正規表現関数による正規表現の拡張とそのパターンマッチングへの応用

山本 篤[†] 山口 和紀[†]

正規表現は，パターンマッチングを行うためのツールとして広く利用されている．しかし，さまざまな応用で拡張されてきたのにともない，次のような問題が出てきている．1) 標準的に使われている正則集合による意味づけでは，後方参照がうまく定義できていない，2) オートマトンを用いたパターンマッチングの実装において，状態数やバックトラックの回数が爆発することがある，3) 正規表現の積や差を直接的に利用できない．本研究では，このような問題を解決するために，正規表現関数と呼ぶ関数を導入する．正規表現関数は，記号列集合を入出力とする関数であり，マッチする記号列を消費して出力するものである．たとえば，正規表現 $a*$ が a の繰返しにマッチすることは，その正規表現関数が，$a*(\{ab, aa, b\}) = \{ab, b, aa, a, \epsilon\}$ という入出力関係を持つことで表される．これを拡張し，変数を扱えるようにすることで，後方参照も含めた正規表現を定義することができる．また，正規表現関数を用いたパターンマッチングの実装が可能であり，後方参照のない場合には計算量の爆発を避けることができ，比較実験でも優位なケースを確認した．さらに，正規表現の積と差を導入し，これらが正規表現関数によって簡単に実装できることを示す．最後に，正規表現の積や差を用いる応用例として HTML などへのパターンマッチングをあげる．

"Regular Expression Function" for Extending Regular Expressions and Its Application to Pattern Matching

Atsushi Yamamoto[†] and Kazunori Yamaguchi[†]

Regular expressions have been used widely for pattern matching. However the following problems are getting serious in some applications. 1) The definition of regular expressions cannot be extended to back reference, which is a popular extension of regular expressions. 2) The implementations of pattern matching in automata suffer from the explosion of time or space complexity for some regular expression patterns. 3) In the conventional regular expressions, we cannot use intersection and difference operators, which are useful in some applications. In this paper, we introduce a "regular expression function" from a set of strings to a set of strings. This function eliminates matching prefixes with given regular expressions from the input strings and outputs the remaining postfixes. For example, $a*(\{ab, aa, b\}) = \{ab, b, aa, a, \epsilon\}$. This function can be extended to give a semantics to regular expressions with back references. Then, we show that we can perform pattern matching by interpreting the regular expression function directly without the explosion of time and/or space complexity, which is confirmed by our preliminary implementation. We also introduce intersection and difference operators and show that the regular expression function can be extended to handle these operators easily. Finally, we briefly show some possible applications of the operators.

1. 導 入

1.1 正規表現について

正規表現は複雑なパターンを比較的効率良く検索できる表現であり，エディタやプログラミング言語，ユーティリティなどにその機能が組み込まれ，広く利用されている．正規表現の意味は正則言語として厳密に定義されており，その処理については，オートマトンとの関係が詳しく研究され[1),2)]，実装方法についての研究[3)]も進んでいる．

しかしながら，正規表現が拡張されて利用されてきているのにともない，以下のような問題が生じている．

定義の問題 正規表現の意味は正規表現に対応する正則集合を定義することにより与えられる．しかし，この方法では正規表現の比較的一般的な拡張である後方参照の意味を正確に定義することが難しい．たとえば，Aho[4)]による定義では，後方参照に用いる変数への値の束縛と変数の値の参照の順序が定義されていないため，参照する値が何かが定義できていない．

処理の問題 ある記号列やその接頭語が正規表現の表

† 東京大学大学院総合文化研究科広域科学専攻広域システム科学系
Department of Graphic and Computer Sciences, Graduate School of Arts and Sciences, The University of Tokyo

す正則集合に含まれるかどうかを判定することをパターンマッチングと呼ぶ．パターンマッチングには有限状態オートマトンを使うのが一般的であるが，DFA（決定的有限状態オートマトン）を用いると状態数が爆発することがあり，後方参照の実装も非常に困難になる．一方，NFA（非決定的有限状態オートマトン）を用いるとバックトラックの回数が爆発することがある．

表現の問題 正規表現 r と s に対して,「r か s にマッチするもの」は $r|s$ で表現できるが,「r と s にマッチする」あるいは「r にマッチして s にマッチしない」ものを表すのは容易ではない．正則言語は積（共通部分）と差について閉じているので，そのような正規表現は存在するはずであるが，もとの r や s との関係が $r|s$ のように直接的ではないために利用しにくい．そのため，積や差は利用されてこなかったが，実際には次のように積や差を用いると簡単に表現できるパターンがある．

- 100 文字以下の URL
- ある特定の IP アドレス以外の IP アドレス

このようなパターンは実用上も価値があるが,「URL」「100 文字以下」「IP アドレス」「特定の IP アドレス」などが正規表現で表現できても，それらを直接組み合わせることによって求めるパターンを記述することはできない．

Perl などに実装されている「先読み肯定」には積と似た効果があるが，これは正規表現外の機能であり，アドホックな拡張にとどまっている．積や差は選択と同等の演算子として定義できる．

1.2 本論文の流れ

本研究では，以上の 3 つの問題点を解決する．

まず，定義の問題については，正規表現の意味を，正則集合としてではなく正規表現関数と名づけた関数によって与えることにより解決する（2 章）．

次に，処理の問題に関しては，正規表現関数を実装することが可能であることを示し，それによってオートマトンを用いるときに生じる問題点が解決されることを示す（3 章）．

最後に，正規表現関数の実装を用いることで，正規表現に積や差を導入することが可能であることを示し（4 章），その応用として，HTML タグのマッチングとシャッフル演算への対応について述べる（5 章）．

