Regular expressions à la carte

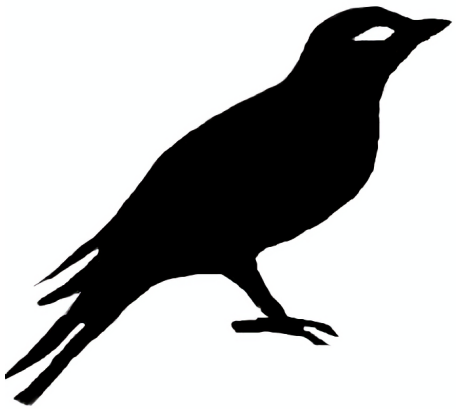
kbkz.tech #9

吉村 優

https://twitter.com/_yyu_
http://qiita.com/yyu
https://github.com/y-yu

March 20, 2016

自己紹介



次のような記事を書きました

VM 型の正規表現エンジンを実装する

http://qiita.com/yyu/items/84b1a00459408d1a7321

正規表現から LLVM へのコンパイラを実装する

http://qiita.com/yyu/items/a0ef2d2204c137707f3f

正規表現の JIT コンパイラを実装する

http://qiita.com/yyu/items/3c4deb39d6b0a7955572

正規表現の微分でサブマッチング

http://qiita.com/yyu/items/1638fd59bedce27ca3a4

アウトライン

正規表現とは？

> **正規表現**とは、文字列の集合を一つの文字列で表現する方法の一つである。
>
> Wikipedia - 正規表現

正規表現の定義

文字の集合 Σ 上の正規表現は次のように定義される

- 空集合を表す ϕ は正規表現
- 空文字を表す ϵ は正規表現
- $a \in \Sigma$ は正規表現
- X と Y が正規表現であるとき次のものも正規表現
  - $X \mid Y$（選択）
  - XY（結合）
  - X^*（0 回以上の繰り返し）

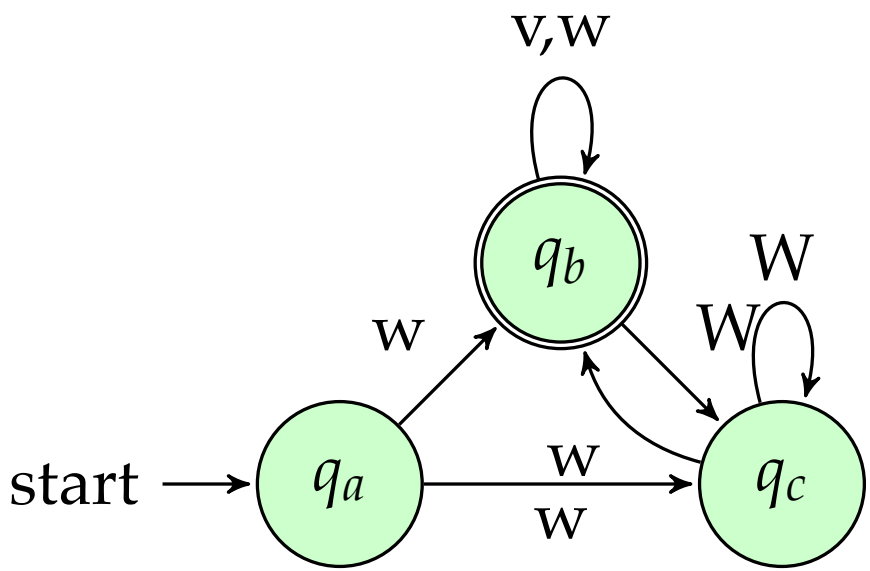
マッチング

主に次のような方法がある

- 有限オートマトンを用いた方法
- Virtual Machine を用いた方法
- 正規表現の微分を用いた方法

非決定性有限オートマトン（NFA）

> “**非決定性有限オートマトン**は、有限オートマトンの一種であり、ある状態と入力があったとき、次の遷移先が一意に決定しないことがあるものである。”
>
> Wikipedia - 非決定性有限オートマトン

