォートで囲まれた文字列のマッチに似ている。ただここでの『クォート』は、<B>および</B>という複数文字例になっている点が異なる。クォートで囲まれた文字列の例のように、引用符の組が複数ある場合には問題が生じる。

…<B>Billions</B>and<B>Ziillions</B> of suns…

「<B>.*</B>」を使うと、マッチ開始位置における開き引用符「<B>」に対応するものでなく、行における最後の</B>とマッチする。

5.3　一回限りのマッチ？

スターとその同種の記号（繰り返し制御文字）は網羅性を持っている。

『一回限り』であると仮定して、一回限り

注１　ドットが改行ともマッチするツールで、データが複数にまたがる文字列を含む場合には、すべての論理行をまたいで文字列末尾にまでマッチする。

のマッチの「<B>.*</B>」と、次ぎの例とのマッチを見てみよう。

…<B>Billions</B> and <B>Zillions</B> of suns…

最後にマッチが完了して、

…<B>Billions</B> and <B>Zillions</B> of suns…

スターと同種の記号の網羅性は、ある場面では非常に役立つが､別の場面では厄介な存在になることもある。一回限りマッチの構文があれば、非常に難しい作業（または不可能な作業）も可能になるので重宝である。現にPerlでは、通常の網羅型の文字に加え、一回限りのマッチ型繰り返し制御文字も提供されている。

筆者としては、この作業を２つの部分に分割し、その一つでは開きデリミタを検索し、もう一つでその位置から閉じデリミタを検索することをお勧めする。

5.4 網羅性は常にマッチを優先させる

浮動小数点表現が持つ問題により、時には“1.625”や“300”となるべき値が、“1.625000000002828”とか“3.0000000002882”になることがあった。次のスクリプトを使って変数$priceに格納された値から、小数点以下２位あるいは３位までを残して切って捨てた

```
$price =~s/(\.\d\d[1-9]?)\d*/$1/
```

「\.\d\」は最初の少数２桁にマッチし、一方「[-9]?」は第３位が０以外の場合に限ってこれをマッチする。

「最低1桁以上」を示す方法は単に「*」を「\d+」と置き換えればよい。

```
$price =~s /(\.\d\d[1-9]?)\d+/$1/
```

マッチは常にマッチ不成立よりも優先されるということである。

5.5 選択は網羅的か？

主な制御文字でまだ詳しく説明していないのは「｜」、すなわち選択である。選択の機能は正規エンジンによってそれぞれの働きが根本的に異なるため、これがどんな機能を持つかは重要な事柄である。

ＮＦＡエンジンの場合を見てみよう。「^(Subject|Date):_」という正規表現を例にとる。最初の選択肢「Subject」がテストされる。もしこれがマッチすると、正規表現の残りの部分である「:_」にチャンスが与えられる。正規表現エンジンが未実行の選択肢がまだ残っている位置までバックトラックを行うもう一つのケースである。

「tour|to|tournament」を‘three_tournaments_won’という文字列に用いた場合、最初の選択肢「tour」はマッチする。残りの選択肢はもうテストされることはない。

NFAに関する限り、選択が網羅的でないことがわかる。網羅型の選択であれば、リスト内のどの位置であっても、可能とされる最長の選択肢（「tournament」）とマッチするだろう。POSIX NFAやDFAなら、実際そうなるのである。

5.6 最小マッチ型の選択に用いる

８ページの『5.4節』の「(\.\d\d[1-9]?)\d*」の例に戻ってみよう。表現全体を「(\.\d\d|\.\d\d[1-9])\d*」と書き直すことができる。

本当にこの新しい表現は「(\.\d\d[1-9]?)\d*)と同一なのだろうか？もし選択が網羅的ならそうだが、網羅型でなければ２つはまったくの別物になる。

最小マッチ型選択の注意点

最小マッチ型の選択には、初心者の思いもよらない落とし穴がある。‘Jan 31’という１月の日付をマッチさせたい場合を考えてみよう。「Jan_[0123][0-9]」では不十分である。これだとJan_00’や‘Jan_39’といった日付が許され、‘Jan_7’は認められない。

