テキスト処理 第5回 (2008-05-20)

田中哲
産業技術総合研究所
情報技術研究部門
akr@isc.senshu-u.ac.jp
http://staff.aist.go.jp/tanaka-akira/textprocess-2008/

今日の内容

- 正規表現から文字列集合を生成する
- それを使って無理矢理正規表現エンジンを作ってみる
- レポート

正規表現から文字列集合を生成

- /cat|dog/ => "cat", "dog"
- /(c|h)at/ => "cat", "hat"
- /(c|f)a(n|t)/ => "can", "cat", "fan", "fat"
- /a*/ => "", "a", "aa", "aaa", ...
- /c(a|d)r/ => "cr", "car", "cdr", "caar", ...

基本的な正規表現

• 空集合 φ	何ともマッチしない
• 空文字列 ε	空文字列 "" のみにマッチする
• 文字 c	その文字 c 自身とマッチする
• 連接 r1 r2	r1 にマッチするものと r2 にマッチするものを連結したものにマッチする
• 選択 r1|r2	r1 にマッチするものか r2 にマッチするものか どちらかにマッチする
• 繰り返し r*	r にマッチするものの 0回以上の繰り返しにマッチする

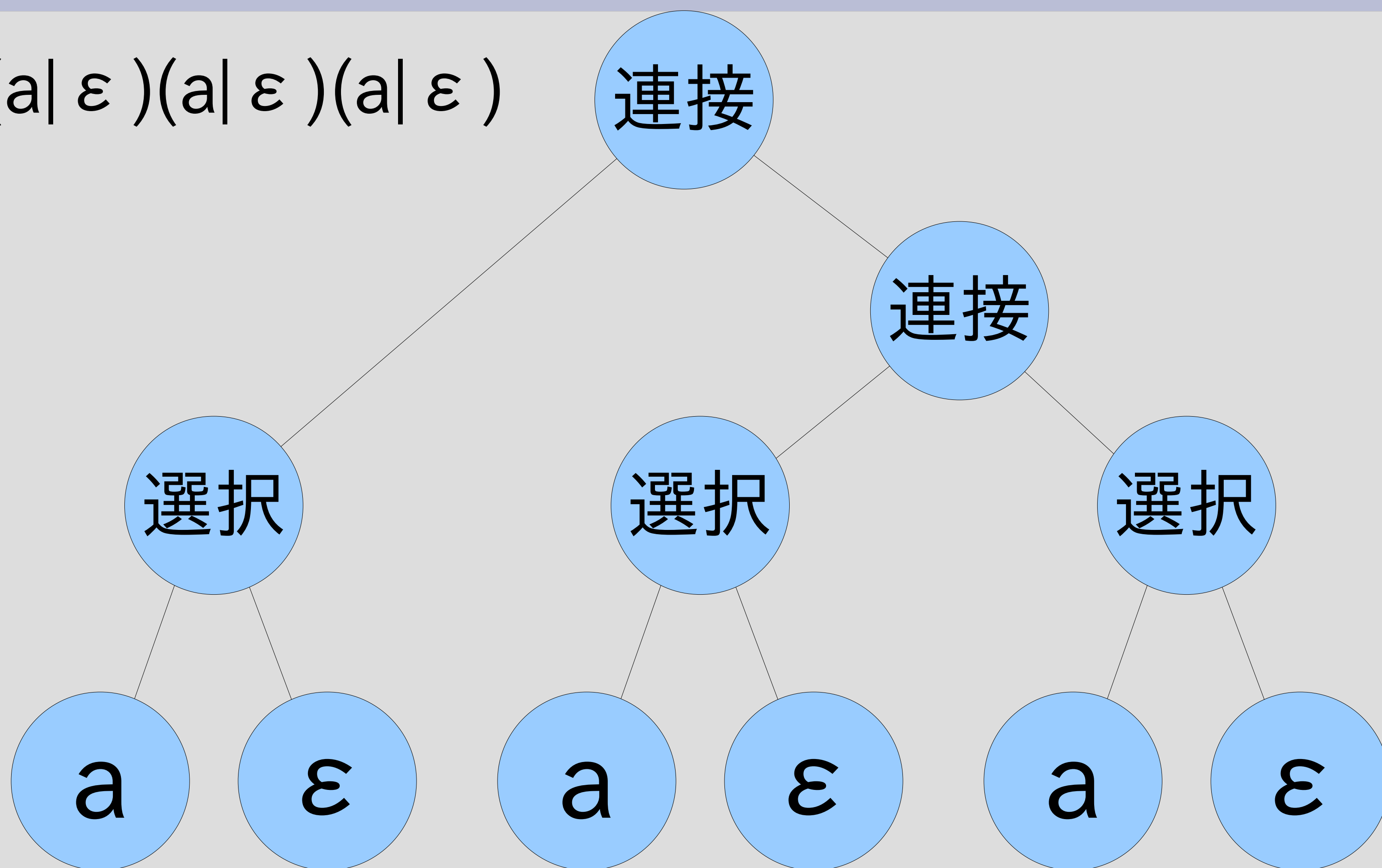
空文字列εと空集合φ

- εをRuby の正規表現で使う場合、空文字列にする
- /(a|)(a|)/ で (a|ε)(a|ε) の意味になる

- φをRuby の正規表現で使う必要があることはあまりない
- マッチしないならそもそも正規表現を使う意味がない

正規表現の例

- (a| ε)(a| ε)(a| ε)



