テキスト処理 第11回 (2007-07-17)

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-2007/

今日の内容

- 前回のレポートの説明
- キャプチャの説明
- MatchData
- キャプチャの実装
- 文字列置換
- レポート

キャプチャ

- マッチ全体ではなく、その一部を得る
- パターン中の丸括弧に対応するところを得る
- 特殊変数 $1, $2, $3, ... で対応する箇所を参照する

```
p /(a*)(b*)(c*)/ =~ "abbccc"    #=> 0
p $1                            #=> "a"
p $2                            #=> "bb"
p $3                            #=> "ccc"
p /(.*)=(.*)/ =~ "foo=bar"      #=> 0
p [$1, $2]                      #=> ["foo", "bar"]
```

キャプチャの使用例

- Ruby プログラムから定義しているメソッド名を取り出す (完璧ではない)

```
ARGF.each {|line|
  if /def +([A-Za-z0-9_]+[?!]?)/ =~ line
    puts $1
  end
}
```

括弧を通らない場合

- 全体がマッチしても、括弧の部分を通ってなければ nil になる
- /(a)|(b)|(c)/ =~ "b"
  p $1　　#=> nil
  p $2　　#=> "b"
  p $3　　#=> nil

括弧のネスト

- 括弧の番号は、左括弧の位置で決まる
- re = /a(b(c)d(e)f(g(h(i)j)k(l)m)n)o/

```
#        1 2    3    4 5 6       7
re =~ "abcdefghijklmno"
p $1     #=> "bcdefghijklmn"
p $2     #=> "c"
p $3     #=> "e"
p $4     #=> "ghijklm"
p $5     #=> "hij"
p $6     #=> "i"
p $7     #=> "l"
```

shy group

- キャプチャしないグループには (?:...) を使う
- /(?:a|b)*/ =~ "abba"
  p $1　　#=> nil

MatchData

- MatchData オブジェクトはマッチ情報を保持する
- マッチに成功した後、特殊変数 $~ で参照できる
- p $~ #=> nil
  p /b+/ =~ "abbccc" #=> 1
  m = $~
  p m #=> #<MatchData:0xb7dbb7c8>
  p m.pre_match #=> "a" マッチ前の文字列
  p m[0] #=> "bb" マッチした文字列
  p m.post_match #=> "ccc" マッチ後の文字列
  p m.begin(0) #=> 1 マッチ開始位置
  p m.end(0) #=> 3 マッチ終了位置
- $~[0] などの 0 はマッチ全体を意味する

特殊変数 $&

- $& は $~[0] とほぼ同じ
- $~ が nil のときには $& も nil
- /b+/ =~ "abbccc"
  p $&　　#=> "bb"
  $~ = nil
  p $&　　#=> nil

特殊変数 $1, $2, ...

- $1, $2, $3, ... は $~[1], $~[2], $~[3], ... に対応
- $~.begin(1), $~.end(1) で $1 の位置を得られる

```
/(a+)(b+)(c+)/ =~ "abbccc"
m = $~
p m[2]          #=> "bb"
p m.begin(2)    #=> 1
p m.end(2)      #=> 3
```

- $~[0], $~.begin(0) の「0」はマッチ全体を表現
- 注意: マッチした部分全体は $& で、$0 ではない

named capture

- 番号は扱いにくい
  - パターンを変更すると番号がずれる
  - 大きなパターンでは数えるのが大変
- Ruby 1.9 (開発版) では名前をつけられる

- p /(?<key>.*)=(?<val>.*)/ =~ "foo=bar"
  #=> 0
  p $~[:key]   #=> "foo"
  p $~[:val]   #=> "bar"

- (?<name>pat) にマッチしたものは $~[:name] で取り出せる

正規表現エンジンの拡張: キャプチャ

- 番号をつけるのは面倒なので named capture を実装する
- (MatchData ではなく) Hash で名前と場所の対応を保持する
- try の引数とブロック引数に Hash を加える
- Hash の鍵はキャプチャの名前
- Hash の値は範囲を表現する Range