2. 正規表現関数

2.1 正規表現関数の基本的定義

まず，以下で使う記法を定義する．記号の集合を Σ とする．$\Sigma^* = \bigcup_{i \geq 0} \Sigma^i = \{\epsilon\} \cup \Sigma \cup \Sigma\Sigma \cdots$ の要素を記号列と呼ぶ．ただし ϵ は空記号列である．$a, b \in \Sigma^*$ に対して左商 $\backslash$ と右商 $/$ を,

$$a \backslash b = \begin{cases} c & c \in \Sigma^*, a = bc \\ \bot & \text{otherwise} \end{cases}$$

$$a / b = \begin{cases} c & c \in \Sigma^*, a = cb \\ \bot & \text{otherwise} \end{cases}$$

と定義する．$I, J \subseteq \Sigma^*$ に対しては，$I \backslash J = \{b \mid ab \in I, a \in J\}$，$I/J = \{a \mid ab \in I, b \in J\}$ と定義する．$\backslash$ と $/$ を使うと，記号列 c の接頭語の集合 H_c は $H_c = \{c\}/\Sigma^*$，接尾語の集合 T_c は $T_c = \{c\}\backslash\Sigma^*$ と表される．

次に正規表現を次のように構成する．

空記号列 ϵ

記号 a，ここで a は記号．

連接 rs，ここで r と s は正規表現．

選択 $r|s$，ここで r と s は正規表現．

閉包 $r*$，ここで r は正規表現．

以下で正規表現に対してその意味を関数として定義し，それを正規表現関数と呼ぶことにする．

正規表現 r に対して正規表現関数 $r\colon \wp(\Sigma^*) \rightarrow \wp(\Sigma^*)$ を次のように定義する☆．

$$r(I) = \begin{cases} I & \text{if } r = \epsilon \\ I \backslash \{a\} & \text{if } r \text{ が記号 a} \\ t(s(I)) & \text{if } r = st \\ s(I) \cup t(I) & \text{if } r = s|t \\ \bigcup_{i \geq 0} s^i(I) & \text{if } r = s* \end{cases}$$

例を示す．

(1) $a(\{ab, ac\}) = \{b, c\}$

(2) $a(\{bc\}) = \emptyset$

(3) $(ab)(\{abc\}) = b(a(\{abc\})) = b(\{bc\}) = \{c\}$

(4) $(a|b)(\{ab\}) = a(\{ab\}) \cup b(\{ab\}) = \{b\} \cup \emptyset = \{b\}$

(5) $(a|ab)(\{ab\}) = a(\{ab\}) \cup ab(\{ab\}) = \{b\} \cup \{\epsilon\} = \{\epsilon, b\}$

(6) $(a*)(\{aa, a\}) = \bigcup_{i \geq 0} a^i(\{aa, a\}) = \{\epsilon, a, aa\}$

正規表現 r に対応する正則集合を $\bar{r}$ とすると次の性質が成り立つ．

$$c \in \bar{r} \Leftrightarrow \epsilon \in r(\{c\}) \quad (1)$$

$$\bar{r} = \{c \mid c \in \Sigma^*, \epsilon \in r(\{c\})\} \quad (2)$$

$$r(\{c\}) = \{x \mid c \in \Sigma^*, c/x \in \bar{r}\} \quad (3)$$

$$H_c \cap \bar{r} \neq \emptyset \Leftrightarrow r(\{c\}) \neq \emptyset \quad (4)$$

正規表現を関数としてとらえる先行研究には以下の

☆ 使い方で分かるので，正規表現の r と正規表現関数の r に同じ記号を用いる．また，$\wp(A)$ は A の冪集合とする．

ものがある．

パーザを実装するために行われたHuttonの研究[5]では，記号列を入力し，マッチする部分を消費して，残った記号列を出力とするすような関数を用いている．複数のマッチの可能性があるので，それらを列挙した記号列のリストを返す．関数の入力は集合ではなく，記号列であり，関数の出力と入力のデータ形式が異なっており，正規表現関数のように連接を関数合成で扱うようなことはできない．また，関数の出力はリストであり，正規表現関数のように集合ではないため，3.4.1 項のような実装上の工夫を適用しにくい．

また，定理の証明のために行われたHarper[6]の研究でも，正規表現を関数と見なしているが，関数の出力は真偽値であり，正規表現関数のように入力と出力の整合性はない．

2.2 後方参照つき正規表現関数

後方参照つき正規表現では，2.1 節の正規表現の定義に以下を追加する．

変数参照 α，ここで α は変数．

変数定義 $r\%\alpha$，ここで r は正規表現，α は変数．

後方参照つき正規表現に対しては，正規表現関数の定義を次のように拡張する．

まず，変数の集合を V とする．2.1 節で定義した正規表現関数の入力の記号列は特別な変数 γ の値として入力することにして，$V_a = V \cup \{\gamma\}$ と定義する．部分関数 $f\colon V_a \to \Sigma^*$ の集合を $Partial(V_a, \Sigma^*)$ とする．この関数 f による変数 x の値 $f(x)$ を，変数 x に束縛された値と呼び，その意味で関数を束縛と呼ぶ．

$r\colon \wp(Partial(V, \Sigma^*)) \to \wp(Partial(V, \Sigma^*))$ として正規表現関数を定義しなおす．この r の定義域と値域は，$f_1, f_2, .., f'_1, f'_2, \ldots \in Partial(V_a, \Sigma^*)$ に対して，$r(\{f_1, f_2, \ldots\}) = \{f'_1, f'_2, \ldots\}$ となるものである．

