テキスト処理 第10回 (2008-06-24)

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

- 前回のレポートの説明
- rep の無限再帰防止
- 再帰を使う拡張
  - 存在するかもしれない: r?
  - 1回以上の繰り返し: r+
  - 怠惰な繰り返し: r*?
  - 存在しないかもしれない: r??
  - m回以上 n回以下の繰り返し: r{m,n}
- レポート

rep の無限再帰

- 無限再帰することがある
- 例: try([:rep, [:empstr]], ["a"], 0, {}) {}

```
def try_rep(re, str, pos, md, &b)
  try(re[1], str, pos, md) {|pos2, md2|
    try(re, str, pos2, md2, &b)
  }
  yield pos, md
end
```

もし pos == pos2 なら、
try は同じ引数で
再帰呼出しされる

rep の無限再帰防止

- pos2 が少しでも進んでいる場合のみ再帰する
- マッチするものがマッチしなくなることはない

```
def try_rep(re, str, pos, md, &b)
  try(re[1], str, pos, md) {|pos2, md2|
    try(re, str, pos2, md2, &b) if pos < pos2
  }
  yield pos, md
end
```

もし pos == pos2 なら、
try は呼び出さない

エンジンの拡張: /r?/

- /r?/ は、e がある場合とない場合にマッチする
- r がある場合を先に試し、ない場合を後に試す
- /r|/ と同じ
- [:opt, r] で表現する (optional の意)

- /behaviou?r/ =~ "behavior"   #=> 0
- /behaviou?r/ =~ "behaviour"   #=> 0

- rx_ends([:opt, "a"], "a", 0)   #=> [1,0]
- rx_ends([:opt, "a"], "b", 0)   #=> [0]

[:opt, r] の実装 (1)

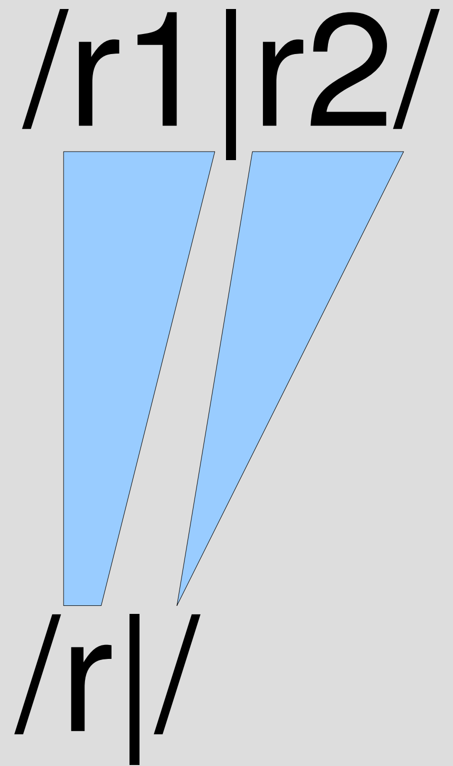
```
def try(re, str, pos, md, &b)
  ...
  when :opt
    try_opt(re, str, pos, &b)
  ...
end
```

[:opt, r] の実装 (2)

```
def try_opt(re, str, pos, md, &b)
  try(re[1], str, pos, md, &b)
  yield pos, md
end
```

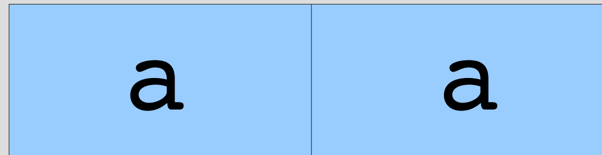
try_alt と try_opt の比較

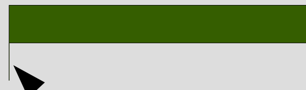
```
def try_alt(re, str, pos, md, &b)
  try(re[1], str, pos, md, &b)
  try(re[2], str, pos, md, &b)
end

def try_opt(re, str, pos, md, &b)
  try(re[1], str, pos, md, &b)
  yield pos, md
end
```



try(//) を展開した形になっている

rx_ends([:opt, "a"], "aa")