日付の部分をマッチさせる最も単純な方法は、「Jan_(0?[1-9]|[12][0-9]|3[01])」である。こ

れは 'Jan 31 is my dad's birthday' のどことマッチするだろうか？最小マッチ型の選択は実際には 'Jan 3' としかマッチしないのである。

日付マッチを行う別の方法としては、「Jan_(31|[123]0|[012]?|[1-9])がある。ここでも選択肢の並べ方に注意する必要がある。３つ目の方法は「Jan_(0[1-9]|[12][0-9]?|3[01]?|[4-9])」である。これなら並び順とは関係なく機能する。

5.7　網羅型選択の概要

最小マッチ型の選択は、網羅型の選択よりも威力がある。

NFAの場合、選択には多くのバックトラックを伴う。選択を絞り込むことは、正規表現をより効率化することにつながる。つまり実行速度が速くなるのである。

5.8　文字クラス対選択

「[abc]」と「a|b|c」とは外見的に似ているので、文字クラスも同じように実装されていると考えるかも知れないが、NFAについては異なる。DFAでは全く同じである。

６．NFA,DFAおよびPOSIX

6.1　最長再左

可能マッチ候補の中で最も左側から最長のマッチが選ばれることから、『最長最左(Longest-Leftmost)』と呼ばれている。

真の最長

'oneselfsufficient' という文字列があった場合、「one(self)?(selfsufficient)?」という正規表現をどうマッチさせるかを考えてみよう。

従来型NFAはonseselfsufficientを返し、未実行のステートを破棄する。

日にちを組み合わせる数通りの方法

158ページ『最小マッチ型選択の注意点』で示した日付マッチの作業には何通りかの方法がある。各正規表現に対応したカレンダーには、正規表現ごとに色分けしたそれぞれの選択肢がそれぞれマッチできるものを示した。

一方、DFAはoneselfsufficientを捕まえる。すなわちDFAは可能なものの中から最も長いものを捕まえる。例えば、継続行をマッチしたいとする。

SRC=array.c builtin.c eval.c field.c gaw kmisc.c io.c main.c \ Missing.c msg.c no de.c re.c version.c

この場合、継続行をマッチするために、「(\ \ \ n.*)*」を正規表現に追加しようと考える。これは理にかなっているように見えるが、従来型NFAでは決してうまくいかない。最初の「.*」が改行に達した時には、すでにバックスラッシュを通過してしまっているのである。

6.2 POSIXと最長最左規則

POSIX標準では、同じ位置で始まるマッチが複数個ある場合、必ず一番多くのテキストにマッチするものを返すことを要求している。

POSOIX標準は『最左中の最長（longest of the leftmost)』という表現を使っている。

正規表現の書き方が不正確であると、その性能にきわめて重大な支障が起きる。その例を示す。

DFAの効率

テキスト制御型DFAは、バックトラックの非効率性をすばらしい方法で回避している。DFAエンジンは、マッチテストの前に、NFAよりも時間とメモリを使って正規表現を（しかも違った方法で）より徹底的に分析する。

6.3 DFAとNFAの比較

DFAとNFAにはともに長所と短所がある。

実行前のコンパイルにおける違い

一般的にNFAのコンパイルの方が速く、必要とする記憶領域も少ない。従来型NFAとPOSIX NFAのコンパイルでは実質的な違いはない。

マッチスピードの差

従来型NFAがマッチなしと結論するためには、正規表現のあらゆる組み合わせ経路をテストしなければならない。速くマッチするNFA正規表現の書き方については後述する。

- ●丸括弧に囲まれた部分パターンがマッチしたテキストを捕捉する。丸括弧で囲まれ各部分パターンがテキスト内のどの位置にマッチしたかを知らせる機能がある。
- ●先読み。暫定先読みは、実質的に『先に進むためにはこの部分パターンにマッチしなければならないが、テキストは消費せずにただマッチだけしてくれ』という命令ができる。否定先読みは『この部分パターンがマッチしてはいけない』という命令に対応する。
- ●最小マッチ型の繰り返し制御文字と選択（従来型NFAのみ)。DFAは最短であることを保証された全体マッチを簡単にサポートすることができるはずだが、先に述べた局所的最小マッチの機能は実装することができない。