正規表現の定義の拡張に応じて正規表現関数を次のように拡張する．ただし $\alpha \in V$ とする．

$$r(I) = \begin{cases} \cdots \\ \{g \mid f \in I, g(\gamma) = f(\gamma)\backslash f(r), \\ \quad g = f \text{ except for } \gamma\} \text{ if } r = \alpha \\ \{g \mid f \in I, h \in s(\{f\}), \\ \quad g(\alpha) = f(\gamma)/h(\gamma), \\ \quad g = h \text{ except for } \alpha^{☆}\} \text{ if } r = s\%\alpha \end{cases}$$

正規表現関数の定義域と値域が変わったことに応じて次の定義を変更する．

☆ $g(\alpha)$ は $h(\alpha)$ ではなく $f(\gamma)/h(\gamma)$ に束縛される．これにより，$(a\%\alpha b)\%\alpha$ では，α の値は（$a\%\alpha$ で α に束縛された値 a が捨てられて）ab に束縛される．

$$r(I) = \begin{cases} \cdots \\ \{g \mid f \in I, g(\gamma) = f(\gamma)\backslash\{\mathrm{a}\}, \\ \quad g = f \text{ except for } \gamma\} \text{ if } r \text{ が記号 a} \end{cases}$$

例を示す．以下，$a, b, c \in \Sigma$，$\alpha, \beta \in V$ とし，$f(a_i) = b_i$ である部分関数 $f\colon V_a \to \Sigma^*$ を $\{a_1 \mapsto b_1, a_2 \mapsto b_2, \ldots\}$ で表す．

(1) $a(\{\{\gamma \mapsto ab\}\}) = \{\{\gamma \mapsto b\}\}$

(2) $\alpha(\{\{\gamma \mapsto a, \alpha \mapsto a\}\}) = \{\{\gamma \mapsto \epsilon, \alpha \mapsto a\}\}$

(3) $\alpha(\{\{\gamma \mapsto a, \alpha \mapsto b\}\}) = \emptyset$

(4) $a\%\alpha(\{\gamma \mapsto ab\}) = \{\{\gamma \mapsto b, \alpha \mapsto a\}\}$

(5) $(a\%\alpha|a\%\beta)(\{\{\gamma \mapsto ab\}\}) = \{\{\gamma \mapsto b, \alpha \mapsto a\}, \{\gamma \mapsto b, \beta \mapsto a\}\}$

(6) $(a|b)\%\alpha(\{\{\gamma \mapsto a\}, \{\gamma \mapsto b\}\}) = \{\{\gamma \mapsto \epsilon, \alpha \mapsto a\}, \{\gamma \mapsto \epsilon, \alpha \mapsto b\}\}$

(7) $(a|b)\%\alpha\alpha(\{\{\gamma \mapsto aa\}, \{\gamma \mapsto bc\}\}) = \alpha(\{\{\gamma \mapsto a, \alpha \mapsto a\}, \{\gamma \mapsto c, \alpha \mapsto b\}\}) = \{\{\gamma \mapsto \epsilon, \alpha \mapsto a\}\}$

(8) $(a\%\alpha)(b\%\alpha)(\{\{\gamma \mapsto ab\}\}) = b\%\alpha(\{\{\gamma \mapsto b, \alpha \mapsto a\}\}) = \{\{\gamma \mapsto \epsilon, \alpha \mapsto b\}\}$

この例に示されているように，正規表現関数では，連接の関数合成によって左から右への束縛・参照の流れが表現されているため，左側で束縛された変数の値を右側で参照することが可能となっている．

後方参照つき正規表現のAho[4]による定義では，左から右への流れが表現されていないため，たとえば同じ変数に複数回束縛がおこる場合にどの値を使うかが規定できていない．

また，XMLデータに対しての変数を含む正規表現の定式化が，Hosoyaら[7]により行われているが，変数の値を参照したり，変数の値を変更したりするというものではなく，後方参照を用いた記号列パターンマッチングにそのまま適用することはできない．

3. 正規表現関数を用いたパターンマッチングの実現

3.1 有限状態オートマトンを用いたパターンマッチング

パターンマッチングで，与えられた記号列の接頭語とのマッチを求めることを考える．このとき，一般には複数のマッチの可能性があるが，マッチを1つ求めればよい場合もあれば，最長のマッチを求める必要があることもある．POSIXの正規表現のパターンマッチング[8]では，後者である必要がある．

正規表現のパターンマッチングには有限状態オートマトンを用いることが多いが，この有限状態オートマトンを使ったパターンマッチングの主なアルゴリズム

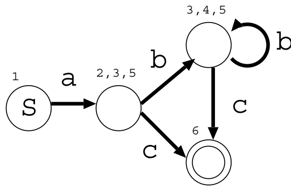

図 1　ab*c に対応する DFA
Fig. 1　A DFA for ab*c.

は次の 4 種類である．

- DFA
- 従来型 NFA
- POSIX NFA
- 状態集合型 NFA

DFA は，決定的有限状態オートマトンを用いるものである．たとえば，**図 1** は，正規表現 $ab*c$ に対する DFA である．

DFA を用いたアルゴリズムでは，初期状態（S）から始まり，記号列の先頭から記号に従って遷移し，受理状態（◎）に到達すれば，パターンにマッチするものとしてそれまでに消費した入力記号列を受理する．マッチする最長の接頭語を求めるには，受理状態に到達した後も可能な限り遷移を続け，最後に受理状態に到達したときに消費していた入力記号列の部分をとる．

この方法は記号列長に比例する時間でマッチングが行えるため高速であるが，後方参照の実装が難しいという欠点がある．また，DFA は後で述べる NFA と同等の能力があるが，これは NFA の複数の状態の組合せを 1 つの状態で表すことによっているので，パターンによっては状態数が爆発するという問題がある．