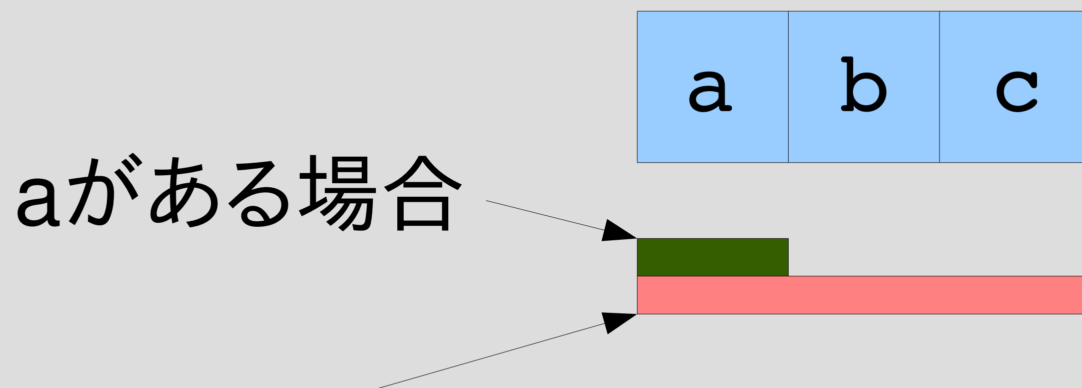




最初に a がある場合

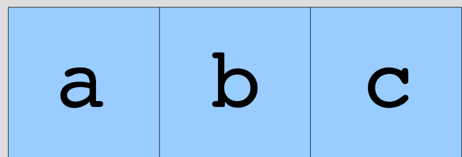
後で a がない場合

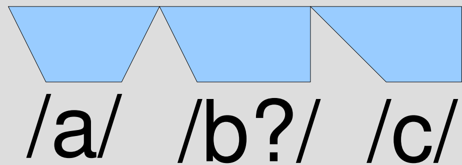
/a?abc/ =~ "abc"

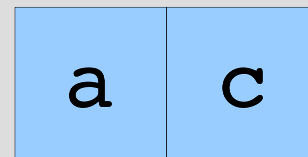


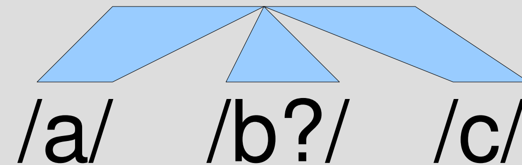
aがなくてabcが続く場合

/ab?c/ =~ "abc", /ab?c/ =~ "ac"









エンジンの拡張: /r+/

- /r+/ は、e の 1つ以上の繰り返し
- [:plus, r] で表現する
- /rr*/ と同じ

- /ab+c/ =~ "ac"   #=> nil
- /ab+c/ =~ "abc"   #=> 0
- /ab+c/ =~ "abbbc"   #=> 0

- rx_ends([:plus, "a"], "aaa", 0) #=> [3,2,1]
- rx_ends([:rep, "a"], "aaa", 0)   #=> [3,2,1,0]

[:plus, e] の実装 (1)

```
def try(re, str, pos, md, &b)
  ...
    when :plus
      try_plus(re, str, pos, md, &b)
  ...
end
```

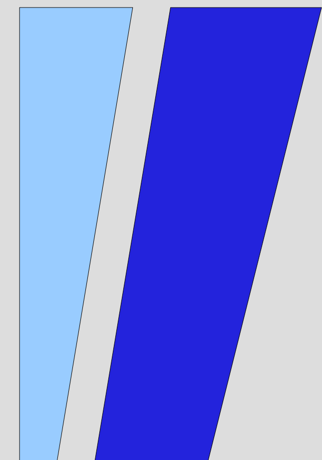
[:plus, r] の実装 (2)

```
def try_plus(re, str, pos, md, &b)
  try(re[1], str, pos, md) {|pos2, md2|
    try([:rep, re[1]], str, pos2, md2, &b)
  }
end
```

try_cat と try_plus の比較

```
def try_cat(re, str, pos, md, &b)
  try(re[1], str, pos, md) {|pos2, md2|
    try(re[2], str, pos2, md2, &b)
  }
end
def try_plus(re, str, pos, md, &b)
  try(re[1], str, pos, md) {|pos2, md2|
    try([:rep, re[1]], str, pos2, md2, &b)
  }
end
```