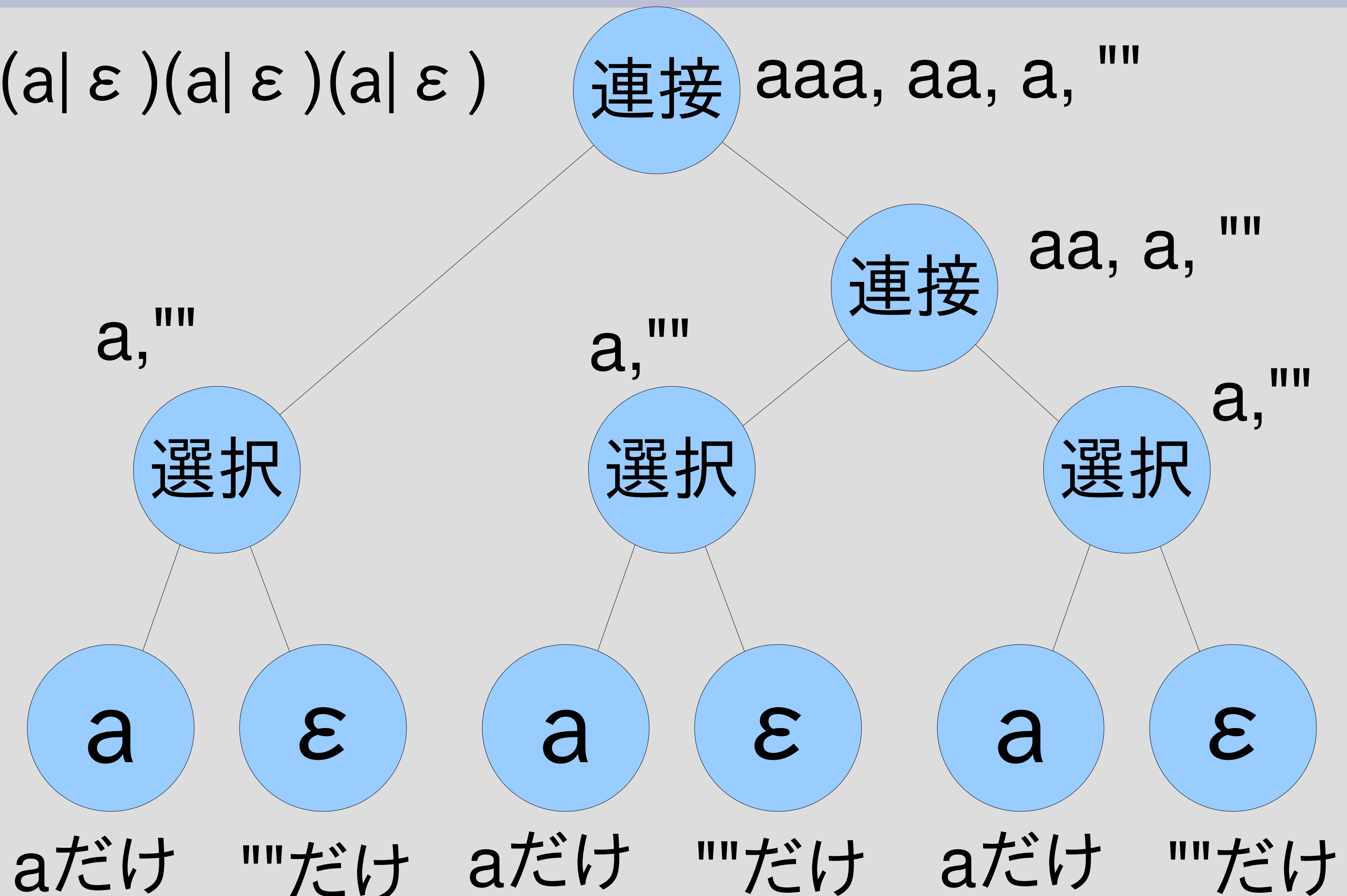
(a|ε)(a|ε)(a|ε)にマッチする文字列

- a にマッチするのは a だけ
- ε にマッチするのは "" だけ
- (a|ε) にマッチするのは a または ""
- (a|ε)(a|ε) にマッチするのは
  - a と a の連接で aa
  - a と "" の連接で a
  - "" と a の連接で a
  - "" と "" の連接で ""

  つまり結局 aa, a, ""
- (a|ε)(a|ε)(a|ε) にマッチするのは aaa, aa, a, ""

正規表現とマッチする文字列

- (a| ε)(a| ε)(a| ε)



文字列集合の生成

- 正規表現を受け取って文字列集合を返す
- 正規表現は配列で表現する
- 文字列集合は文字列の配列で表現する
- 繰り返しがあると無限に文字列が増えてしまうので扱わない

正規表現の配列表現

- 空集合 ϕ [:empset]
- 空文字列 ε [:empstr]
- 文字 c [:char, "c"]
- 連接 r1 r2 [:cat, R1, R2]
- 選択 r1|r2 [:alt, R1, R2]
- 繰り返し r* [:rep, R]

R1, R2, R は r1, r2, R を 配列に変換したもの

配列表現の例

- /cat|dog/
```
[:alt, [:cat, [:char, "c"],
              [:cat, [:char, "a"],
                     [:char, "t"]]],
       [:cat, [:char, "d"],
              [:cat, [:char, "o"],
                     [:char, "g"]]]]
```
- /(c|h)at/
```
[:cat, [:alt, [:char, "c"], [:char, "h"]],
       [:cat, [:char, "a"], [:char, "t"]]]
```

(a| ε)(a| ε)(a| ε)の配列表現

- a [:char, "a"]