従来型 NFA，POSIX NFA と状態集合型 NFA は，非決定的有限状態オートマトンを用いるものである．たとえば，**図 2** は，正規表現 $ab*c$ に対する NFA である．

非決定的に到達しうる状態を求めるためには，バックトラックを用いる方法と到達しうる状態をすべて求めていく方法[9)]がある．従来型 NFA と POSIX NFA は前者であり，状態集合型 NFA は後者である．

従来型 NFA は，マッチする接頭語を 1 つ求めたら終了する方法であり，POSIX NFA はマッチする接頭語をすべて求めるものである．マッチの可否のみを知りたい場合，あるいは任意のマッチする接頭語を求めればよい場合には従来型 NFA でよいが，マッチする接頭語の中で最長のものを求めたい場合には POSIX NFA を用いる．

バックトラックを用いるアルゴリズムは，閉包 $*$ の組合せがあるとバックトラックが多発することがあり，

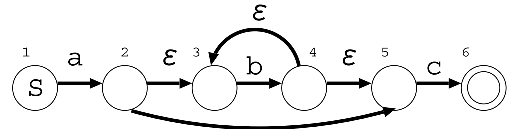

図 2　ab*c に対応する NFA
Fig. 2　A NFA for ab*c.

その場合にはマッチングに非常に時間がかかる．特に POSIX NFA ではその傾向が顕著である．

状態集合型 NFA は，バックトラックにともなう問題は起きないが，記号列を読むたびに遷移可能な状態をすべて求めなければならないので，その部分のコストが大きい．また，DFA と同様に後方参照などの拡張を行うことが困難になるという問題がある．

3.2　正規表現関数を用いたパターンマッチング

ある有限長の記号列 c が正規表現 r にマッチするかどうかは 2 章の式 (1) を用いて，$r(\{c\})$ が ϵ を要素として含むか否かを調べればよい．また，c の接頭語が正規表現 r にマッチするかどうかは式 (4) の性質を用いて $r(\{c\})$ が空でないかどうかを調べればよい．

したがって，$r(I)$ を具体的に計算するアルゴリズムがあれば，それらのマッチを調べることができる．さらに，$r(I)$ が具体的に定まれば，その中にはすべてのマッチの可能性が入っているため，マッチする最長の接頭語を求めることもできる．

ここでは後方参照のない正規表現 r に対する，$r(I)$ を計算するための疑似コードを以下に示す．

$r(I) = \{$
　switch(r)
　case $\epsilon \Rightarrow I$
　case a(記号) $\Rightarrow I \backslash \{a\}$
　case $st \Rightarrow t(s(I))$
　case $s|t \Rightarrow s(I) \cup t(I)$
　case $s* \Rightarrow T$ such that
　　$T \leftarrow \emptyset, U \leftarrow I$
　　while $U - T \neq \emptyset$
　　　$T \leftarrow T \cup U$
　　　$U \leftarrow s(U)$
　　end
　end
$\}$

閉包 $s*$ に対して，$s*(I)$ は $\bigcup_{i\geq 0} s^i(I)$ と定義されているが，これは一般には計算できない．しかし，有限長の記号列に対しては必ず計算が終了するので，上記のアルゴリズムは有限時間内に実行可能である．

正規表現 $ab*c$ に対応する正規表現関数を表したのが**図 3** である．□は 1 記号のマッチングする正規表

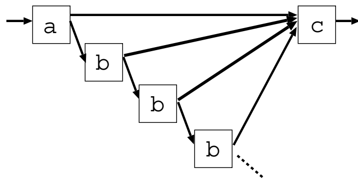

図 3　ab*c に対応する正規表現関数
Fig. 3　A regular expression function which corresponds to ab*c.

現関数（□中の記号にマッチする）である．閉包は 1 記号のマッチングに展開して表してある．

3.3 計 算 量

以下では，正規表現の長さを R とし，対象記号列の長さを N とする．また，後方参照は考えない．

3.3.1 有限状態オートマトンを用いた場合の計算量

有限状態オートマトンを作るための時間計算量は NFA の場合には $O(R)$ であり，DFA の場合には最悪の場合には $O(2^R)$ となる．

有限状態オートマトンによるマッチングの時間計算量は，DFA では $O(N)$ であるが，NFA では，最悪の場合には $O(R^N)$ となる．これは `(((.*)*)*)*`☆のようなパターンの場合である．なお，NFA に関しては，バックトラックを用いず，到達しうる状態をすべて求める状態集合型 NFA の場合の計算量は $O(R \times N)$ であることが知られている[9]．

空間計算量は，DFA では，最悪の場合は状態数が正規表現の大きさに対して指数的になる．遅延評価を行うことでこの問題を回避する方法も使われているが，時間計算量が増大し，空間計算量と時間計算量のトレードオフとなる．また，実装も複雑となる．

3.3.2 正規表現関数を用いた場合の計算量

正規表現関数を用いた場合はオートマトンの作成は必要なく，パーズを行うのみであるが，この計算量は $O(R)$ となる．

また，マッチングにおいて正規表現関数の呼び出しの回数は $O(R)$ である．それぞれの正規表現関数の処理は，扱う集合の大きさがたかだか $N+1$ であるため，閉包演算以外の処理時間は $O(N)$ 以下である．閉包演算はたかだか $N+1$ 回の集合演算で終了する．以上をまとめると，正規表現関数でマッチングを行った場合の時間計算量は $O(R \times N^2)$ となる．

3.3.3 計算量のまとめ

各アルゴリズムの計算量と後方参照の実装の容易さについてまとめると**表 1** のようになる．

表 1　パターンマッチングアルゴリズムの計算量
Table 1　Complexity of pattern matching algorithms.

