正規表現における非包含オペレータの提案
Absent Operator for Regular Expression

田中 哲
Tanaka Akira
akr@fsij.org

産業技術総合研究所 情報技術研究部門
Information Technology Research Institute,
National Institute of Advanced Industrial Science and Technology (AIST)

概要

正規表現の拡張として非包含オペレータを提案する。正規表現は昨今スクリプト言語等で活用されているが、もともと形式言語理論により定義されたものである。形式言語理論においては、正規言語における補集合の閉包性が証明されており、ある正規表現から導出できない文字列すべてだけを導出できる正規表現が存在することが知られている。補集合は C 言語のコメントや CR LF 終端の行などを表現するのに役に立つが、現実にそのような正規表現を構成するのは繁雑であり難しい。そこで正規表現エンジンで補集合オペレータを直接サポートすることが考えられるが、これは Perl, Ruby 等の既存の正規表現エンジンで採用されているバックトラッキングを用いるアルゴリズムでは効率よく実装できない。また、既存の正規表現の拡張には最短一致の繰り返し、バックトラックの抑制、否定先読みなど容易な記述を可能とするものもあるが、それらは理論による適切な意味付けができない。そこで本論文では、既存のバックトラック型の正規表現エンジン上で容易に効率よく実装でき、かつ、形式言語理論による適切な意味付けが可能な非包含オペレータを提案する。

This paper proposes a new operator, absent operator, for regular expressions. Recently, regular expressions are widely used with scripting languages. There are several usages which is too difficult to write down in a regular expression: a comment in C language, CR LF terminated lines, etc. They are possible to represented by a regular expression because the formal language theory proves complement of a regular language is also regular. The absent operator eases them. The operator is easy to implement on a backtracking regular expression engine and it has proper semantics on the theory unlike lazy match, atomic grouping, negative lookahead.

1 はじめに

昨今スクリプト言語などで正規表現 [3] が活用されている。正規表現はもともと形式言語理論 [4] により定義されたものであり、形式言語理論における知見がスクリプト言語でも利用できることが期待される。

しかし、現実には必ずしも利用できるとは限らない。たとえば、理論ではある正規言語の補集合に関する閉包性が証明されている。これは、ある正規表現から導出可能な文字列集合に対し、ちょうどそれ以外の文字列集合を導出可能な正規表現が存在することを示している。

補集合は例えば C 言語のコメントや、CR LF で終了する行を表現するのに使用できる。C 言語のコメントは “/*” で始まり “*/” で終わるが、その内部には “*/” は出現できない。また、CR LF で終了する行はその終端以外に CR LF は出現できない。他にも Python[7] の """..."""" という 3 つのクォートで括る文字列、HTTP[2] におけるヘッ

ダの CR LF CR LF による終端、SMTP[1] におけるメールの CR LF . CR LF による終端など、その終端が長さ 2 以上の文字列であって、その内容に終端の部分文字列が出現し得るものは珍しくない。

これらはいずれも終端文字列が含まれない文字列の集合 A として表せる。A の補集合に対応する正規表現は終端文字列が含まれている文字列であるから「(任意の文字列)(終端文字列)(任意の文字列)」として記述できる。したがって補集合の閉包性から A に対応する正規表現も存在することがわかる。しかし、A に対応する正規表現を構成することは容易な作業ではない。

Perl[8] 等では最短一致、バックトラックの抑制、否定先読みなどの機能を使用すれば C 言語のコメントなどを容易に記述できる。しかし、これらの機能は形式言語理論から逸脱しており、理論的な取扱いに問題がある。

本論文では、容易に記述でき、かつ、理論的な取扱いも問題のない正規表現の拡張として非包含オペレータを提案する。

なお、正規表現エンジンの簡単な実装を示すが、その記述言語には Ruby[9] を用いる。

2 正規表現

正規表現 r の構文を以下に示す。

$$
\begin{array}{lr}
r = \varepsilon & \text{空文字列} \\
\quad | \ c & \text{文字} \\
\quad | \ rr & \text{連接} \\
\quad | \ r|r & \text{選択 (集合和)} \\
\quad | \ r* & \text{0 回以上の繰り返し (Kleene 閉包)}
\end{array}
$$