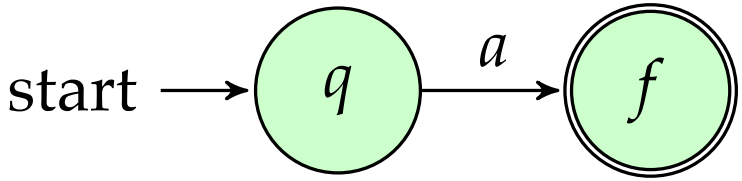


NFA を用いたマッチング

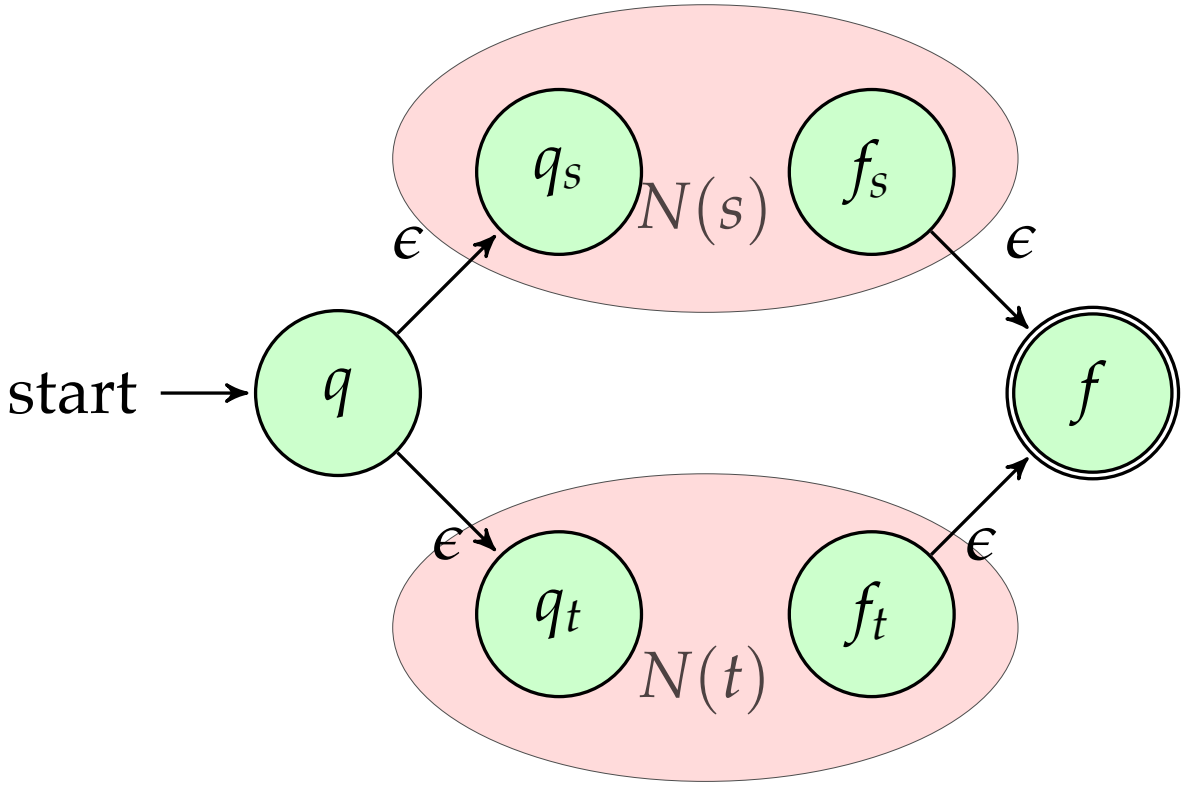
- 全ての正規表現には対応する非決定性有限オートマトンが存在する

正規表現と NFA

文字 $N(a)$

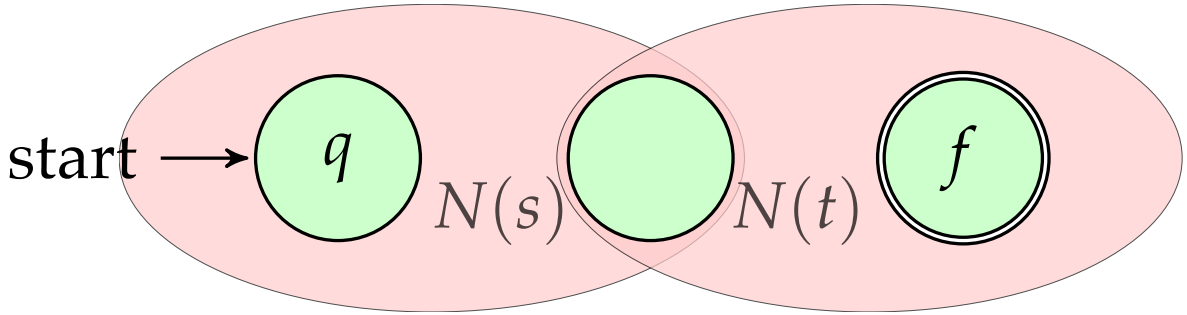


選択 $N(s \mid t)$

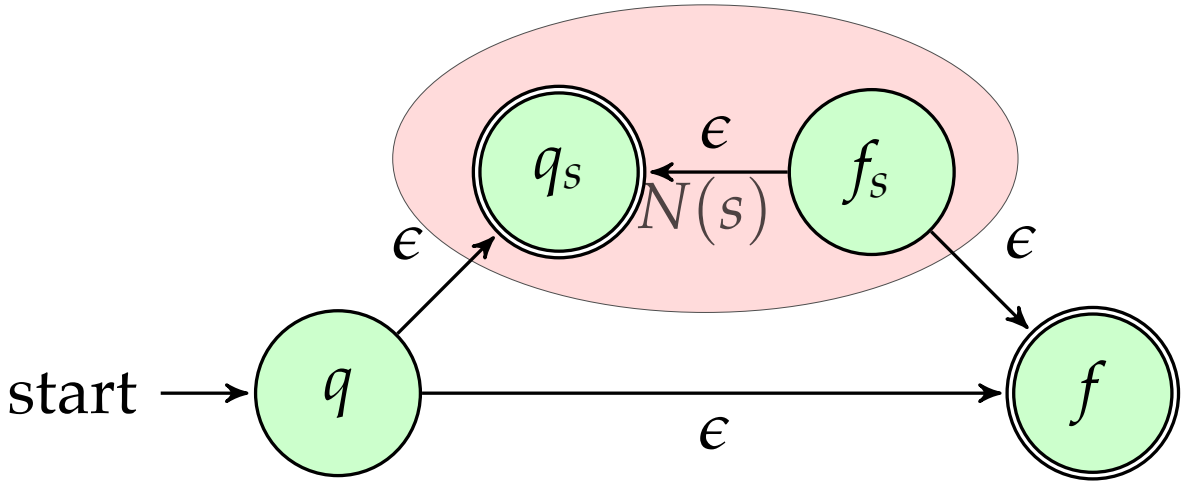


正規表現と NFA

結合 $N(st)$



繰り返し $N(s^*)$

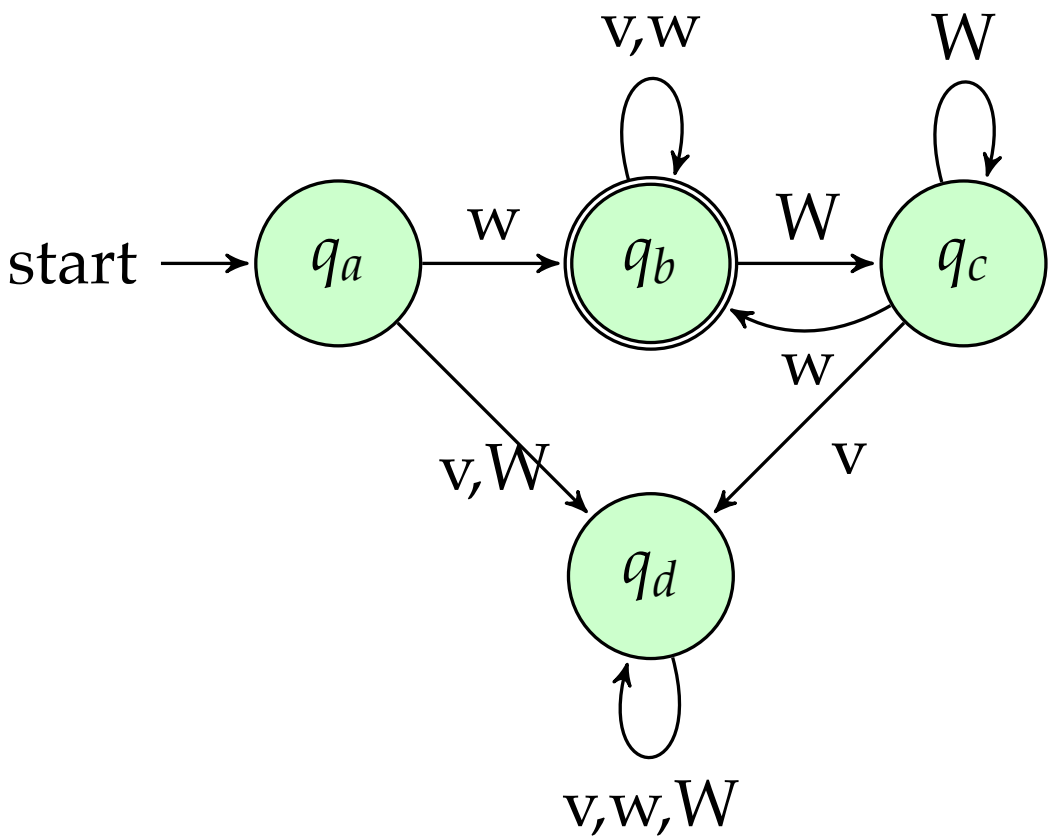


NFA を用いたマッチング

- 全ての正規表現には対応する非決定性有限オートマトンが存在する
- 遷移先が一意に決まらないので、バックトラックをする必要がある

決定性有限オートマトン（DFA）

> **決定性有限オートマトン**は、状態と入力によって次に遷移すべき状態が一意に定まる有限オートマトンである。
>
> Wikipedia - 決定性有限オートマトン



DFA を用いたマッチング

- 非決定性有限オートマトンから機械的に変換できる（サブセット構成など）
