中級正規表現

大林一平

KMC/Dept. Math., Kyoto U./JST CREST

2013/08/17,24

Kobe Rubyist Meetup 1st

+ α

自己紹介

- 大林一平 (ohai)
- 理学研究科数学教室
  - 特定研究員
  - 力学系
  - 最近は歩行の数理モデルの相空間のグローバルな力学系構造の解析
- Ruby
  - るりま (http://doc.ruby-lang.org)
  - Ruby/SDL

題目

- 正規表現
- NFA
- 絶対最大量指定子とアトミックグループ
- ~~先読み，後読み，アンカー~~
- 名前付きキャプチャ
- 参考資料

正規表現

- 文字列を判別するためのプログラミング言語
  - 元は数学的な「言語」理論のために発明された
  - DSL
- 高度な制御機構
  - 高い抽象性
  - 後で説明する「バックトラック」
- 短く書ける構文
- 低い抽象化能力
  - 変数，関数，に相当する機能が貧弱
    - ないことはない
  - 上部の言語 (Ruby) との連携能力が弱い
    - 例えば正規表現の処理中に Ruby のコードを呼びだすことはできない
  - そのため何でも正規表現で処理するのはよくない

NFA

Non determistic Finite Automaton

- 大半の正規表現エンジンで使われている仕組み
  - Perl
  - PCRE(Perl Compatible Regular Expression)
    - ★ PHP, Apache, R, etc.
  - Oniguruma, Onigmo
    - ★ Ruby 1.9 以降, PHP5 以降
  - Python
- 別の仕組みとして DFA, Posix NFA, ハイブリッド, などがある
- DFA と比較して柔軟で複雑な処理が可能だが，性能には劣る

NFA の利点と欠点

DFA と比べて以下の利点がある.

- 後方参照 \1, ... が使える
- 先読みなどができる
- さらなる拡張が可能
  - 条件分岐
  - 部分式呼び出し
- Finite Automaton の数理的能力を越える

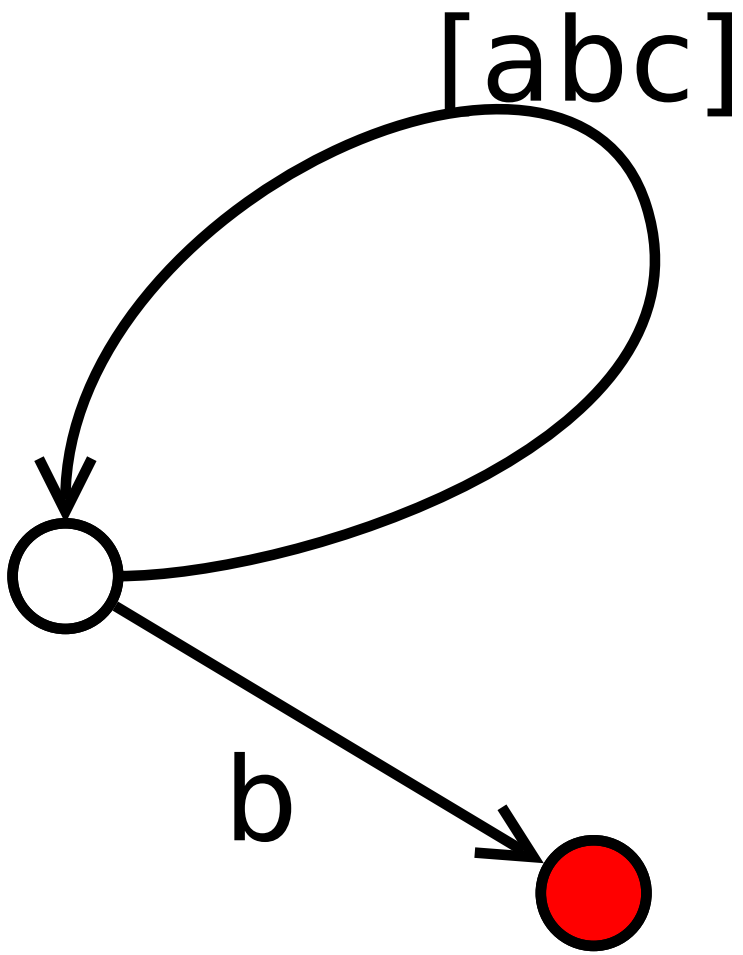
欠点は性能上 DFA に比べ劣ること. 特に NFA の場合は「うっかり」酷い性能の正規表現を書いてしまえる