ここで、c はアルファベット Σ の要素である。

正規表現は文字列の集合の記法であり、正規表現 r に対し、対応する文字列の集合を $L(r)$ と記述し、r に対応する言語と呼ぶ。また、正規表現に対応する言語を正則集合という。

なお、本論文では、正規表現の記法として Unix[*1] 的な記法を使用する。この記法では伝統的な形式言語理論の記法に対し、選択 (集合和) の記法が異なる。形式言語理論において選択に用いられる + 記号は Unix 的には 1 回以上の繰り返しに用いる。

表 1　正規表現の記法

形式言語理論	Unix	意味
$r+r$	$r\|r$	選択
rr	rr	連接
$r*$	$r*$	0 回以上の繰り返し
なし	$r+$	1 回以上の繰り返し

また、任意の文字を表す正規表現として . (ドット) を使用する。Perl, Ruby 等での . は改行を含まないこともあるが、本論文の . はそのような例外のない全ての文字を表現するものとする。

正規表現 r とある文字列 s があって、$s \in L(r)$ のとき、文字列 s は正規表現 r から導出可能という。また、$s = uvw$ として $v \in L(r)$ のとき、文字列 s は正規表現 r を含むという。

3 正規表現エンジンの実装

昨今のスクリプト言語[*2] や POSIX[*3] は正規表現エンジンを提供している。正規表現エンジンは正規表現と文字列があたえられたとき、その文字列がその正規表現から導出可能かどうか、あるいは導出可能な文字列を含んでいるかどうかを調べることができる。

これを調べる方法はいくつか考えられる。まず、正規表現と文字列の対応をバックトラックを用いて探索する方法がある。また、形式言語理論により正規表現と有限オートマトンの同値性が示されていることから、正規表現を DFA (決定性有限オートマトン) に変換し、その DFA を文字列に適用する方法もある。

一般に、正規表現エンジンではバックトラックを用いることが多い。この理由としてはキャプチャを DFA で提供することは難しいことがあげられ

[*1] egrep, awk, Perl, Ruby 等

[*2] Perl, Python, Ruby 等

[*3] regex.h

表 2　DFA とバックトラックの比較

	バックトラック	DFA
キャプチャ	容易	困難
正規表現サイズに対する最悪時間計算量	線形	指数関数的
正規表現サイズに対する最悪空間計算量	線形	指数関数的
文字列長に対する最悪時間計算量	指数関数的	線形
文字列長に対する最悪空間計算量	線形	定数

る。キャプチャとは文字列が正規表現を含んでいるとき、正規表現中の括弧で括った部分正規表現に対応した部分文字列を取り出す機能である。例えば Perl では `"foobar" =~ /f(.*)a/` とすると特殊変数 `$1` に `"oob"` が得られる。DFA の状態遷移の経路はもともとの正規表現と一意に対応しないため、DFA における状態遷移からキャプチャに必要な情報を取り出すことは簡単ではない。

また、DFA の利点は文字列長に対する計算量が抑えられることであるが、逆に正規表現自体のサイズに対する計算量は悪化する。とくに、空間計算量が指数関数的になりうることは他のプロセスを swap out させてしまうなど、計算機全体に影響する大きな問題を発生させることがある。

以上をまとめたものを表 2 に示す。これらの理由により、正規表現エンジンはバックトラックを用いることが多い。以下の節では、バックトラックを用いた簡単な正規表現エンジンを示す。

3.1　正規表現の抽象構文木

Ruby による正規表現エンジンを示すにあたり、正規表現の抽象構文木を表 3 に示す。抽象構文木の各ノードは配列で表現し、配列の最初の要素のシンボルによって種類を表現する。それ以降の要素で各種類に応じた情報を表現する。

3.2　基本的な正規表現エンジン

以下に正規表現エンジンの実装を示す。

`try` が正規表現エンジンの中心となる手続きである。これは文字列中の特定の位置で始まる部分文字列であたえられた正規表現から導出可能なものをすべて検出する。そのために正規表現、文字列、部分文字列の開始位置を受け取り、部分文字列が検出されるたびにあたえられたブロックを呼び出して終端位置を渡す。