- ε [:empstr]
- a| ε [:alt, [:char, "a"], [:empstr]]
- (a| ε)(a| ε)
  ```
  [:cat, [:alt, [:char, "a"], [:empstr]],
         [:alt, [:char, "a"], [:empstr]]]
  ```
- (a| ε)(a| ε)(a| ε)
  ```
  [:cat, [:alt, [:char, "a"], [:empstr]],
         [:cat, [:alt, [:char, "a"], [:empstr]],
                [:alt, [:char, "a"], [:empstr]]]]
  ```

文字列集合の生成

```
def enumre(r)
  case r[0]
  when :empset
    ???
  when :empstr
    ???
  when :char
    ???
  when :cat
    ???
  when :alt
    ???
  end
end
```

:char, :cat などで場合分けして生成する

:cat, :alt で再帰する

空集合 :empset の処理

```
def enumre(r)
  case r[0]
  when :empset
    ???
  when :empstr
    ???
  when :char
    ???
  when :cat
    ???
  when :alt
    ???
  end
end
```

[:empset] の処理の内容

```
when :empset
  []
```

- 空集合なので空の配列を返す
- enumre([:empset]) => []

空文字列 :empstr の処理

```
def enumre(r)
  case r[0]
  when :empset
    ???
  when :empstr
    ???
  when :char
    ???
  when :cat
    ???
  when :alt
    ???
  end
end
```