6.4 実装し易さの違い

簡易版のDFAおよびNFAエンジンは理解も実装も簡単だ。単純であることは、必ずしも『機能が欠落』しているということではない。

７．正規表現の実践技法

高度な正規表現構築の技法を学ぶことにする。

7.1 鍵となる条件

- ●目標とマッチさせる。だが必要なものに限ること。
- ●正規表現は管理や理解がしやすいようにすること。
- ●NFAの場合は効率に気を配ること。

こうした問題は多くの場合状況に依存す

る。重要なスクリプトを扱っている場合には、適確な正規表現を書くのには時間と労力を費やすのに価値がある。スクリプト中であっても、効率は状況によって左右される。

7.2　厳密に考えてみる

９ページの『真の最長』の継続行の例をさらに続けよう。従来型NFAで「^\w+=.*(\\\n.*)*」を適用しても、次の２行にはマッチしないことがわかった。

```
SRC=array.c builtin.c eval.c field.c gaw
kmisc.c io.c main.c \missing.c msg.c no
de.c re.c version.c
```

バックスラッシュをマッチさせたくない場合、次のような正規表現を使う必要がある。

「^\w+=[^\n\\]*(\\\n[^\n\\]*)*」

IPアドレスとマッチ

これから取り上げる別の例として、IP（インターネットプロトコル）アドレスのマッチを行う。ここで要求されているのはピリオドが３つあることだけだ。

「^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]$」

「^\d\d\d\.\d\d\d\.\d\d\d\.\d\d\d$」

しかし、今度は厳密になりすぎてしまう。「\d|d\d\|[01]\d\d」となる。

これらを合わせると「2[0-4]\d|25[0-5]」という表現になる。

「[01]?\d\d?|2[0-4]\d|25[0-5]」とすることが可能だ。

「^([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])$」

これは非常に長い表記法である。それだけ有効なのだろうか？それは自分のニーズに照らし合わせて自分自身で判断しなければならない。

自分のニーズとの兼ね合いを考え、それ以上厳密にしても意味がない時点、つまり損益分岐点を判断しなければならない。

コンテキストを把握する

この正規表現を機能させるには２つのアンカーが必要であることを認識しておくことが重要だ。

7.3　困難な問題と不可能な問題

たいていのものを許したい場合、「".*"」の例で見たとおりだ。『ダブルクォート以外のものなら何でも』許したかったのである。そのために「"[^"].*"」と書くのが最も正しかった。

残念ながら、時にはそれほど明確には表現を書けないこともある。

丸括弧やブラケットなどの対になる組をマッチっさせる際には別の困難が生じる。

丸括弧内の表現をマッチさせるには、とりあえず次のような正規表現が考えられる。

１．「\(.*\)」
　　間に何かが入ったリテラルの丸括弧
２．「\([^)]*\)」
　　開き丸括弧から次の閉じ丸括弧まで
３．「\([^( )]*\)」
　　開き丸括弧から次の閉じ丸括弧までだが、途中他の開き丸括弧は許されない