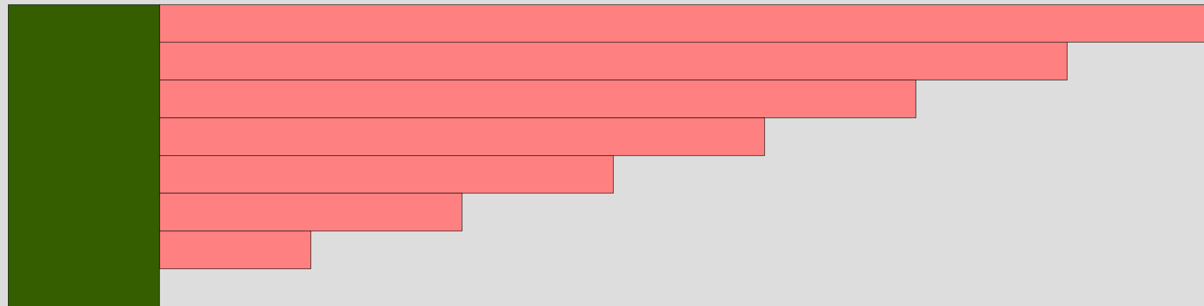
/r1r2/



/rr*/

try(/r*/) を展開した
構造になっている

/a+/ の動作



エンジンの拡張: /r*?/

- /r*?/ は、r の 0個以上の繰り返し
- [:rep_lazy, r] で表現する
- r* とは逆に、少ない繰り返しから試す
- いままでの組合せでは表現できない

- rx_ends([:rep_lazy, "a"], "aaa", 0)
  #=> [0,1,2,3]
- rx_ends([:rep, "a"], "aaa", 0)
  #=> [3,2,1,0]

r*? と (r*)? の違い

- r*? は [:rep_lazy, r]
- (r*)? は [:opt, [:rep, r]]
- *? はひとつの機能で、* と ? の組合せではない

lazy

- a* は繰り返しが多い場合から続きを試す
  この順序を greedy (貪欲) という
  とりあえずたくさん食べてみる、というイメージ

- a*? は繰り返しが少ない場合から続きを試す
  この順序を lazy (怠惰) もしくは nongreedy (非貪欲) という
  なるべくなら食べないで済ます、というイメージ

- 最終的にはすべて試すのでマッチするかどうかが変わることはない
  (マッチの場所やキャプチャは変わるかもしれない)

r*? の用途 (1)

- C のコメントを取り出す
  - /¥/¥*.*?¥*¥// =~ "ab /* ccc */ de /* xxx */"
  - /¥/¥*.*¥*¥// =~ "ab /* ccc */ de /* xxx */"

*? でなく * を使うと
複数のコメントにマッチしてしまう

/¥/¥*[^¥*]*¥*+([^¥/¥*][^¥*]*¥*+)*¥// とすれば
*? を使わなくても書ける (むしろ正しいが、難しい)

r*? の用途 (2)

- HTML のタグの対を取り出すのにも使われる
  - /.*?<¥/b>/ =~ "aabbbcccdddee"
  - /.*<¥/b>/ =~ "aabbbcccdddee"
- 残念ながらあまり正しいやりかたではない
  - ネストしていたらうまくいかない
  - /.*?<¥/b>/ =~ "aabbbcccdddee"
  - 閉じタグがないとうまくいかない
  - /.*?<¥/b>/ =~ "
bbbccc
dddee
"

[:rep_lazy, r] の実装 (1)

```
def try(re, str, pos, md, &b)
  ...
  when :rep_lazy
    try_rep_lazy(re, str, pos, md, &b)
  ...
end
```

[:rep_lazy, r] の実装 (2)

```
def try_rep_lazy(re, str, pos, md, &b)
  yield pos, md
  try(re[1], str, pos, md) {|pos2, md2|
    try(re, str, pos2, md2, &b) if pos < pos2
  }
end
```

rep と rep_lazy

```
def try_rep(re, str, pos, md, &b)
  try(re[1], str, pos, md) {|pos2, md2|
    try(re, str, pos2, md2, &b) if pos < pos2
  }
  yield pos, md
end
```