- 入力によって状態が一意に決まるので、バックトラックをする必要がない
- 非決定有限オートマトンから変換すると、最悪の場合、**状態数が指数関数的に増加する**
- そのため、必要になった状態だけ変換するというテクニックがある

Virtual Machine（VM）を使ったマッチング

次のような VM を用いて正規表現のマッチングが可能

- PC と SP という 2 つのレジスタがある
- 次の命令がある

  `char c`
  : SP の先頭の文字と c を比較

  `match`
  : マッチに成功（スレッドを終了）

  `jmp x`
  : アドレス x へジャンプ（PC を x にする）

  `split x, y`
  : スレッドを二つに分割する。片方は PC を x にし、もう片方は PC を y にする

Virtual Machine（VM）を用いたマッチング

正規表現を次のように変換する

文字（c）

```
    char  c
```

選択（$e_1 \mid e_2$）

```
        split L1 L2
L1 :    e1 の命令列
        jmp L3
L2 :    e2 の命令列
L3 :
```

連結（$e_1 e_2$）

```
    e1 の命令列
    e2 の命令列
```

繰り返し（$e*$）

```
L1 :    split L1 L3
L2 :    e2 の命令列
        jmp L1
L3 :
```

Virtual Machine（VM）を用いたマッチング

正規表現/aa*bb*/を VM のバイトコードへ変換する

```
0       char a
1       split 2, 4
2       char a
3       jmp 1
4       char b
5       split 6, 8
6       char b
7       jmp 5
8       match
```

- VM の命令を LLVM や JVM に変換すれば、より高速になる

正規表現の微分を用いたマッチング

正規表現の微分

$c : wc$ という文字列があるとする*。ある正規表現 r が文字列 $c : wc$ にマッチするならば、r を文字 c で微分した正規表現 r_c は文字列 wc にマッチする。

1. マッチング対象の文字列から 1 文字ずつ取り出し、正規表現を微分していく
2. 文字列が空になった時に、微分された正規表現が空文字を受理するならばマッチングに成功