Hash

- 整数以外でもアクセスできる Array みたいなもの
- 鍵(key) と値(value) の対応を記録
- {key1=>val1, key2=>val2, ...} で表現
- 今回はシンボルを鍵として使う
- h = {}　　# 空ハッシュ
  h[:foo] = "bar"
  h[:hoge] = "fuga"
  p h　　#=> {:hoge=>"fuga", :foo=>"bar"}
  p h[:foo]　　#=> "bar"
  p h[:baz]　　#=> nil

Range

- 範囲を表すオブジェクト
- 3..7 とか 2...5 とか
- 点がふたつのは終端を含む
- 点がみっつのは終端を含まない
- r = 2...5

```
p r.begin                  #=> 2
p r.end                    #=> 5
p r.exclude_end?           #=> true
p (2..5).exclude_end?      #=> false
```

- じつは ary[s..e] は ary[(s..e)] と動作する

キャプチャの表現

- [:capture, name, exp] で(名前付)キャプチャを表現
- name はシンボル
- exp は正規表現の抽象構文木
- Ruby 1.9 の (?<name>exp) に対応する

キャプチャ対応 try

- try(exp, seq, pos, md) {|pos2, md2| ... }
- 以前の try に md, md2 を追加
- md, md2 はキャプチャされた名前から範囲へのハッシュ
- 範囲は s...e という Range で表現
- md にはその try の呼び出しまでに行ったキャプチャの情報を渡す
- md2 は pos から pos2 までのマッチに含まれるキャプチャを md に加えたものになる

try の例

- try([:capture, :n, [:lit, "a"]],
  ["a"], 0, {}) {|pos, md|
  p pos #=> 1
  p md #=> {:n=>0...1}
  }

try の実装

- def try(exp, seq, pos, md, &block)
  case exp[0]
  when :empseq
    try_empseq(seq, pos, md, &block)
  when :lit
    _, sym = exp
    try_lit(seq, pos, md, &block)
  ...
  end
- 引数で渡された md を try_xxx にそのまま渡す

try_empseq

- def try_empseq(seq, pos, md)
  yield pos, md
  end
- 渡された md をそのまま引きわたす

try_lit

- def try_lit(sym, seq, pos, md)
  if pos < seq.length && seq[pos] == sym
  yield pos+1, md
  end
  end
- 渡された md をそのまま引きわたす

try_cat

- def try_cat(e1, e2, seq, pos, md, &block)
  try(e1, seq, pos, md) {|pos2, md2|
  try(e2, seq, pos2, md2, &block)
  }
  end
- md を try(e1) に渡し、md2 を try(e2) に渡す
- pos と同じ流れで渡していく

try_alt

- def try_alt(e1, e2, seq, pos, md, &block)
  try(e1, seq, pos, md, &block)
  try(e2, seq, pos, md, &block)
  end
- md を try(e1) と try(e2) に渡す
- pos と同じ流れで渡していく

try_rep

- def try_rep(exp, seq, pos, md, &block)
  try(exp, seq, pos, md) {|pos2, md2|
  try_rep(exp, seq, pos2, md2, &block) if pos < pos2
  }
  yield pos, md
  end
- pos と同じ流れで渡していく

他の try_xxx

- 同様に pos と同じ流れで渡していく

try の [:capture, n, e] 対応

- def try(exp, seq, pos, md, &block)
  case exp[0]
  ...
  when :capture
  _, n, e = exp
  try_capture(n, e, seq, pos, md, &block)
  ...
  end

try_capture

- def try_capture(n, e, seq, pos, md, &block)
  try(e, seq, pos, md) {|pos2, md2|
    md3 = md2.dup　　# ハッシュをコピー
    md3[n] = pos...pos2　# キャプチャ情報を格納
    yield pos2, md3
  }
  end
- e に対するマッチに成功したら情報を追加
- md2 自体は変更せず、コピーに追加