バックトラック

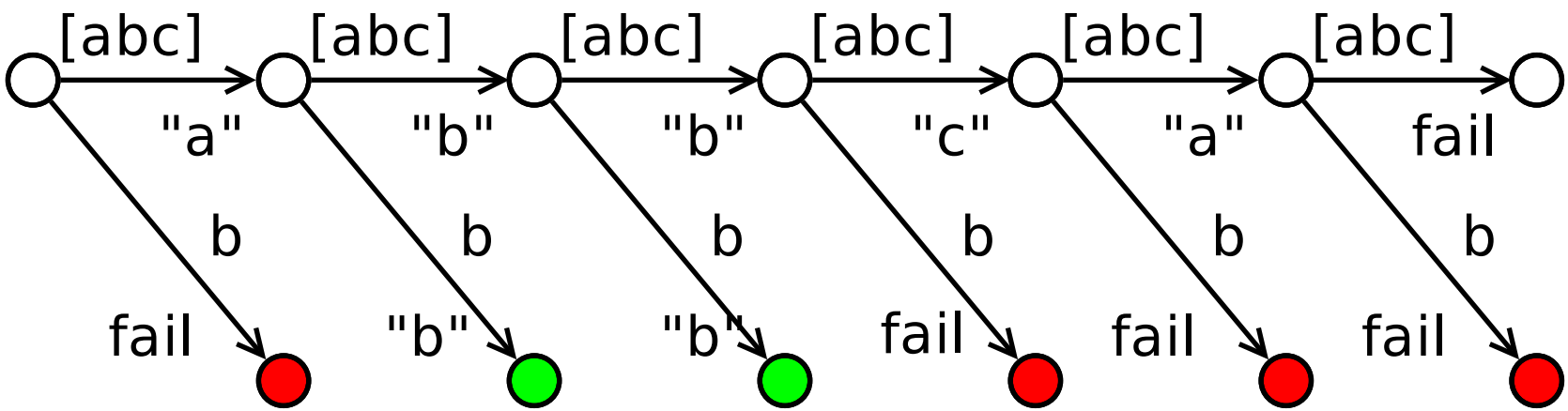
- NFA の動作の基本原理
- 再帰呼び出しのようなもの
  - 深さ優先探索
- 2 つの選択子がある場合，一方を保留しつつもう一方のマッチを試す
  - 例えば /a+b/ で “a” にマッチした場合，
    - ★ もう一文字 a のマッチをする
    - ★ b のマッチをする

    の 2 つがある
- 試したマッチがその先で失敗した場合，保留したところまで戻って別の選択子を試す

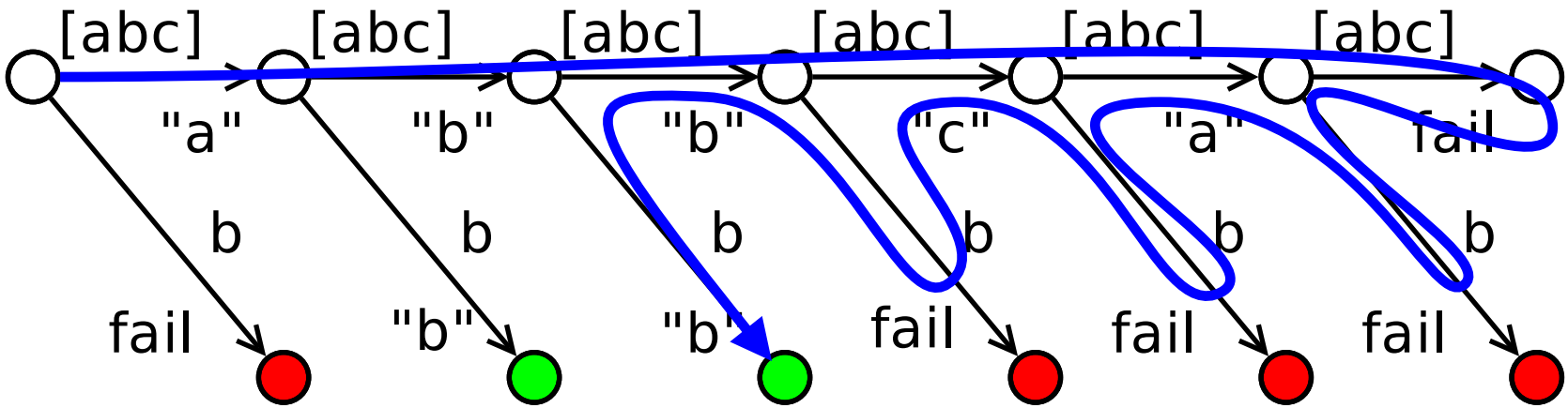
```
/[abc]*b/.match("abbca")
```



```
/[abc]*b/.match("abbca")
```