表 3　Ruby による正規表現の抽象構文木

正規表現	Ruby
ε	`[:empseq]`
c	`[:lit, "`c`"]`
$r_1 r_2$	`[:seq,` r_1`,` r_2`]`
$r_1 \vert r_2$	`[:alt,` r_1`,` r_2`]`
$r*$	`[:rep,` r`]`

ここで、正規表現は前節で述べた抽象構文木であり、文字列は単独の文字からなる文字列を要素とする配列とする。

`try` により、ある文字列のある位置から始まる部分文字列で、ある正規表現から導出可能なものをすべて得ることができる。

たとえば、以下のように実行すると (a|b)* という正規表現を “aba” という文字列の先頭から適用し、正規表現から導出可能な部分文字列 “aba”, “ab”, “a”, “” の各終端となる $3, 2, 1, 0$ を順に表示する。

```
try([:rep, [:alt, [:lit, "a"],
                  [:lit, "b"]]],
    ["a", "b", "a"],
    0) {|pos|
  p pos
}
```

`try` の中身は個々の構文毎に分岐を行い、具体的

な処理は個々の手続きを呼び出す。たとえば、空文字列に対しては try_empseq を呼び出す。

```
# re: 正規表現の抽象構文木
# str: 文字の配列
# pos: 開始位置
# block: 検出時に起動するブロック
#        （コールバック）
def try(re, str, pos, &block)
  case re[0]
  when :empseq
    try_empseq(str, pos, &block)
  when :lit
    _, ch = re
    try_lit(ch, str, pos, &block)
  when :cat
    _, r1, r2 = re
    try_cat(r1, r2, str, pos, &block)
  when :alt
    _, r1, r2 = re
    try_alt(r1, r2, str, pos, &block)
  when :rep
    _, r = re
    try_rep(r, str, pos, &block)
  else
    raise "unexpected AST: #{re.inspect}"
  end
end

def try_empseq(str, pos)
  yield pos
end

def try_lit(ch, str, pos)
  if pos < str.length && str[pos] == ch
    yield pos + 1
  end
end

def try_cat(r1, r2, str, pos, &block)
  try(r1, str, pos) {|pos2|
    try(r2, str, pos2, &block)
  }
end

def try_alt(r1, r2, str, pos, &block)
  try(r1, str, pos, &block)
  try(r2, str, pos, &block)
end

def try_rep(r, str, pos, &block)
  try(r, str, pos) {|pos2|
    if pos < pos2
      try_rep(r, str, pos2, &block)
    end
  }
  yield pos
end
```

try を文字列内のすべての位置から呼び出すことにより、文字列が正規表現を含むかどうかを調べることができる。これを行う has_match 手続きは以下のように実装できる。

```
def has_match(r, str)
  0.upto(str.length) {|i|
    try(r, str, i) {|pos|
      return [i, pos]
    }
  }
  nil
end
```

4　非包含オペレータの提案

正規表現エンジンはバックトラックを用いることが多いが、このことは形式言語理論の知見の適用を難しくする。

たとえば 1 節で述べたように、ある正規表現の補集合を扱いたいことがある。その場合 DFA ならば受理状態とそうでない状態を入れ換えるだけで実現

できる。しかし、詳しくは 7.3 節で述べるが、バックトラック型正規表現エンジンでは補集合を簡単には実現できない。

そこで、非包含オペレータを提案する。

5 非包含オペレータ $!r$

非包含オペレータはある正規表現 r に対し、r から導出される文字列を含まない文字列を表す記法であり、$!r$ と記述する。$!r$ が示す文字列の集合 $L(!r)$ は以下となる。

$$L(!r) = \Sigma^* - L(.*r.*)$$

ここで $.*$ は任意の文字 (ドット) の 0 回以上の繰り返しであり、任意の文字列を表す。Σ^* は任意の文字列であり、これを全体集合として $L(!r)$ は $L(.*r.*)$ の補集合となっている。