アルゴリズム	時間計算量	空間計算量	後方参照
DFA	$O(N+2^R)$	$O(2^R)$	困難
従来型 NFA	$O(R^N)$	$O(R)$	容易
POSIX NFA	$O(R^N)$	$O(R)$	容易
状態集合型 NFA	$O(R \times N)$	$O(R)$	困難
正規表現関数	$O(R \times N^2)$	$O(N)$	容易

正規表現関数は，N^2 のオーダが登場するが，バックトラック法を用いる場合と比較すれば最悪の場合の性質は良い．

3.4 正規表現関数に関連する技術

3.4.1 集合のビット列表現による高速化

マッチングを $r(\{c\})$ で始めるとすると，$r(\{c\})$ の処理中に現れる記号列集合の要素は，記号列 c の接尾語である．記号列 c の接尾語の集合は長さ $N+1$ のビット列で表現可能である．たとえば，長さ i の接尾語が接尾語の集合に含まれているかどうかを，i 番目のビットで表現すればよい．

このビット列による集合の表現を用いると，集合の和，差，積がビット演算を用いて計算でき，1 記号読み進めるのはシフト演算で計算できるため，処理が効率的に行える．

テキストに対するパターンマッチングのような応用で，N の大きさに制限がある場合は，この方法が適用できる．

ビット演算可能なワード長（ただし，一般に N よりも小さいと考える）を W とするならば，集合の和は $O(N/W)$ で計算可能である．したがって，閉包演算の処理の計算量は $O(N^2/W)$ となるので，正規表現関数の計算時間は $O(N^2/W)$ といえる．よって，マッチング全体の計算量は $O(R \times N^2/W)$ となる．

このようなビット演算を用いた計算を適用する場合，状態集合型 NFA を用いるアルゴリズムでも Myers[10] によって $O(RN/W)$ の計算量でマッチングできることが示されており，同様にワード長だけ高速となる．

3.4.2 任意の部分文字列のマッチの判定

今までのマッチングはある文字列の接頭語のマッチを判定するものであった．任意の部分文字列のマッチを判定するためには，記号列の接尾語のそれぞれに対して，接頭語のマッチを判定する方法が考えられる．記号列 c の長さを N とし，長さ i の c の接尾語を $c_i (\in T_c)$ とすると，オートマトンの場合は，$c_N, c_{N-1}, \ldots, c_0$ をオートマトンに入力して受理されるかどうかの判定を行うことになる．正規表現関数を用いる場合には

☆ . はすべての記号を選択で結合したものを表す略記法で，意味的には任意の 1 記号を表す．

$r(\{c_i\})$ が空であるかどうかを $c_N, c_{N-1}, \ldots, c_0$ について判定することになる．したがって，マッチの判定に要する時間計算量は，文字列の接頭語のマッチングの N 倍のオーダとなる．あるいは，正規表現の先頭に`.*`[☆1]を補ってマッチさせることも可能だが，追加によって DFA の状態数が指数的に増加する可能性がある．状態集合型 NFA も同等の問題が起きる．またバックトラックを用いる場合は，追加部分のバックトラックが，各接尾語を順番に入力するのとほぼ等しい意味を持つ．

しかし，正規表現関数を用いる場合は，$\bigcup_{i=0}^{N} r(\{c_i\})$ が空であるかどうかを判定することで，その行にマッチが存在するかどうかを知ることができるので，

$$\bigcup_{i=0}^{N} r(\{c_i\}) = r(\bigcup_{i=0}^{N}\{c_i\}) = r(T_c)$$

を使うと，$r(T_c)$ が空であるかを確認すればよいことになる．$r(T_c)$ は，3.4.1 項の方法で，初期入力としてすべての $N+1$ ビットを立てたものを関数に与えることによって容易に実現できる．このアルゴリズムの場合は，与えられた正規表現とマッチするものが記号列の途中にあるかどうかの判定に要する時間計算量は，特定の文字列のマッチングと同じ $O(R \times N^2)$ で済む．

なお，このアルゴリズムではマッチの終了位置しか分からないため，開始位置が必要な場合は，終了位置から開始位置を調べることになる．これはオートマトンの場合は，オートマトンを逆にたどることで可能である[☆2]が，正規表現関数では次のように実行する．

まず，正規表現 r と記号列 c を反転させた r'，c' に対して $r'(T_{c'})$ を計算する．この $r'(T_{c'})$ の要素 c'' はマッチの開始位置までの記号列を反転したものであるので，$r'(T_{c'})$ の要素を反転することでマッチの開始位置が分かる．特に，最左マッチ（記号列の先頭から一番近い位置から始まるマッチ）を求めたい場合は，$r'(T_{c'})$ の要素の中で最短の c'_{min} を求める．c'_{min} を反転させた c_{min} が，最左マッチの開始位置であり，$c \backslash c_{min}$ が最左マッチである．最左マッチの終了位置は $r(\{c \backslash c_{min}\})$ で分かる．

3.5 計算速度の実験

ここでは，正規表現関数と従来型 NFA，POSIX NFA，状態集合型 NFA を用いたアルゴリズムの計算速度の比較を行った．DFA はここでは比較の対象としなかった．

3.5.1 実験の条件

Java を用いて，正規表現関数，従来型 NFA，POSIX NFA，状態集合型 NFA に関して実装を行った．ただし，正規表現関数については，3.4.2 項で述べた技術を (a) 使っていないものと (b) 使ったものの 2 種類を試している．(a) は，各接尾語に対してマッチングを行っている．(b) は，マッチ開始位置を求める処理は行っていない．また，3.4.1 項で述べたビット列による集合の表現はどちらの実装でも採用している．従来型 NFA，POSIX NFA による 2 つの実装は，ほとんどコードが共有されている．状態集合型 NFA は，状態集合をビット列で表現する手法をとっており，正規表現関数で用いたのと同じ多倍長ビット列のライブラリを利用している．