後に yield

```
def try_rep_lazy(re, str, pos, md, &b)
  yield pos, md
  try(re[1], str, pos, md) {|pos2, md2|
    try(re, str, pos2, md2, &b) if pos < pos2
  }
end
```

先に yield

greedy: /a*/ =~ "aa"

try(/a*/, "aa", 0)

re は実際には配列表現

"aa" も実際には配列
キャプチャ情報は省略

greedy: /a*/ =~ "aa"

2,1,0

下が成功したそれぞれについて

try(/a*/, "aa", pos)

try(/a/, "aa", 0)

成功

try(/a*/, "aa", 0)

greedy: /a*/ =~ "aa"

2,1,0

try(/a*/, "aa", 1)

下は1回しか成功しない

try(/a/, "aa", 0)

成功

try(/a*/, "aa", 0)

greedy: /a*/ =~ "aa"

2,1,0

下の成功それぞれについて

try(/a*/, "aa", pos)

try(/a/, "aa", 1)

try(/a*/, "aa", 1)

try(/a/, "aa", 0)

成功

try(/a*/, "aa", 0)

greedy: /a*/ =~ "aa"

2,1,0

try(/a*/, "aa", 2)
下は1回しか成功しない

try(/a/, "aa", 1)

try(/a*/, "aa", 1)

try(/a/, "aa", 0)

成功

try(/a*/, "aa", 0)

greedy: /a*/ =~ "aa"

2,1,0

下の成功について

try(/a*/, "aa", pos)

try(/a/, "aa", 2) 成功

try(/a*/, "aa", 2)

try(/a/, "aa", 1) 成功

try(/a*/, "aa", 1)

try(/a/, "aa", 0) 成功

try(/a*/, "aa", 0)

greedy: /a*/ =~ "aa"

2,1,0

下は成功しない

try(/a/, "aa", 2) 成功

try(/a*/, "aa", 2)