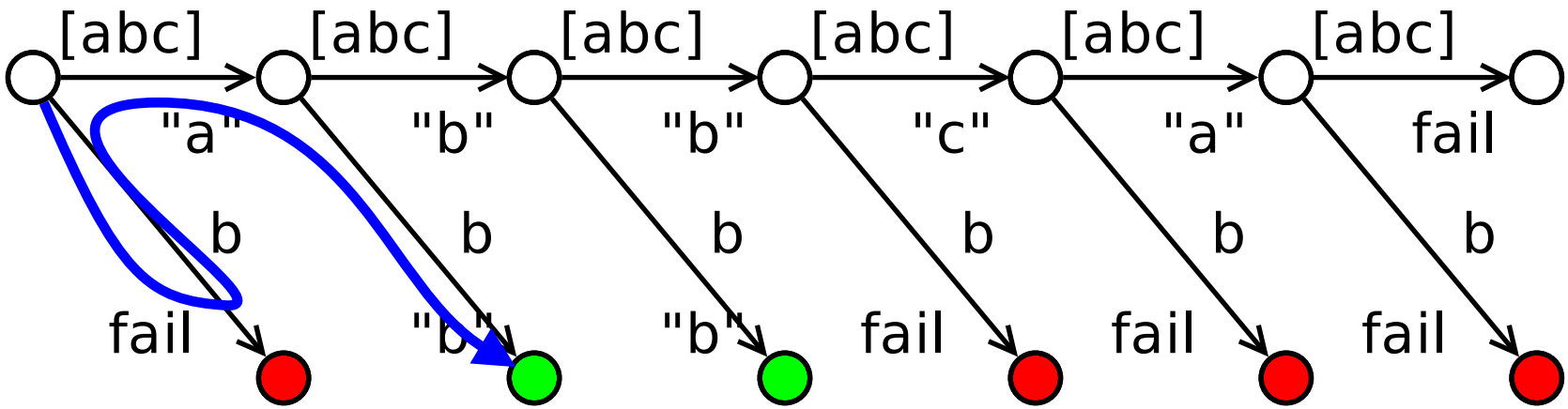


```
/[abc]*b/.match("abbca")
```



* は greedy なので，[abc] の繰り返しマッチを優先する

```
/[abc]*?b/.match("abbca")
```



*? は reluctant なので，b のマッチを優先する

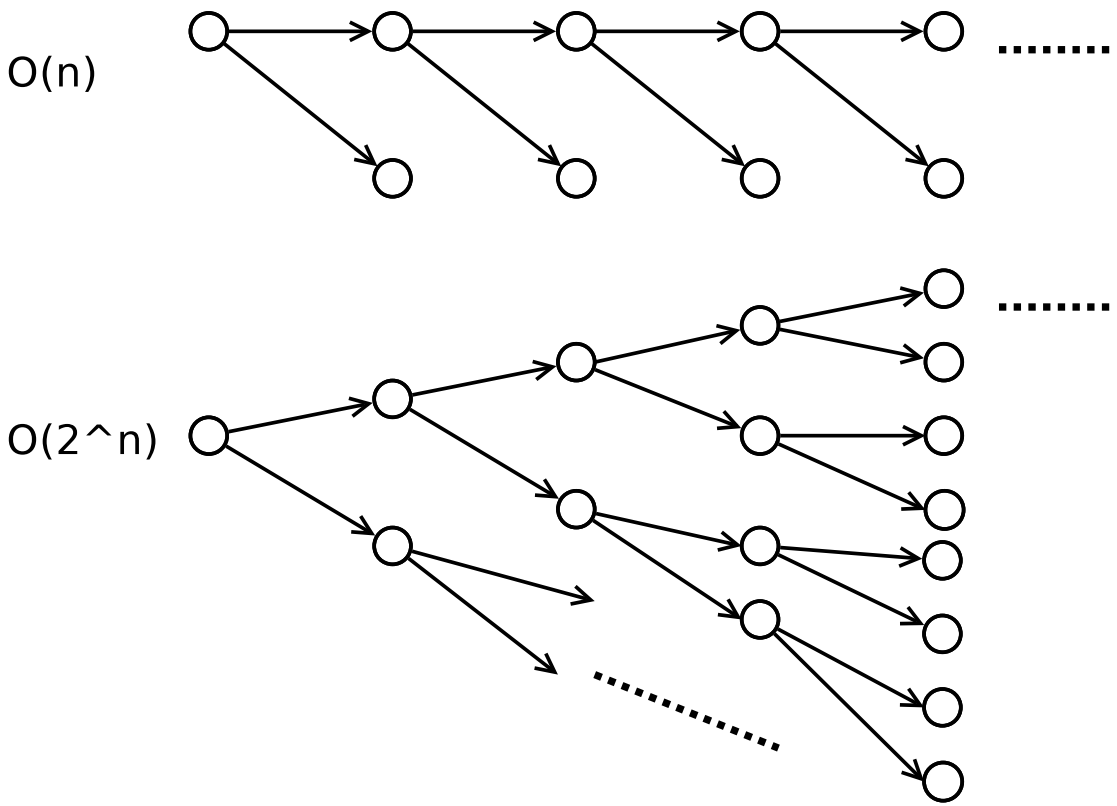
最小量指定子の活用例

ダブルクオートで囲まれたテキストを取り出す.

```
/"(.*?)"/
```

普通の量指定子を使うと 3 つ以上のダブルクオートが文字列に含まれていると問題になる.