[:empstr] の処理の内容

```
when :empstr
  [""]
```

- 空文字列をひとつ含む配列を返す
- enumre([:empstr]) => []

文字 :char の処理

```
def enumre(r)
  case r[0]
  when :empset
    ???
  when :empstr
    ???
  when :char
    ???
  when :cat
    ???
  when :alt
    ???
  end
end
```

[:char, c] の処理理の内容

```
when :char
  [r[1]]
```

- 空文字列をひとつ含む配列を返す
- enumre([:char, "a"]) => ["a"]

連接 :cat の処理

```
def enumre(r)
  case r[0]
  when :empset
    ???
  when :empstr
    ???
  when :char
    ???
  when :cat
    ???
  when :alt
    ???
  end
end
```

[:cat, r1, r2] の処理の内容

```
when :cat
  set1 = enumre(r[1])
  set2 = enumre(r[2])
  ret = []
  set1.each {|str1|
    set2.each {|str2|
      ret << str1+str2
    }
  }
  ret
```

- r1, r2 について再帰してそれぞれ文字列集合を求める
- その組み合わせを2重ループで計算する
- Array#<< は配列に要素を追加する
- enumre(
  [:cat, [:char, "a"],
  [:char, "b"]])
  =>
  ["ab"]

Array#<<

- 配列の末尾に要素を追加する

- 例

```
a = [1,2,3]
a << 100
p a                #=> [1,2,3,100]
```

選択 :alt の処理

```
def enumre(r)
  case r[0]
  when :empset
    ???
  when :empstr
    ???
  when :char
    ???
  when :cat
    ???
  when :alt
    ???
  end
end
```

[:alt, r1, r2] の処理の内容

```
when :alt
  set1 = enumre(r[1])
  set2 = enumre(r[2])
  set1 + set2
```

- r1, r2 について再帰してそれぞれ文字列集合を求める
- 集合和を配列の連結で求める
- enumre(
  [:alt, [:char, "a"],
  [:char, "b"]])
  =>
  ["a", "b"]

実行例

- a
  enumre([:char, "a"])#=> ["a"]
- ε
  enumre([:empstr])　　#=> [""]
- a| ε
  enumre([:alt, [:char, "a"], [:empstr]])
  #=> ["a", ""]
- (a| ε)(a| ε)
  enumre([:cat, ..., ...])　#=> ["aa", "a", "a", ""]
- (a| ε)(a| ε)(a| ε)
  enumre([:cat, ..., [:cat, ..., ...]])
  #=> ["aaa", "aa", "aa", "a", "aa", "a", "a", ""]

配列表現の例

- /cat|dog/

```
enumre([:alt, [:cat, [:char, "c"],
                     [:cat, [:char, "a"],
                            [:char, "t"]]],
              [:cat, [:char, "d"],
                     [:cat, [:char, "o"],
                            [:char, "g"]]]])
#=> ["cat", "dog"]
```

正規表現エンジン

- 正規表現と文字列が与えられたときマッチするかどうか調べる機構
- /¥Ar¥z/ =~ str
  str が r とちょうど対応するかどうか
  - ¥A は文字列の先頭
  - ¥z は文字列の末尾
- /r/ =~ str
  str に r に対応する部分が含まれているかどうか

enumre による正規表現エンジン

- /¥Ar¥z/ =~ str

```
def ematch_exact(r, str)
  enumre(r).include? str
end
```

Array#include?

- 配列にある要素が含まれているかどうか調べる
- 例
  - [1,2,3].include?(2)　　　#=> true
  - [1,2,3].include?(100)　　#=> false
  - ["apple", "orange"].include?("apple")
    #=> true
  - ["banana", "pineapple"].include?("apple")
    #=> false

ematch_exact の実行例

- ematch_exact([:char, "a"], "a")　#=> true
- ematch_exact([:char, "a"], "b")　#=> false
- ematch_exact(

```
[:alt, [:cat, [:char, "c"],
               [:cat, [:char, "a"],
                      [:char, "t"]]],
       [:cat, [:char, "d"],
               [:cat, [:char, "o"],
                      [:char, "g"]]]], "dog")
#=> true
```

enumre による正規表現エンジン (2)

- /r/ =~ str

```
def ematch_include(r, str)
  enumre(r).each {|s|
    return true if str.include? s
  }
  false
end
```

String#include?