Hash#dup

- ハッシュのコピーをつくる
- h = {:a => 1}
  h2 = h.dup
  h2[:b] = 2
  p h2　　#=> {:b=>2, :a=>1}
  p h　　#=> {:a=>1}　元のハッシュはそのまま

try_capture の Hash#dup

- def try_capture(n, e, seq, pos, md, &block)
  try(e, seq, pos, md) {|pos2, md2|
  md3 = md2.dup
  md3[n] = pos...pos2
  yield pos2, md3
  }
  end

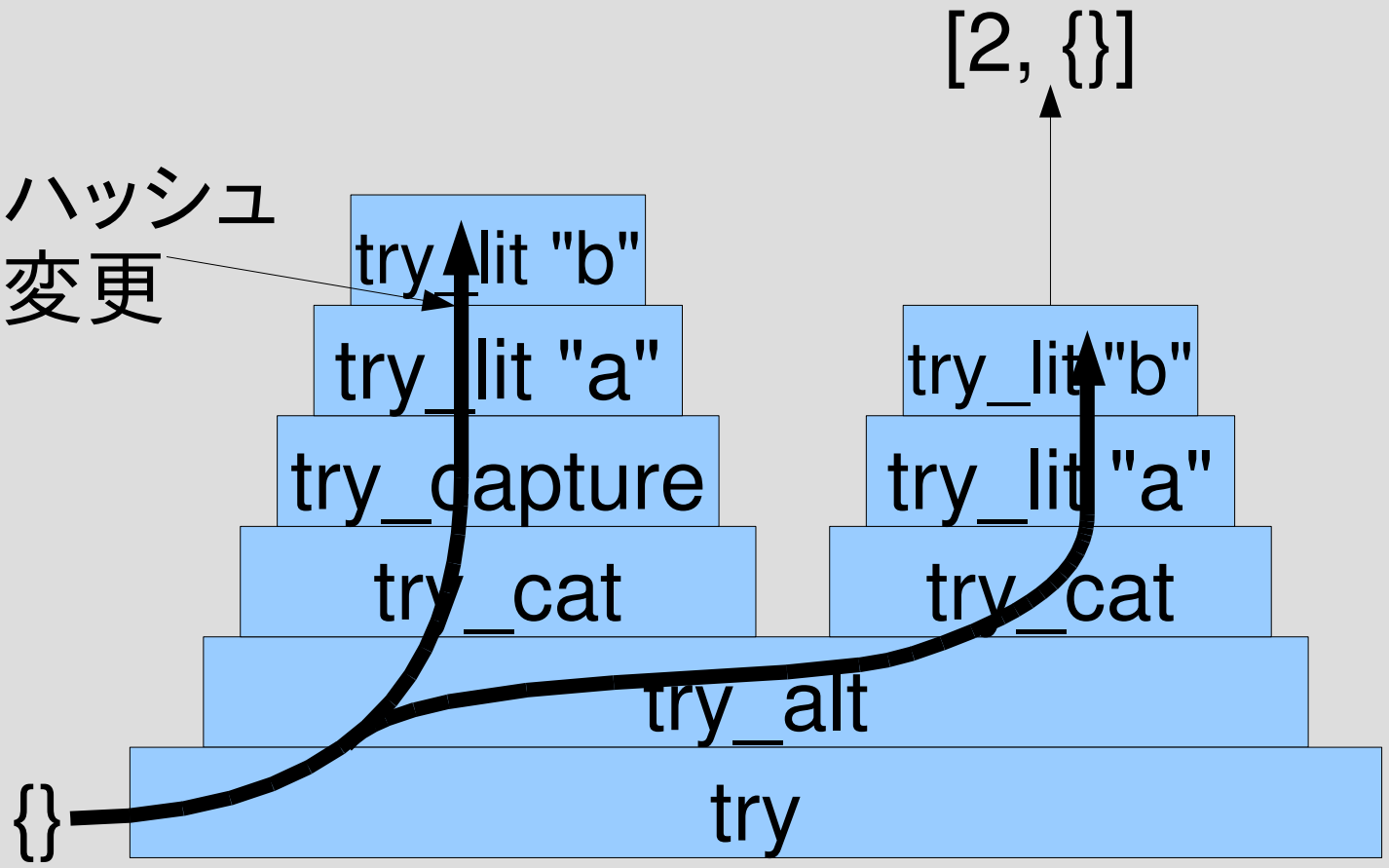
コピーの必要性

- コピーせずに変更すると呼び出し側に影響する
- キャプチャの追加の影響はマッチに成功した場合だけに限る
- /(?<n>a)b|ac/ =~ "ac"
- try(