たとえばアルファベットを $\Sigma = \{a, b\}$ として $L(!a)$ は $\{\varepsilon, b, bb, bbb, \ldots\}$ である。この集合は a を含まず、さらに a を含む $aa, ab, ba, aaa, \ldots$ なども含まない。結果として、b のみからなる長さ 0 以上の文字列の集合となる。

また、$L(!(ab))$ は $\{b^m a^n | 0 \leq m \wedge 0 \leq n\}$ となる。a の後に b が出現すると ab を含んでしまうため、b の後に a が表れる文字列の集合となる。

$L(!r)$ の性質として、$L(!r)$ に含まれない文字列 s があれば、それをプリフィクスとする文字列はすべて $L(!r)$ に含まれない。つまり、以下が成り立つ。

$$s \notin L(!r) \rightarrow \forall t \in \Sigma^* \; st \notin L(!r)$$

これは、s は r から導出できる文字列 $v \in L(r)$ を部分文字列として持つため、st も v を部分文字列として持ち、その結果 st は $L(!r)$ には含まれない。この単調性により、実装において r から導出できる文字列を最初に発見した以降の探索が不要となる。

6 非包含オペレータの実装

$!r$ の抽象構文木を `[:absent, r]` として以下に非包含オペレータの実装を示す。これに付随して、`try` も修正し、抽象構文木 `[:absent, r]` を認識して r を取り出し `try_absent` を呼ぶように変更する。

```
def try_absent(r, str, pos)
  limit = str.length
  pos2 = pos
  while pos2 <= limit
    try(r, str, pos2) {|pos3|
      limit = pos3-1 if pos3-1 < limit
    }
    pos2 += 1
  end
  limit.downto(pos) {|pos4|
    yield pos4
  }
end
```

`try_absent` はまず `r` から導出可能な文字列を `str` 中の `pos` 以降で探索する。そのとき、複数の箇所が見つかる可能性があるが、各箇所の右端の中で最も左にある位置を求める。ここで最左の位置が求まればいいため、すでに見つかった右端よりも右の位置からは探索を行う必要はない。

その最左位置のさらにひとつ左 (`limit`) までの文字列には r から導出可能な文字列は含まれていない。また、前節で述べた単調性から、`limit` よりも右のいずれかの位置までの文字列には r から導出可能な文字列が常に含まれる。つまり $!r$ に含まれるのは `pos` から `limit` までの範囲だけであり、その範囲の各位置についてあたえられたブロックを呼び出して呼出元に伝える。

なお、正規表現エンジンの戦略は基本的に最長一致であるため、呼出元に伝える順序は降順とする。必要であれば、別に昇順を提供するオペレータを提供しても良い。

7 非包含オペレータの利点

以下に非包含オペレータの利点を述べる。

7.1 記述の容易さ

非包含オペレータの利点は、まず、記述が困難だったものを容易に記述可能とすることである。

たとえば、C 言語のコメントは非包含オペレータを使用しない場合、以下のように記述しなければならない。(/ と * がメタキャラクタ[*4]であるというそれらの文字自身に起因する困難さを排除するため、2 行目に / と * を a と b に置換したものを示す。)

```
\/\*[^\*]*\*+(([^\*\/][^\*]*)\*+)*\/
ab[^b]*b+(([^ba][^b]*)b+)*a
```

非包含オペレータを用いれば以下のように記述できる。

```
\/\*!(\*\/)\*\/
ab!(ba)ba
```

非包含オペレータを用いないものは、“*/” を各文字に分解して複雑に組み合わせる必要がある。しかし、非包含オペレータを用いた場合、「“*/” が含まれない」という意図を直接的に表現できており、容易かつ自然な記述となっている。この容易さの違いは、含まれていないことを示したい文字列が 2 文字よりも長くなると、さらに顕著となる。

7.2 形式言語理論との対応

C 言語のコメントを表現する正規表現は前節であげたように、非包含オペレータを用いなくても記述できる。しかし、これを実際に構成するのは困難であるため、他の記述で間に合わされることがある。しかし、それらの記述には形式言語理論との対応において不適切な点がある。