また，5 種ともほぼ同等の機能を有しており，一部に最適化[☆3]を行っている．ただし，後方参照の実装は行っていない．

実験は Duron 850 MHz，メモリ 1 GB の PC で，Linux2.4.18 上の J2SDK1.4.1 を用いて，上述の 5 種類の実装によるパターンマッチを行った．対象としたデータは，1 行が平均して 210 文字程度からなる Apache のアクセスログ 20,000 行である．このデータに対していくつかのパターンをマッチさせ，その時間を UNIX の time コマンドによって計測した．結果に示したのは，5 回計測した平均値から，0 行のデータに対するマッチングの時間を除いたものである．

3.5.2 実験結果

パターンマッチングの実験結果を**表 2** に示す．

ここでは，パターンの記述に，よく使われる以下の略記法を使っている．`r{n}`は r を n 回連接したもの，`r+`は r の 1 回以上の繰返し $(rr*)$，`[a-z]` は指定された範囲の記号の選択 $(a|b|c\ldots x|y|z)$[☆4]，`[^...]` は... 以外の記号の選択，`.` は Σ のすべての記号の選択である．

`^`は行頭を表す特殊記号である．正規表現の先頭に`^`がない場合は，行途中からのマッチングも行う．`$`は行末を表す特殊記号である．

結果のなかの,「Java 組み込み」は，Java 組み込みの正規表現（従来型 NFA）の結果を参考のために示したものである．

N/A は数分待っても 1 行のマッチングも終了しなかったことを示す．

☆1 ピリオドは Σ のすべての記号の選択を意味する．

☆2 ViVi [11)] で実装されている．

☆3 固定記号列検索など．

☆4 実装上では，高速化のために [a-z]$(I) = I \backslash \{a, b, \cdots, z\}$ という処理を行っている．次の 2 つも同様である．

表 2 パターンマッチングの実行時間
Table 2 Execution time of pattern matching.

パターン	Java 組み込み	正規表現関数 (a)	正規表現関数 (b)	従来型 NFA	POSIX NFA	状態集合型 NFA	マッチ行数
`(a|b|c|d|e|f){4}`	3.39 s	16.88 s	2.50 s	30.06 s	31.15 s	11.18 s	439
`[a-f]{4}`	0.84 s	2.15 s	1.30 s	3.03 s	3.12 s	3.17 s	439
`[ab]d+`	1.04 s	1.81 s	1.02 s	1.51 s	1.53 s	3.11 s	1541
`a.+`	1.62 s	6.35 s	2.93 s	9.81 s	16.06 s	1.42 s	20000
`.+`	1.58 s	6.84 s	1.30 s	10.32 s	17.28 s	1.21 s	20000
`.+.+`	1.51 s	6.96 s	1.38 s	10.55 s	1680.32 s	1.24 s	20000
`(.+)+`	1.74 s	7.29 s	1.57 s	10.87 s	N/A	1.20 s	20000
`^(.+)[^"]$`	1.51 s	6.53 s	6.51 s	22.25 s	22.38 s	9.55 s	0
`^(.+)+[^"]$`	N/A	6.68 s	6.70 s	N/A	N/A	9.36 s	0

従来型 NFA では，マッチングが成功した時点で計算を終了できるが，POSIX NFA はすべてのマッチングを調べなければならないので，マッチングに成功しても終了することができない．そのため，`(.+)+`のマッチングは，従来型 NFA が 10.87 秒で終了しているのに，POSIX NFA では N/A となってしまっている．

ここで対象としたアクセスログは行末が必ず`"`であるが，最後の 2 パターンは`$`の直前が`"`以外でなければならないと指定しているので，マッチングに失敗する．マッチングに失敗する場合は，従来型 NFA も途中で終了することができないので，パターン`^(.+)+[^"]$`では，POSIX NFA だけでなく従来型 NFA も N/A となっている．

状態集合型 NFA は，どの正規表現に対しても平均的な速度でマッチング可能である．

正規表現関数を用いたアルゴリズムは，バックトラックを行わないので，`^(.+)[^"]$`と`^(.+)+[^"]$`で大きく時間が変わるようなことはない．また，速度的には，ほとんどのパターンに対して，20,000 行を数秒で処理可能であり，このテスト実装でも十分に実用的であるという結果が得られた．特に，正規表現関数でも (b) のほうは非常に高速である．

なお，正規表現関数の出力はマッチの可能性をすべて尽くしているので，最長なものや最短なもの[12]を必要に応じて選択することができる．

4. 正規表現の積と差の処理

4.1 正規表現関数の積と差への拡張

正規表現 r と s のどちらかにマッチする記号列を表現するためには選択 $r|s$ を用いる．これは，集合としての正規表現でいえば $\bar{r} \cup \bar{s}$ にあたる．しかし，r と s の両方にマッチするもの，あるいは r にマッチして s にマッチしないものなどは簡単には表せない．これらは集合で表現すれば $\bar{r} \cap \bar{s}$ や $\bar{r} - \bar{s}$ となり，正則集合は補集合と共通部分に対して閉じているので，対応する正規表現が存在するはずであるが，その正規表現が r や s との関係が選択 $r|s$ のように単純ではなく，パターンの記述には使いにくい．

そこで，$\bar{r} \cap \bar{s}$ と $\bar{r} - \bar{s}$ に対応する正規表現として，積と差を導入する．

積 $r\&s$，ここで r と s は正規表現である．

差 $r - s$，ここで r と s は正規表現である．

正規表現関数の定義は次のように拡張する．

$$r(I) = \begin{cases} \cdots \\ (s\&t)(I) = s(I) \cap t(I) & \text{if } r = s\&t \\ (s-t)(I) = s(I) - t(I) & \text{if } r = s-t \end{cases}$$