  [:alt, [:cat, [:capture, :k, [:lit, "a"]], [:lit, "b"]],
  [:cat, [:lit, "a"], [:lit, "c"]]],
  %w[a c], 0, {}) {|pos, md| p [pos, md] }
  #=> [2, {}]
- /(?<n>a)b|ac/ で、ac の部分にマッチするキャプチャの所は通らない

コピーの必要性

コピーせずに共有してると
外に影響が出る

/(?<n>a)b|ac/ =~ "ac"

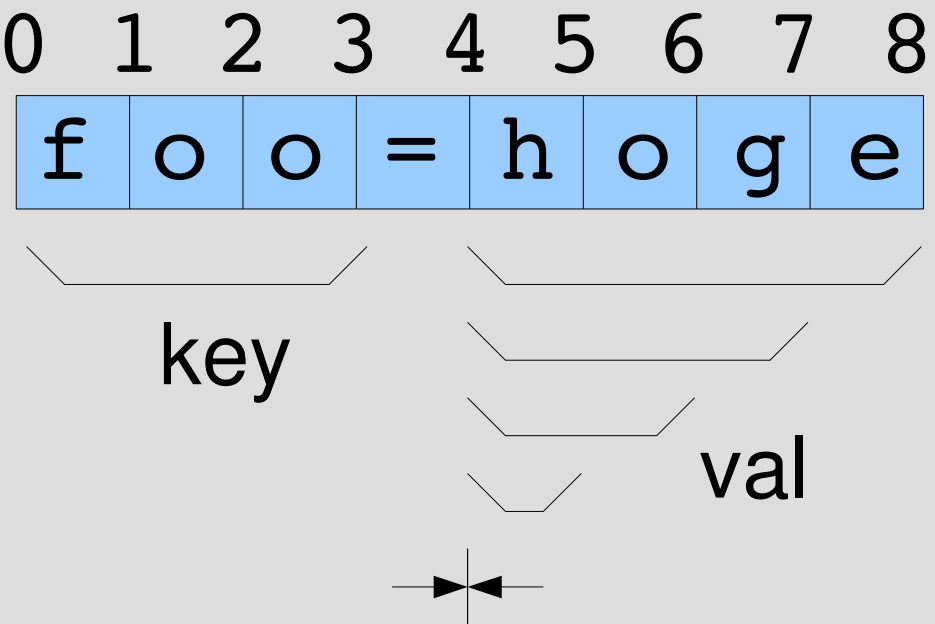


try_capture の実行

- try([:capture, :n, [:rep, [:lit, "a"]]],
  %w[a a b], 0, {}) {|pos, md|
  p [pos, md]
  }
  #=>
  [2, {:n=>0...2}]
  [1, {:n=>0...1}]
  [0, {:n=>0...0}]

try_capture の実行 (2)