それらに対し、非包含オペレータは理論との対応に問題がない。非包含オペレータで記述可能な集合は正則集合である。これは $L(!r) = \Sigma^* - L(.*r.*)$ であり、$L(.*r.*)$ が正則集合であることと正則集合の補集合が正則集合であることから導かれる。

以下では正規表現のいくつかの機能を使った簡易的な記述の欠点を述べる。

7.2.1 最短一致の繰り返し

C 言語のコメントに使われる正規表現としてよくあげられるものに最短一致の繰り返しを用いたものがある。それは以下の正規表現である。

```
\/\*.*?\*\/
ab.*?ba
```

.*? は .* と同じく任意の文字の 0 回以上の繰り返しであるが、繰り返しの回数の少ない可能性を先に試す最短一致の戦略を用いる。つまり、導出可能な文字列の集合は同じで、$L(r*?) = L(r*)$ である。

ここで最短一致について述べる。最短一致の繰り返しの抽象構文木を [:rep_lazy, r] とすると、try に以下の手続きを呼ぶ分岐を追加することにより実装できる。

```
def try_rep_lazy(r, str, pos, &block)
  yield pos
  try(r, str, pos) {|pos2|
    if pos < pos2
      try_rep_lazy(r, str, pos2, &block)
    end
  }
end
```

try_rep_lazy は try_rep と基本的に同じであるが、ブロックを呼ぶ yield の位置が異なる。try_rep は帰りがけ順でブロックを呼ぶが、try_rep_lazy は行きがけ順でブロックを呼ぶ。これにより、try_rep が呼出元に繰り返しが多い場合から通知するのに比べ、try_rep_lazy は繰り返しが少ない場合から通知する。この違いが最長一致・最短一致の戦略の違いである。異なるのは通知の順序のみであり、どちらかで通知される終端位置はもう一方でも通知される。

この最短一致の繰り返しを用いて C 言語のコメントを記述することが行われるが、これは通知の順序という戦略に依存している。つまり、最初に通知されるものを用いるのであれば正しく C 言語のコメントが得られるが、そうでなければ正しいとは

[*4] * は繰り返しであり、/ は一般に正規表現の区切りに用いられるため、それらの文字自身を表すためには \ でエスケープしなければならない。

限らない。つまり、最初に通知された後にバックトラックが起きて次の通知に移れば、C 言語のコメントとして不適切なものが得られる。

たとえば、C 言語のコメントの直後で行が終わっている (改行 \n がある) という正規表現を以下のように記述したとすると、これは不適切な箇所を検出してしまう可能性がある。

```
\/\*.*?\*\/\n
ab.*?ba\n
```

たとえば、1/*2*/3/*4*/\n というコメントが複数存在する行に対して、/*2*/3/*4*/\n を検出してしまう。これは、コメントの終端までたどりついた後、改行と一致せずにバックトラックが起こるからである。

それに対し、以下のどちらかを用いれば正しく /*4*/\n を検出する。

```
\/\*[^\*]*\*+(([^\*\/][^\*]*)\*+)*\/\n
\/\*!(\*\/)\*\/\n

ab[^b]*b+(([^ba][^b]*)b+)*a\n
ab!(ba)ba\n
```

この問題は最短一致の繰り返しが結局は最長一致の繰り返しと同じ文字列を導出可能という点に起因する。そのために、他の正規表現との組合せ (例えば \n との連接) において問題が生じる。

つまり、最短一致の繰り返しは記述が容易であるが、正規表現の組み合わせの部品として用いるには適切でない。それに対し非包含オペレータは記述が容易であり、かつ、正規表現の組合せの部品として問題なく用いることができる。

7.2.2 バックトラックの抑制

Perl, Ruby には実験的な機能としてバックトラックの抑制を行うオペレータ (?>r) がある。これは r に対し正規表現エンジンが検出する終端位置のうち、最初のひとつだけを使い、それ以外を無視するというものである。抽象構文木を [:nobacktrack, r] とすると、以下のように実装できる。

```
def try_nobacktrack(r, str, pos)
  try(r, str, pos) {|pos2|
    yield pos2
    return
  }
end
```