図１は、これらのマッチがコードのサンプル行に対してどこでマッチするかを図示したものである。

この表現単独であれば '(this)' にマッチするが、fooの直後にならなければならないため、マッチは失敗する。

実は、正規表現では任意の入れ子構造をマッチさせることはできないという問題がある（不可能なのである）。

7.4　不要なマッチに注意する

自分が本当に意図するものを正確に表現することが重要である。浮動小数点には必ず数字が１つ以上含まれる。さもなければそれは数値ではない（！）。この正規表現を構築す

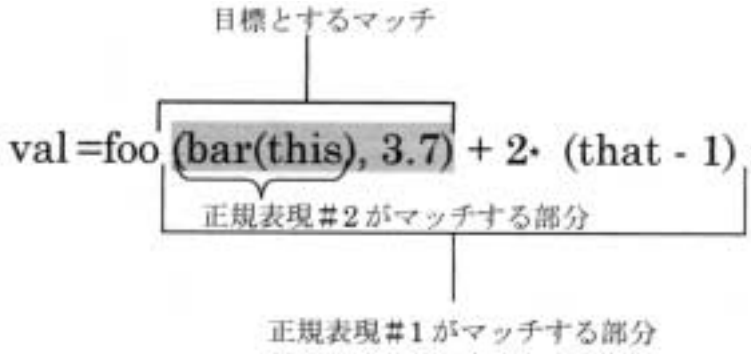

図１　正規表現例のマッチ位置

るために、

「-?([0-9]+(1\.[0-9]*)?|\.[0-9]+)」

このようにしてもとの表現は大部分改善されたが、使い方によってはまだ問題が残る。

7.5　区切られたテキストのマッチ

あるテキストによって区切られたテキストをマッチしない場合である。

●'/*' および '*/' で区切られている、Cのコメントとマッチする。

●HTMLタグとマッチを行う。これは<CODE>のように<...>で囲まれている。

●'<A_HREF="...">anchor_text</A>'

と言うリンク中の 'anchor_text' のような、HTMLタグ間の要素を抜き出す。

●.mailrcファイル中の行をマッチする。このファイルは電子メールの別名を定義するもので、各行は 'alias jeff jfriedl@ora.com' のように、次の形式に従う（ここでのデリミタは、各要素間および行末におかれた空白文字である）。

alias　省略　　完全なアドレス

●引用符で囲まれた文字列とのマッチだが、'for your passport, you need a "2 \"x3 \ "likeness" of yourself' のように、引用符がエスケープされていれば、これを含める。

一般的に、こうした作業で要求されることを流れに沿って言葉で示すと、次のようになる。

１．開きデリミタをマッチする

２．主要部のテキストをマッチする（実際には『閉じデリミタ以外なら何でもマッチさせる』ことになる）

３．閉じデミリタをマッチする

ダブルクォート文字列の中でのエスケープされた引用符を許す

開きデミリタと閉じデミリタは単純な引用符だが、閉じデミリタとはマッチさせずにどう主要部テキストをマッチさせるかが問題になる。

残念ながら、正規表現でまだ後読みをサポートしているものはない。『バックスラッシュが前に置かれたダブルクォートなら良い』とは表現できないが、『ダブルクォートが後続するバックスラッシュなら良い』であれば表現することはできる。これは「\\"」と書ける「"([^"]|\\")*"」が出来上がる。

残念なことに、これは２つの理由からうまくいかないのである。

DFAやPOSIXエンジンを使っている場合、これは問題にならない。

「"(\\?"|[^"])*"」を実行すると、ねらった通り次の部分をマッチする。

２つ目の問題は、すべてのタイプのエンジンに影響をするもので、ほとんどの場合期待通りにマッチするが、例外も存在するというような状況である。次の例を見ると、

"someone has?" forgotten?" the closing quote

正規表現がマッチしてほしくない『変則的』なケースでは、どんな結果が起こるのかを常に考慮しなければならない。重要な状況では、実際何が起こっているかを真に把握し、また万一のために網羅的なテストを行うより方法がない。

「"(\\"|[^\\"])*"」のように、最も可能性が高いケースを前に置くことができる。バックトラックをしないDFAや、どのような順序であろうとすべての組み合わせをテストするPOSIX NFAではこうしても効率はまったく同じだが、従来型NFAでは効率が向上する。

その他のエスケープされた要素を許す

もし正規表現の特性によってドットが改行とマッチしなければ、この正規表現でエスケープされた改行を許したいときに問題が生じる。

7.6　自分のデータを把握して仮定を立てる

正規表現を適用するデータや状況に対して立てた前提についての意識を持つことが肝心である。ある人にとっては当たり前の仮定でも、それが別の人にもはっきりわかるとは限らない。

7.7　全網羅型の追加例

確かに網羅性が役に立つこともある。いくつかの簡単な例を見てみよう。

ファイル名から冒頭のパスを取り除く

例えば、/usr/local/bin/gccをgccに直すように、もし変数$filenameがあれば、次の部分プログラムを使って先頭のパスをきれいに取り除くことができる。