- try([:cat, [:capture, :key, [:rep, [:anysym]]],
  [:cat, [:lit, "="],
  [:capture, :val, [:rep, [:anysym]]]]],
  %w[f o o = h o g e], 0, {}) {|pos, md|
  p [pos, md]
  }
  #=>
  [8, {:val=>4...8, :key=>0...3}]
  [7, {:val=>4...7, :key=>0...3}]
  [6, {:val=>4...6, :key=>0...3}]
  [5, {:val=>4...5, :key=>0...3}]
  [4, {:val=>4...4, :key=>0...3}]



find_match

- def find_match(ary, pat, beg=0)
  beg.upto(ary.length) {|s|
  try(pat, ary, s, {}) {|e, md| return [s, e, md] }
  }
  nil
  end
- 先頭 (もしくは beg) から探して見付かったら終了
- [マッチ先頭,マッチ終了,キャプチャ] を返す
- マッチしなかったら nil を返す

省略可能引数

- メソッドの定義で「仮引数=デフォルト式」と書く
- 省略可能引数は必須引数より後
- 実引数が省略された場合、デフォルト式が評価されて仮引数の値となる
- デフォルト式はメソッド内部の環境で評価され、左の引数も参照できる

```
def m(a1, a2, a3=10, a4=a1+a3)
  p [a1,a2,a3,a4]
end
m(1,2)        #=> [1,2,10,11]
m(1,2,3)      #=> [1,2,3,4]
```

find_match の省略可能引数

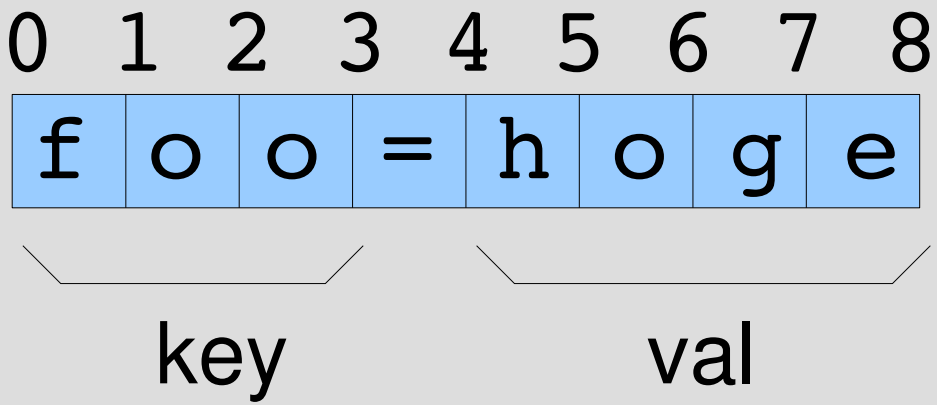
- find_match(a,e) と呼び出してもよい
- find_match(a,e,pos) と呼び出してもよい

```
def find_match(ary, pat, beg=0)
  beg.upto(ary.length) {|s|
    try(pat, ary, s, {}) {|e, md| return [s, e, md] }
  }
  nil
end
```

find_match(a,e) は
find_match(a,e,0) と同じ

find_match の実行

- seq = %w[f o o = h o g e]
  s,e,md = find_match(seq,
    [:cat, [:capture, :key, [:rep, [:anysym]]],
      [:cat, [:lit, "="],
        [:capture, :val, [:rep, [:anysym]]]]])
  p [s,e,md] #=> [0, 8, {:val=>4...8, :key=>0...3}]
  p seq[md[:key]] #=> ["f", "o", "o"]