これと最短一致の繰り返しを組み合わせると C 言語のコメントは以下のように記述できる。

```
\/\*(?>.*?\*\/)
ab(?>.*?ba)
```

この正規表現は改行と連接して以下のようにしても問題なく動作する。

```
\/\*(?>.*?\*\/)\n
ab(?>.*?ba)\n
```

しかし、バックトラックの抑制はバックトラックの戦略に強く依存するため、理論的な扱いに問題が生じる。

たとえば、正規表現が 2 つあったとき、それらの等価性を判定することができる。それは正規表現それぞれを DFA に変換・最小化して合同かどうかを調べることで実現できるが、バックトラックの抑制を導入した場合、どのようにすれば DFA に変換できるか明らかでない。

また、正規表現から導出される文字列を左右逆転することを考える。これは基本的な正規表現であれば、連接の左右を逆転させれば良い。しかし、バックトラックの抑制があると、可能かどうか明らかでない。単に連接だけを逆にすると (?>\/*.*?)*\/ となるが、これは .*? が最初に空文字列を検出した段階でそれ以上のバックトラックが抑制されるため、コメントの中身が空である /**/ しか検出しない。(?>\/*.*?*\/) とすれば左右逆転しても同じになるが、そうすると C 言語のコメントは後ろからはコメントの開始を一意に決定できないという曖昧性を表現できない。たとえば、/* A /* B */ という C 言語によるソースコードがあったとすると、左から調べていけば全体がコメントであること

が一意に判明するが、右からの場合 /* B */ まで調べた段階ではコメントの開始がそこかどうか判断してはならない。しかし、(?>\/*.*?*\/) という正規表現を使って右から調べた場合、そこがコメントの開始であると判断してしまう。

7.2.3 否定先読み

また、否定先読み (negative lookahead assertion) を用いる方法がある。否定先読みを使用して C 言語のコメントを表す正規表現は以下のようになる。

```
\/\*((?!\*\/).)*\*\/
ab((?!ba).)*ba
```

ここで (?!r) が否定先読みであり、これは直後に r から導出可能な文字列がない空文字列を検出する。コメントの中身の各文字は終端文字列の最初の文字にならない文字であるから、終端文字列を否定する先読みと任意の文字を連接すれば中身の文字を表現できる。したがってコメントの開始文字列、中身の文字の 0 回以上の繰り返し、終端文字列を連接することにより C 言語のコメントを表現する正規表現を実現できる。

否定先読みは抽象構文木を [:nlookahead, r] とすると以下のように実装できる。

```
def try_nlookahead(r, str, pos)
  matched = false
  try(r, str, pos) {|pos2|
    matched = true
    break
  }
  yield pos if !matched
end
```

否定先読みは明らかに理論から逸脱している。否定先読みが生成する言語 $L(\texttt{(?!}r\texttt{)})$ を定義することは出来ない。あえて考えれば空集合 ϕ か空文字列だけからなる集合 $\{\varepsilon\}$ のどちらかになるが、そのどちらになるかは直後の文字列に依存する。このため、理論的な扱いは困難である。

また、終端文字列の長さが可変な場合、それを含まない文字列を示す正規表現を記述するのは困難である。

7.3 補集合の非効率性

C 言語のコメントのような用途に対して、理論と直接的に対応する補集合をそのまま提供することが考えられる。しかし、これをバックトラッキング型正規表現エンジンに実装するのは非効率的である。

抽象構文木を [:negation, r] として以下に補集合オペレータの実装を示す。

```
def try_negation(r, str, pos)
  h = {}
  try(r, str, pos) {|pos2|
    h[pos2] = true
  }
  str.length.downto(pos) {|pos2|
    yield pos2 if !h[pos2]
  }
end
```

pos で始まる r から導出できない部分文字列を知るためにはまず r から導出できる部分文字列をすべて得なければならない。上記の実装では try を呼び出してそれを行っており、ハッシュ h を使用して全ての終端を記録している。そして、検出された終端以外の箇所を呼出元に通知している。