- 文字列にある部分文字列が含まれているかどうか調べる
- 例
  - "abc".include?("b")　　　#=> true
  - "abc".include?("z")　　　#=> false
  - "pineapple".include?("apple") #=> true
  - "orange".include?("apple")　　#=> false

ematch_include の実行例

- ematch_include([:char, "a"], "abc")
  #=> true
- ematch_include([:char, "z"], "abc")
  #=> false
- ematch_include(

```
[:alt, [:cat, [:char, "c"],
              [:cat, [:char, "a"],
                     [:char, "t"]]],
       [:cat, [:char, "d"],
              [:cat, [:char, "o"],
                     [:char, "g"]]]], "education")
#=> true
```

問題点

- 繰り返しが扱えない
  [:rep, r]
- 無駄が多い
  - 可能性の数に比例したメモリが必要
    連接を繰り返すと指数関数的に長くなる
  - 無駄に可能性を試す
    最初の方でマッチしないことが分かれば、その後は
    無駄

繰り返しの扱い

- 繰り返しがあると結果は無限集合になる
  a*　　　=> "", a, aa, aaa, aaaa, ...
  (foo)*　=> "", foo, foofoo, foofoofoo, ...
  (a|b)*　=> "", a, b, aa, ab, ba, bb, aaa, ...
- 配列の長さは有限なので無限集合は扱えない

指数関数的なメモリ消費

- a|b => a, b の 2個
- (a|b)(a|b) => aa, ab, ba, bb の 4個
- (a|b)(a|b)(a|b) => aaa, aab, aba, abb,
  baa, bab, bba, bbb の8個
- (a|b)(a|b)(a|b)(a|b) => 16個
- (a|b)(a|b)(a|b)(a|b)(a|b)(a|b)(a|b)(a|b)=> 256個
- (a|b) を n個連接 => 2**n個

無限でなく有限ではあるが、
メモリに格納するには多すぎる

可能性の無駄な探索

- 文字列 abcdef
- 正規表現 (a|b|c)(a|b|c)(a|b|c)

まともな正規表現エンジン

- 繰り返しを扱う
- メモリ消費がひどくない
- 無駄な探索はあまりしない

そういう正規表現エンジンは次回以降

レポート

- enumre を以下のように拡張せよ
  - 単一文字を配列でなく記述できるようにする
  - :cat で任意個の引数をとれるようにする
- 以下について考察せよ
  - 繰り返しを使っても結果が無限集合にならない場合
- 〆切 2008-05-27 12:00
- RENANDI
- 拡張子が txt なプレインテキストが望ましい
  - docx は読めません (MS Word の XML 形式?)
  - doc も避けてください (MS Word ファイル)

単一文字の配列でない記述

- [:char, "x"] を "x" と書けるようにする
- [:alt, [:cat, [:char, "c"],
  [:cat, [:char, "a"],
  [:char, "t"]]],
  [:cat, [:char, "d"],
  [:cat, [:char, "o"],
  [:char, "g"]]]] を
  [:alt, [:cat, "c", [:cat, "a", "t"]],
  [:cat, "d", [:cat, "o", "g"]]] と書けるようにする

ヒント

- 文字列 (だけ) は to_str メソッドを持つ

:cat の任意個引数

- [:cat, "d", [:cat, "o", "g"]] と書くのは面倒
- [:cat, "d", "o", "g"] と書きたい
- なので書けるようにせよ

繰り返しを使っても結果が無限集合にならない場合

- 繰り返しを使うと正規表現に対応する文字列集合はだいたい無限集合になる
  - a*　　#=> "", a, aa, aaa, aaaa, ...
  - (a|b)* #=> "", a, b, aa, ab, ba, bb, aaa, ...
  - (pine|apple)* #=> "", pine, apple, pinepine, pineapple, applepine, appleapple, ...
- じつは無限集合にならない場合が存在する
- それはどんな場合か考えよ

まとめ

- 正規表現から文字列集合を生成する
  - やっぱり再帰
- 無理矢理正規表現エンジンを作ってみる
  - ある程度は動く
- レポートを出した