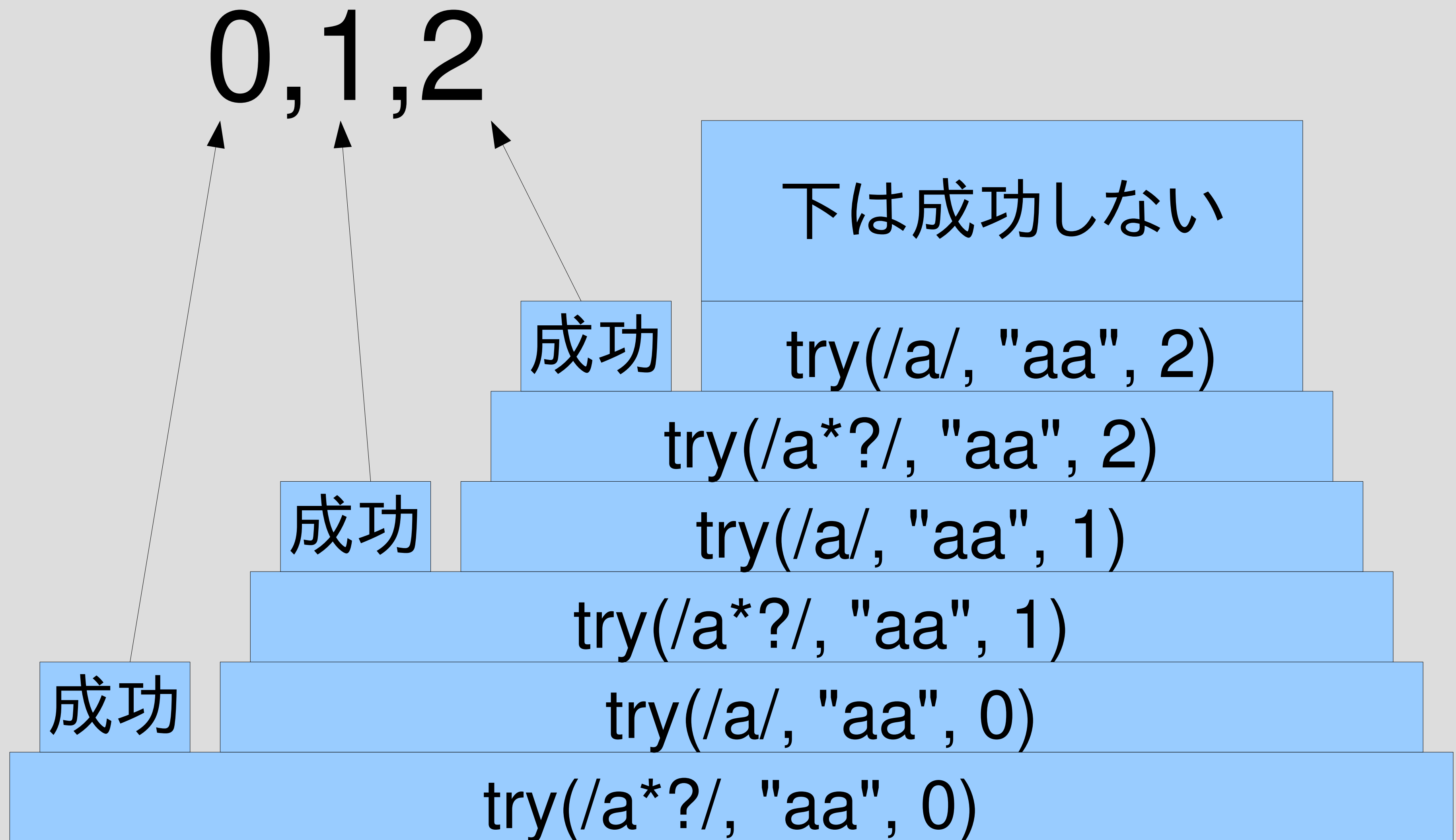
try(/a/, "aa", 1) 成功

try(/a*/, "aa", 1)

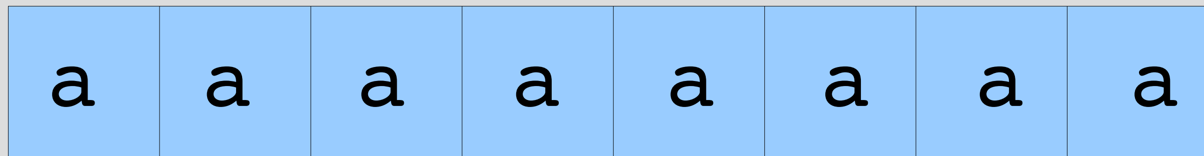
try(/a/, "aa", 0) 成功

try(/a*/, "aa", 0)

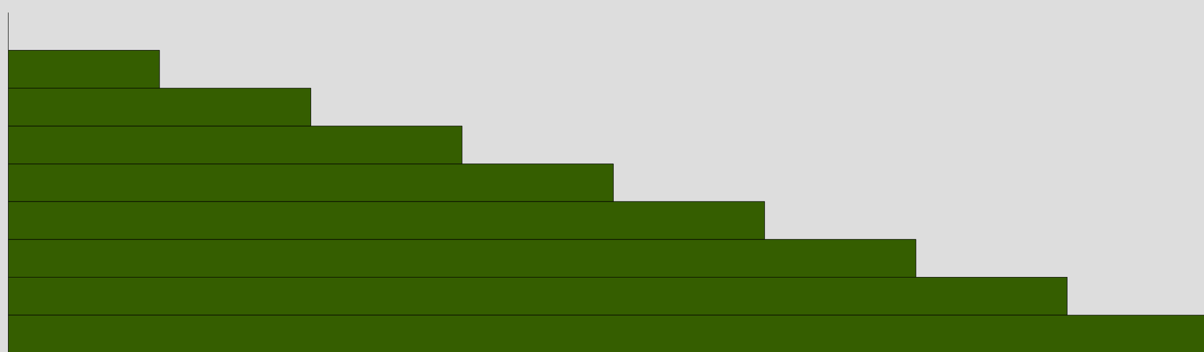
lazy: /a*?/ =~ "aa"



a*? の動作



aが0回



aが8回

エンジンの拡張: /r??/

- /r??/ は、r がない場合とある場合にマッチする
- r がない場合を先に試し、ある場合を後に試す
- /|r/ と同じ
- [:opt_lazy, r] で表現する