ここで、C 言語のコメントのような、ある終端文字列が含まれない文字列を探索する場合、r に「(任意の文字列)(終端文字列)(任意の文字列)」という正規表現が与えられる。そのため、str 中の pos 以降にあるすべての終端文字列を検索し、さらにそれ以降の任意の文字列を探索する必要がある。しかし、終端文字列は最左のひとつを検出できればそれでよく、すでに見つかっているものより右の検索は不要である。また、見つかった終端文字列より右に任意の文字列があることを確認することも不要である。そして、その不要な検索の結果をすべてハッシュに記録しておくという空間的な無駄も生じる。

これに対し、非包含オペレータで記録するのは終端文字列の右端のなかで最も左の位置という整数ひとつであり、それを求める最小限の探索しか行わない。そのため、空間的・時間的に効率が良い。

8 関連研究

8.1 Ragel

Ragel[6] は正規表現から状態機械へのコンパイラであり、本論文と関連するオペレータとして以下のものを持っている。

- negation オペレータ $!r$
  $!r$ から導出可能な文字列は r からは導出できない文字列の集合である。つまり、これは 7.3 節で述べた補集合オペレータである。
- difference オペレータ $r_1 - r_2$
  $r_1 - r_2$ は r_1 から導出可能だが r_2 からは導出できない文字列の集合である。ここで r_1 をすべての文字列を導出可能な正規表現とすれば補集合オペレータになる。
- strong difference オペレータ $r_1 -- r_2$
  $r_1 -- r_2$ は r_1 から導出可能だが r_2 を含まない文字列の集合である。ここで、r_1 をすべての文字列を導出可能な正規表現とすれば本論文で提案する非包含オペレータになる。

negation オペレータおよび difference オペレータはバックトラッキング型正規表現エンジンで効率よく実装することは困難である。strong difference オペレータの実現においては、r_1 から導出可能であることを確認する必要があり、バックトラッキング型正規表現エンジンで実現する場合、まず非包含オペレータと同様の機構によって r_2 を含まない範囲を求め、その範囲内において r_1 から導出可能な文字列を探索することになる。

8.2 Perl

Perl には最短一致の繰り返し、バックトラックの抑制、否定先読みがある。これにより、非包含オペレータに似た効果を得ることができる。しかし、7.2 節で詳しく述べたようにこれらは形式言語理論からすると適切に扱えず、正規表現の組合せなどに問題が生じる。

8.3 Grail

Grail[5] は C++ 用のライブラリで、正規表現や有限状態機械を操作するものである。本論文に関連する操作としては、補集合を求める操作がある。この操作は有限状態機械を対象にして受理状態を反転させるものである。このため正規表現に対して適用するにはまず有限状態機械に変換しなければならない。つまりバックトラッキング型の正規表現エンジンには適用できない。

9 まとめ

ある正規表現から導出可能な文字列を含まない文字列を示す非包含オペレータを提案した。非包含オペレータは理論的な意味がはっきりしており、かつ、C 言語のコメントなどの実際の用途を持つ。そして、広く使われているバックトラッキング型正規表現エンジンの拡張として容易に導入できる。非包含オペレータがない場合、同様のことを実現するには複雑な正規表現を構成するか、理論的な扱いに問題の生じる機能を用いなければならない。それに対し非包含オペレータは、記述が容易で理論的な扱いにも適している。

参考文献

[1] Simple mail transfer protocol. RFC 2821, Internet Engineering Task Force, April 2001.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, June 1999.

[3] Jeffrey E. F. Friedl. *Mastering Regular Expressions, 2nd ed.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[4] J. ウルマン J. ホップクロフト. オートマトン言語理論 計算論 I. サイエンス社, 1984.

[5] Darrell Raymond. Grail: a C++ library for finite-state machines and regular expressions. In *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, p. 60. IBM Press, 1994.

[6] Adrian Thurston. Ragel state machine

compiler. http://www.cs.queensu.ca/~thurston/ragel/ (2007–11–30).

[7] Guido van Rossum. Python programming language. http://www.python.org/ (2007–11–30).

[8] Larry Wall. *Programming Perl, 3rd edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[9] まつもとゆきひろ. オブジェクト指向スクリプト言語 ruby. http://www.ruby-lang.org/ (2007–11–30).