*この文字列 $c : wc$ は文字 c が文字列の先頭の 1 文字で、cw は文字列の先頭以外の残りを表す

正規表現の限界

```
/^1?$|^(11+)\1+$/
```

正規表現の限界

1 が非素数個ある文字列の正規表現

```
/^1?$|^(11+)\1+$/
```

例

次のような文字列がマッチする

- $\underbrace{1}_{1?}$
- $\underbrace{11}_{11+?}\underbrace{11}_{\backslash 1}$
- $\underbrace{111}_{11+?}\underbrace{111}_{\backslash 1}\underbrace{111}_{\backslash 1}$

正規表現と非正規表現

- **ポンピング補題**などを使うと、正規表現で表せる集合かどうか証明できる
- たとえば、次のものは**正規**である
  - ある正規表現の補集合を表す正規表現
  - 先読み
- たとえば、次のものは**非正規**である
  - 後方参照
  - 再帰
- 非素数にマッチする正規表現は後方参照を用いていたので、非正規表現である
- 非正規になると、正規表現が持つよい性質が失われる

正規表現 vs C++[†]

[†]この内容はほとんどが新屋さんの成果です

w を表示するプログラム

w を表示するプログラム

“
```
main(){__builtin_puts("w");}
```
”

C++で 40 バイトの Hello World を書いた

- w を表示する C++のプログラム
- 全体で 28 Byte

Grass

Grass

> *Grass is a functional grass-planting programming language.*
>
> Grass the grass-planting programming language

- C++と同じくチューリング完全なプログラム言語
- `w` と `W` と `v` の組み合せでプログラムが記述できる
- 文法が次のように定義される

$$
\begin{aligned}
app &::= \mathtt{W}^{+}\ \mathtt{w}^{+} \\
abs &::= \mathtt{w}^{+}\ app^{*} \\
prog &::= abs \mid prog\ \mathtt{v}\ abs \mid prog\ \mathtt{v}\ app^{*}
\end{aligned}
$$

- 文法を**正規表現**で表せる
  `w((w|v)*(W+w)*)*`

w を表示するプログラム

w を表示するプログラム

```
wwWwvWWwwwwvwvwvWwvwWwwvvvvvv
```

- Grass で w を表示するプログラム
- このプログラムは 29 Byte
- C++より 1 Byte 大きい

RANS

RANS

> *RANS's concept is very simple, just calculates the number from the given string on a regular language.*
>
> http://sinya8282.github.io/RANS/

- ある文字列が、ある正規表現が表す集合の何番目に位置するのか計算するプログラム
- 文字列から番号、番号から文字列、どちらも変換できる
- 番号に変換すれば、C++並に短くなるはず

w を表示するプログラム

w を表示するプログラムは

469787137681

番目の Grass プログラム

- この数字はテキストで 12 Byte
- C++と比べて 16 Byte 小さい
- C++に勝利した

おすすめの文献


新屋良磨, 鈴木勇介, 高田謙.
正規表現技術入門 — 最新エンジン実装と理論的背景 (WEB+DB PRESS plus).
技術評論社, 4 2015.

Thank you for listening!

ポンピング補題

L が正規言語[‡] であるならば、次が成り立つ。

任意の正規言語 L について、言語 L についてのみ依存する反復長 $l > 0$ が存在し、$|w| \geq l$ となる全ての $w \in L$ について次が成り立つ

1. $|y| > 0$ かつ $|xy| \leq l$ となる2 を満す w の分割 $w = xyz$ が存在する
2. すべての $i \geq 0$ について $xy^iz \in L$ が成り立つ

[‡]正規表現で表現できる言語であるという意味

ポンピング補題

非素数言語を $L_{np} = \{1^{\alpha \cdot \beta} \mid \alpha > 1, \beta > 1\}$ として、
$w = 11\,11\,11 \in L_{np}, l = 1$ とする

反例

$l = 1$ かつ $|y| > 0, |xy| \leq l = 1$ より、$|x| = 0, |y| = 1$ となる。従って $w = \underbrace{}_{x}\underbrace{1}_{y}\underbrace{11111}_{z}$ とすると、

$|xy^2z| = |xyz| + |y| = 6 + 1 = 7$ となり、7 は素数であることから、$xy^2z \notin L_{np}$ となり、ポンピング補題を満さない