4.2 有限状態オートマトンによる積と差の処理

積や差に対応する有限状態オートマトンは，状態集合の直積を状態集合とすることで構成可能であるが，状態集合が非常に大きくなる．有限状態オートマトンの最小化の方法で不要な状態をなくすことはできるが，最小化に時間がかかり，状態数が減らないものもある．

状態集合の直積を作らずに，有限状態オートマトンの機能を拡張して対応することも考えられるが，後方参照などが可能となる有効な拡張は知られていない．

4.3 正規表現関数による積と差の処理

正規表現の積 $r\&s$ と正規表現の差 $r - s$ は正規表現関数を用いて次のように計算できる．

$r(I) = \{$
  switch (r)
  ...
  case $s\&t \Rightarrow s(I) \cap t(I)$
  case $s - t \Rightarrow s(I) - t(I)$
  end
$\}$

これらは既存の正規表現の枠組でも表現可能であるため，言語的な拡張を行っているわけではない．しかし，有限状態オートマトンと異なり，これを実装するのは非常に容易である．$r|s$ で集合の和をとっている部分を集合の共通部分，あるいは差をとるように変え

るだけである．

積や差のみでなく，他の集合演算に関しても同様に実装可能である．

4.4 後方参照つき正規表現の積と差への拡張

後方参照がある場合の積と差での変数の値の扱いについて検討する．まず，次のような正規表現を考える．

```
<font (.*size=("\w+")%α.*)&
(.*color=("\w+")%β.*)>
```

この場合，変数 α にはサイズを表す記号列を束縛し，変数 β には色を表す記号列を束縛し，積の全体にマッチングしたところで，α と β の両方を束縛しておきたい．ところが，$((r\%\alpha)\&(s\%\beta))(I)$ を単に $(r\%\alpha)(I) \cap (s\%\beta)(I)$ と計算すると，$(r\%\alpha)(I)$ と $(s\%\beta)(I)$ に共通に含まれる束縛しか残らないため，このようなものを求めることができない．

そこで，$(r\%\alpha)(I)$ と $(s\%\beta)(I)$ の束縛を合成するような $\cap'$ を定義する．

まず，$Partial(V_a, \Sigma^*)$ に次のように半順序関係 $\succeq$ を導入する．

$$f \succeq g \Leftrightarrow (\forall\alpha\in V_a)(g(\alpha)\text{が未定義}\vee f(\alpha)=g(\alpha))$$

つまり，「より多くの変数について定義されており，定義されている部分については等しい」場合に大きいと考える．この $\succeq$ に関する f と g の最小上界を $f \vee g$ と表す．ここで，$\cap'$ を次のように定義する．

$$F \cap' G = \{f \vee g \mid f \in F, g \in G, f \vee g\text{が存在する}\}$$

この $\cap'$ を使って $(r\&s)(I)$ を次のように定義する．

$$(r\&s)(I) = r(I) \cap' s(I)$$

差は，不要な束縛を除くため次のように定義する．

$$F -' G = \{f \mid f \in F, (\forall g \in G)(f\vee g\text{が存在しない})\}$$

$$(r - s)(I) = r(I) -' s(I)$$

5. 応用

5.1 HTML タグのマッチング

HTML のタグの属性の順序は決まっていない．たとえば，font タグはclass やcolor といった属性を持ちうるが，順不同である．

順不同なもののマッチングは正規表現の不得意な領域であった．選択を用いれば等価なものを表現できるが，属性が n 通りなら，$n!$ の出現順序があることになり，それらを網羅しなければならないからである．

積を用いることによってこれを解決することが可能である．たとえば，「font タグで，少なくともsize, face 属性を持つもの」を積を使って次のパターンで表すことができる．

```
<font\s*(.*size="[^"]+".*)&
(.*face="[^"]+".*)>
```

この例では，同じ属性が複数回存在することを禁止していないが，差を用いて禁止することも可能である．

さらに，後方参照を用いれば，size やface 属性に対応する値を変数に束縛することが可能である．たとえば，次のパターンでは，マッチングが成功した場合には，size の値が α に，face の値が β に束縛される．

```
<font\s*(.*size="([^"]+)%α".*)&
(.*face="([^"]+)%β".*)>
```

オプションにしたい属性がある場合には次のようにも表現できる．

```
<font\s*(.*size="([^"]+)%α".*)&
(.*(face="([^"]+)%β")?.*)>
```

ここで，r?は $(r|\epsilon)$ を意味する略記法である．この場合には，face はなくてもかまわないが，もしもあった場合は，その値が β に束縛される．

また，ある属性を禁止したい場合には差演算が利用できる．このように積と差演算は，正則集合の枠組みの中で容易に記述できるパターンを拡張することができるものであり，有効性が高いものである．

Perl などではパターンマッチングに「先読み肯定」という機能があり[3),13)]，積に似た効果を得ることができるが，正規表現とは別な機能であるために，意味を正確に定義することが困難であり，変数の扱いも整理されていない．

5.2 シャッフル演算への対応

正規表現の順不同のものへの拡張にはシャッフル演算 $\circ$ が提案されている[14)]．シャッフル演算は次のように再帰的に定義される☆．

- $u \circ \epsilon = \epsilon \circ u = u \ (u \in \Sigma^*)$
- $au\circ bv = a(u\circ bv)\cup b(au\circ v) \ (u, v \in \Sigma^*, a, b \in \Sigma)$