正規表現の性能



tree 構造によって性能は劇的に異なる.

- 特にマッチが失敗する場合はチェックする場合の数が膨大になる可能性がある

特に量指定子がネストしていると場合の数の爆発が起きやすい

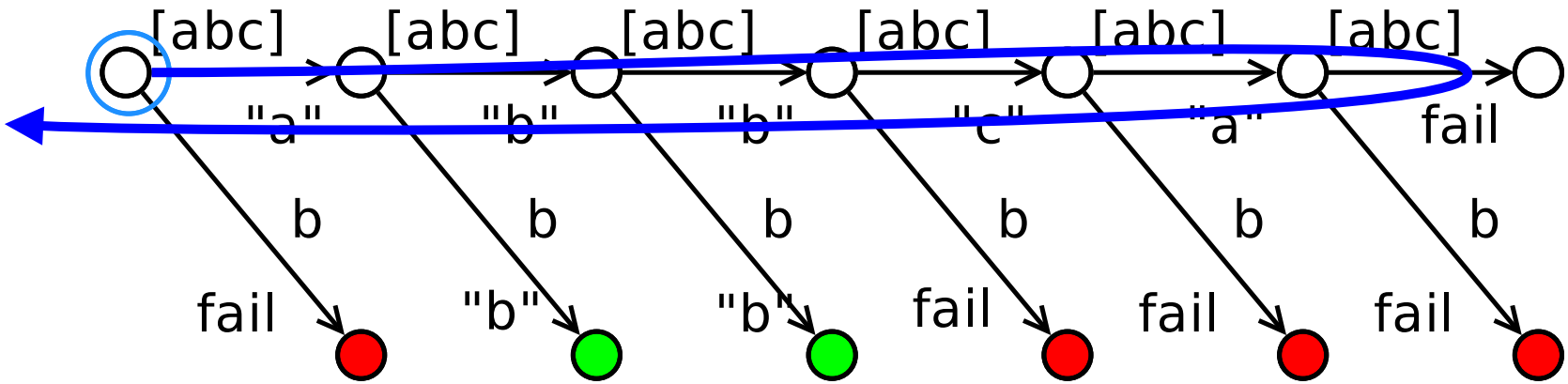
```
/X(.+)+X/
```

バックトラックの抑制

高機能な NFA エンジンには，バックトラックを抑制する仕組みを持つものがある．

- 絶対最大量指定子 (possessive quantifier)
  - 繰り返し内でのバックトラックを一切しない
- アトミックグループ
  - グループ内のマッチが一度成功すると，その内側ではバックトラックしなくなる

```
/[abc]*+b/.match("abbca")
```



水色の○の手前まで戻る (つまり失敗する)

- マッチが失敗しやすくなる
  - 失敗したほうが都合が良い場合に使えなくもない
- 性能の向上に使える
  - 特にマッチ失敗時の性能に良く効く

例

メールヘッダの取り出し

```
/\A[^:]+:/
/\A[^:]++:/
```

絶対最大量指定子を使うと，メールヘッダ「でない」場合には失敗するまでの計算時間が短くなる

名前付きキャプチャ

- キャプチャや後方参照は通常 1, 2, . . . , という名前が付けられる
- 識別子で名前を付けることができる
- (?<name>pat) でキャプチャし，\k<name> で後方参照
- MatchData#[sym] でマッチ後に参照できる
- =~ でリテラル正規表現をマッチする場合はローカル変数に束縛される
- $1 とかを使う必要がなくなる

例

```
str = "x = 128"
m = /\A(?<var>[a-zA-Z_])\s*=\s*(?<val>\d+)\s*\z/.match(str)
m[:var] # => "x"
m[:val] # => "128"
/\A(?<var>[a-zA-Z_])\s*=\s*(?<val>\d+)\s*\z/ =~ str
var # => "x"
val # => "128"
```

参考資料

- 詳説正規表現 (Mastering Regular Expressions) Jeffrey E. F. Friedl 著

Q & A

Q. 以下の 2 つの正規表現のうちどちらが良いの？

```
/"[^"]*"/  /".*?"/
```

A. 一般的に言って，最小量指定子は性能が良くないことが多いようです (上の例では数%の差が生じるようです)．また，最小量指定子を知らない人も多いので，前者を使ったほうが良いように思われます．もっと複雑な括弧の対応 (例えば HTML のタグ) を取る場合には最小量指定子が役に立つでしょう．