- /behaviou??r/ =~ "behavior"   #=> 0
- /behaviou??r/ =~ "behaviour"   #=> 0

- rx_ends([:opt_lazy, "a"], "aa", 0) #=> [0,1]
- rx_ends([:opt_lazy, "a"], "b", 0)   #=> [0]

r? と r??

- r? は r がある場合を先に試す: greedy
- r?? は r がない場合を先に試す: lazy

[:opt_lazy, r] の実装 (1)

```
def try(re, str, pos, md, &b)
  ...
    when :opt_lazy
      try_opt_lazy(re, str, pos, md, &b)
  ...
end
```

[:opt_lazy, r] の実装 (2)

```
def try_opt_lazy(re, str, pos, md, &b)
  yield pos, md
  try(re[1], str, pos, md, &b)
end
```

try_opt と try_opt_lazy の比較

```
def try_opt(re, str, pos, md, &b)
  try(re[1], str, pos, md, &b)
  yield pos, md                              後に yield
end

def try_opt_lazy(re, str, pos, md, &b)
  yield pos, md                              先に yield
  try(re[1], str, pos, md, &b)
end
```

エンジンの拡張: **r{m,n}**

- /r{m,n}/ は r の m回以上 n回以下の繰り返し
- 抽象構文木では [:times, m, n, r] で表現
- たくさん繰り返した方から試す (greedy)

- rx_ends([:times, 2, 4, "a"], "aaaaa", 0)
  #=> [4,3,2]
- rx_ends([:times, 2, 4, "a"], "aaa", 0)
  #=> [3,2]
- rx_ends([:times, 2, 4, "a"], "a", 0)
  #=> []

a{2,4} の動作

[:times, m, n, r] の実装 (1)

```
def try(re, str, pos, md, &b)
  ...
  when :times
    try_times(re, str, pos, md, &b)
  ...
end
```

[:times, m, n, r] の実装 (2)

```
def try_times(re, str, pos, md, &b)
  m = re[1]; n = re[2]; r = re[3]
  if 0 < n
    try(r, str, pos, md) {|pos2, md2|
      try([:times, m-1, n-1, r], str, pos2, md2, &b)
    }
  end
  yield pos, md if m <= 0
end
```

各種繰り返し

	0〜∞	0〜1	1〜∞
greedy	r*	r?	r+
lazy	r*?	r??	r+?

	m〜n	m	m〜∞
greedy	r{m,n}	r{m}	r{m,}
lazy	r{m,n}?	r{m}?	r{m,}?

レポート

- 以下を実装して解説せよ
  - r+?
  - r{m,}

- 実装したらユニットテストで確認すること
- 〆切 2008-07-01 12:00
- RENANDI
- 拡張子が txt なテキストファイル希望

/r+?/

- /r+/ の lazy 版
- 配列表現では [:plus_lazy, r]

- rx_ends([:plus_lazy, "a"], "aaa", 0) #=> [1,2,3]

/r{m,}/

- m回以上の r の繰り返し
- 配列表現では [:moretimes, m, r]
- greedy

- rx_ends([:moretimes, 3, "a"], "aa", 0) #=> []
- rx_ends([:moretimes, 3, "a"], "aaa", 0) #=> [3]
- rx_ends([:moretimes, 3, "a"], "aaaa", 0) #=> [4,3]
- rx_ends([:moretimes, 3, "a"], "aaaaa", 0) #=> [5,4,3]

まとめ

- 前回のレポートの説明
- rep の無限再帰防止
- 再帰を使う拡張
  - 存在するかもしれない: r?
  - 1回以上の繰り返し: r+
  - 怠惰な繰り返し: r*?
  - 存在しないかもしれない: r??
  - m回以上n回以下の繰り返し: r{m,n}
- レポートを出した