効率面から言えば、正規表現エンジンがどうのように処理を行うかを知っておくことが重要である。

パスからファイル名にアクセスする

別の実現方法としては、パスの部分を飛び越して、単にパスを除く後続ファイル名をマッチし、このテキストを別の変数に入れる方法がある。

```
$Wholepath=~m!([^/]*)$!;
#変数$pathを正規表現でチェックせよ
$FileName = $1;
#マッチしたテキストを保存する
```

‘/usr/local/bin/prel’ と言った短い例でさえ、最終的にマッチするまで40回以上ものバックトラックが行われるのである。

正規表現の解説書だからといって、必ずしも正規表現が唯一の正解となるわけではない。たとえばTclではパス名を分解する特別なコマンドが提供されている。

先頭のパスとファイル名の両方を扱う

次の段階は、フルパスを前に置かれたパスとファイル名の部分に分割する作業である。$1には先頭のパス全体が入り、$2には後続のファイル名が入る。

大きな問題は、この正規表現が文字列中のスラッシュを最低１個は要求する点である。

```
if($WholePath = ~m !^(.*)/(.*)$!){
  $LeadingPath = $1;
  $FileName = $2;
  }else{
  $LeadingPath = ".";
  #このため “file.txt” は "./file.txt" のように見
  える
  $FileName = $WholePath;
  }
```

次のTclの部分プログラムを例に取る。

```
if [regexp -indices.*/$WholePath Match]
```

言語	部分プログラム
Perl	$filename = ~S!^.*/!!;
Tcl	regsub"^.*/" $filename " "filename
Python	filename = regsub.sub("^.*/","" ", filename)

```
{
#マッチが見つかった。マッチ末尾のインデックスを使ってスラッシュを見つけよ
set LeadingPath [string range $Whole Path 0 [expr [lindex $ Match 1] -1]] set File Name [string range $ Whole path [expr [Index $ Match 1]+1] end]
  } {
#マッチなし。名前全体がファイル名
set LeadingPath.
set FileName $Whole Path
  }
```

やはりこの例でも、最後のスラッシュを見つけるのに正規表現を使うのは無駄である。rindexあるいはそれに類する関数を使ったほうが速い。

８．まとめ

8.1　マッチメカニズムのまとめ

正規表現エンジンを実装する上で、一般的に２つの基本技術が用いられている。『正規表現制御型NFA』と『テキスト制御型DFA』

●従来型NFA
（消費型で、強力な）エンジン

●POSIX NFA
（消費型だが標準に依拠した）エンジン

●DFA（POSIX及び非POSIX）
（省エネ型）エンジン

効果を最大限に引き出すには、どのタイプのエンジンが使われているかを理解し、正規表現を適切に工夫する必要がある。

DFAテキスト制御エンジン
可能な最長のマッチを見つける。この一言につきる。一貫性があり極めて高速だが、正規表現の違いによる動作の違いは現れない。

NFA正規表現制御型エンジン
『努力を重ねて』マッチを見つける。
NFAマッチングの心臓部はバックトラックである。

POSIX NFA
自動的に最長マッチを見つける。だが、効率を考慮しなければならないので、解説するのは意味がある。

従来型NFA
正規表現制御型というエンジンの性質を生かし、必要なマッチをずばり工夫できるので、最も表現力豊かな正規表現エンジンといえる。

8.2　マッチメカニズムの実際上の効果

できるだけ厳密に考えるように心がけ、どういう時に不要なマッチが入り込むか注意する必要がある。（NFAでは）効率性の間でバランスを取ることがしばしば要求される。

NFAの場合、効率性が極めて重要になることから、次は、効率的なNFA正規表現を工夫する方法を考えてみる。

あとがき

大きく分類されたNFAとDFAの２つのエンジンについて、その表現の仕方によってどのような動作を行うかについて見てきた。同様な表記でもまったく異なるマッチを行う。また、微妙な表記の違いによって有効なマッチを行う場合もあれば、無駄な努力になる場合もあることを見てきた。すなわち、用いるエンジンを充分に知ることによってのみ、その性能を充分に引き出せることをも学んだ。今後具体的に個々の『正規表現』エンジンについて、どのような工夫が有効であるかについて検討してみたい。

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ー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－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.