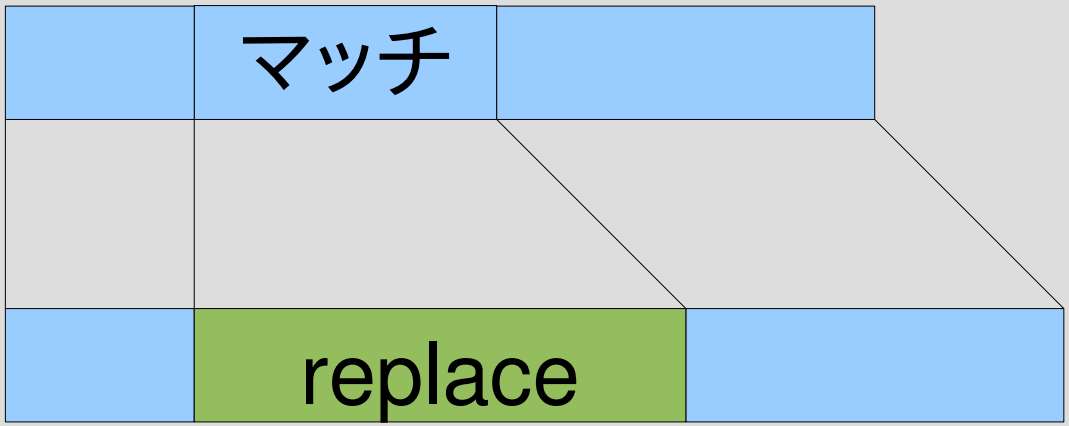


文字列置換

- 文字列中でマッチした部分を置き換える
  - String#sub　　最初にマッチしたのを置き換える
  - String#gsub　　ぜんぶ置き換える

String#sub

- 文字列の置換 (substitution)
  - str.sub(/pat/) { replace }
  - 文字列の一部を正規表現で指定する
  - マッチした最初の場所をブロックの結果で置き換える
  - 置き換えた結果を新しい文字列として返す (非破壊的)
- "abc".sub(/b/) { "z" }　#=> "azc"
- "abcabc".sub(/b/) { "XXX" }　#=> "aXXXcabc"



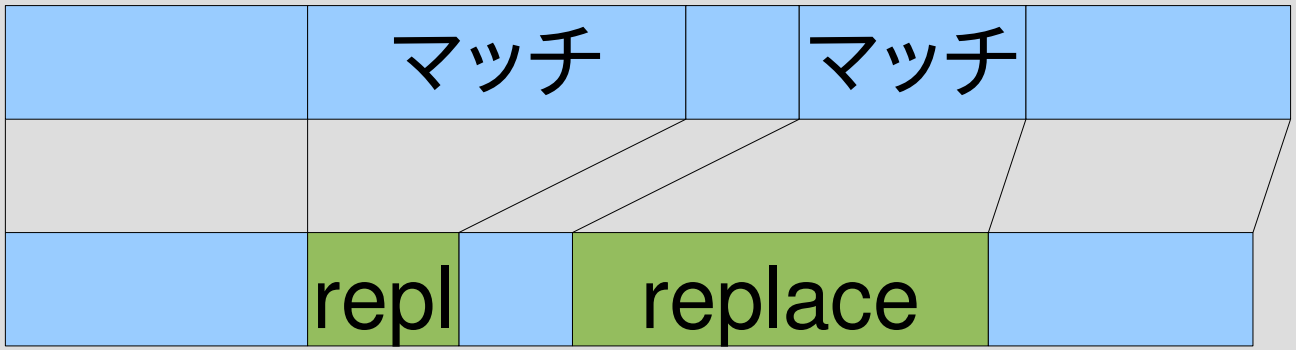
String#sub（続き)

- replace 内では $~ が使える ($& や $1, ... も)
- "abc".sub(/b/) { $& * 4 } #=> "abbbbz"
- "abc".sub(/b/) {
  "[#{$~.begin(0)}..#{$~.end(0)}]" }
  #=> "a[1..2]c"
- "this is a pen.".sub(/[a-z]+/) {
  $&.capitalize }
  #=> "This is a pen."