たとえば，$ab\circ cd$ は $abcd|acbd|acdb|cabd|cadb|adab$ と等価である．

記号列 $u = u_1 \cdots u_n$ と $v = v_1 \cdots v_m$ に対して，同じ記号が現れないとすると☆☆，$u\circ v$ と等価なパターンは，積を用いて $.\{n+m\}\&(.*u_1.*u_2\cdots u_n.*)\&(.*v_1.*v_2\cdots v_m.*)$ と表せる．つまり，「$n+m$ 文字」であり「u_1 から u_n が順番に現れる」もので「v_1 から v_m が順番に現れる」ものであるということである．

6. 結論・今後の課題

本研究では，正規表現の後方参照などの拡張への基

☆ ここでは u と v には同じ記号は現れないとする．

☆☆ シャッフル演算の主な用途であるXML ツリーへのマッチングでは，XML の関連技術の規格上，重複は起こらないので，これは大きな制限ではない．

礎として，正規表現関数を提案した．この正規表現関数により，従来は無視されてきた正規表現の左から右への処理が自然に表現され，後方参照に対しても自然な意味を与えることができた．

正規表現関数は意味の定義だけでなく，実装のモデルとしても使用できる．まず，計算量による考察では，後方参照を使わない場合には，状態数の爆発とバックトラックによる計算時間の爆発の両方を避けることができることを示した．次に，実験により，この実装が十分に実用的な速度で実行可能であることを示した．特に，NFA のようにパターンによっては合理的な時間内に計算が終わらないという問題がなく，安定しているという特長が裏づけられた．

最後に，正規表現の拡張として，積と差を提案し，正規表現関数で容易に実装が可能であることを示した．

今後は，後方参照つき正規表現についても実装と実験を行って，従来の方法と比較してどの程度の効率かを確認したい．後方参照つき正規表現の場合は，現行のアルゴリズムでは，扱う集合の大きさが $O(N^R)$ となり，時間計算量が $O(R \times N^R)$ となる．後方参照つき正規表現のマッチングは NP 完全である[4]ことが示されているため，おそらくは時間計算量自体を下げることはできないが，具体的なパターンに対する時間効率と空間効率を調べる予定である．

また，積や差は従来の実装方法の制限から利用が検討されてこなかったパターンであるので，性質や応用について研究を行っていきたい．特に，後方参照を含む正規表現の定義で使った $\cup$，$\cap'$，$-'$ については通常の集合演算と違ってド・モルガンの法則などが成り立たないので，その性質を調べていきたい．

参 考 文 献

1) Thompson, K.: Regular Expression Search Algorithm, *Comm. ACM*, Vol.11, No.6, pp.419–422 (1968).
2) Hopcroft, J.E. and Ullman, J.D.: *Introduction to Automata Theory, Languages and Computation*, Addison Wesley (1979). 野崎昭弘，高橋正子，町田　元（訳）：オートマトン言語理論計算論 I/II，サイエンス社 (1984).
3) Friedl, J.E.F.: *Mastering Regular Expressions*, O'Reilly & Associates (1997). 歌代昭正（訳）：詳説正規表現，オライリー・ジャパン (1999).
4) Aho, A.V.: Algorithms for Finding Patterns in Strings, *Handbook of Theoretical Computer Science*, Vol.A, No.6, pp.256–300 (1990).
5) Hutton, G.: Parsing Using Combinators, *Proc. 1990 Glasgow Workshop on Functional Programming* (*GWFP 1990*) (1990).
6) Harper, R.: Proof-directed debugging, *Journal of Functional Programming* (*Functional Pearls*), Vol.9, No.4, pp.463–469 (1999).
7) Hosoya, H. and Pierce, B.: Regular Expression Pattern Matching for XML, *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2001).
8) IEEE: Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 2 (Shell and Utilities), Section 2.8 (Regular Expression Notation) (IEEE 1003.2) (1992).
9) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers Principiles, Techniques, and Tools*, Addison Wesley (1986). 原田賢一（訳）：コンパイラ I 原理・技法・ツール，サイエンス社 (1990).
10) Myers, G.: A Four Russians Algorithm for Regular Expression Patter Matching, *J. ACM*, Vol.39, No.4, pp.430–448 (1992).
11) 津田伸秀：ViVi のドキュメント (1998). http://hp.vector.co.jp/authors/VA007799/vivi.htm
12) Clarke, C.L.A. and Cormack, G.V.: On the Use of Regular Expressions for Searching Text, *ACM Trans. Prog. Lang. Syst.*, Vol.19, No.3, pp.413–426 (1997).
13) Wall, L., Christiansen, T. and Schwartz, R.L.: *Programming Perl*, Second Edition, O'Reilly (1986). 近藤嘉雪（訳）：プログラミング Perl 改訂版，オライリー・ジャパン (1997).
14) Jedrzejowicz, J. and Szepietowski, A.: Shuffle languages are in P, *Theoretical Computer Science*, Vol.250, No.1-2, pp.31–53 (2001).

(平成 14 年 9 月 13 日受付)
(平成 15 年 5 月 6 日採録)

山本　篤（学生会員）
1976 年生．2000 年東京大学広域科学科広域システム分科卒業．2002 年東京大学大学院総合文化研究科修士課程修了．現在，同研究科博士課程在学．テキスト検索，ウェブを用いた教育システムの研究等を行っている．

山口　和紀（正会員）

1956 年生．1979 年東京大学理学部数学科卒業．1981 年東京大学理学部助手．1985 年理学博士（東京大学）．1989 年筑波大学電子情報工学系講師．1992 年東京大学教養学部助教授．1999 年東京大学情報基盤センター教授．コンピュータのためのモデリング全般に興味を持つ．