String#capitalize は先頭の一文字を大文字にする

String#gsub

- 文字列の置換 (global substitiution)
  - str.gsub(pat) { replace }
  - 文字列の一部を正規表現で指定する
  - マッチしたすべての場所をブロックの結果で置き換える
  - 置き換えた結果を新しい文字列として返す (非破壊的)
- "abc".gsub(/b/) { "z" }　　#=> "azc"
- "abcabc".gsub(/b/) { "XXX" }
  #=> "aXXXcaXXXc"



String#gsub (続き)

- $~ が使える (もちろん $& も)
- "this is a pen.".gsub(/[a-z]+/) { $&.capitalize }
  #=> "This Is A Pen."

正規表現エンジンで sub を実現

- 最初のマッチを置き換える:
  subst(str, pat) {|s, h| replace }
- ブロック引数には以下を渡す
  - マッチした部分の文字列 s
  - キャプチャした名前から文字列へのハッシュ h
- find_match を使って実現

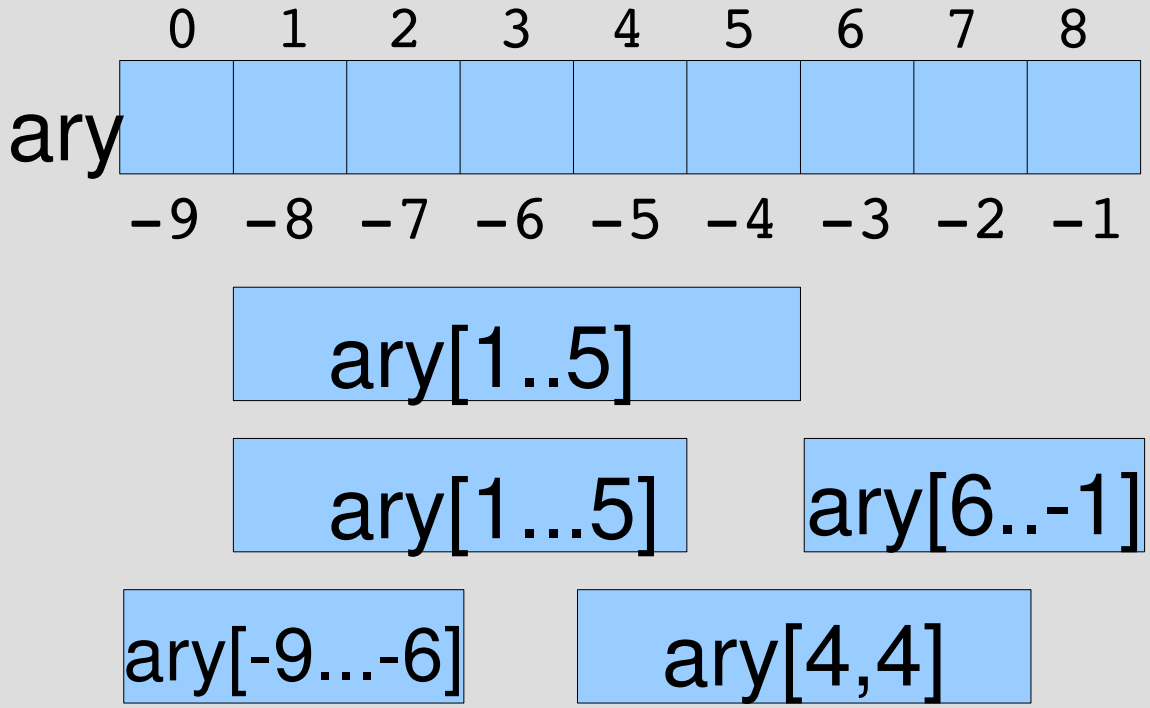
```
subst("abcabc", [:lit, "b"]) {|s, h| "[#{s}]" }
#=> "a[b]cabc"
```

subst

```
def subst(str, pat)
  ary = str.split(//)
  r = find_match(ary, pat)
  return str if !r
  s, e, md = r
  h = {}
  md.each {|k, r| h[k] = ary[r].join }
  ary[0...s].join +
    yield(ary[s...e].join, h) +
    ary[e..-1].join
end
```

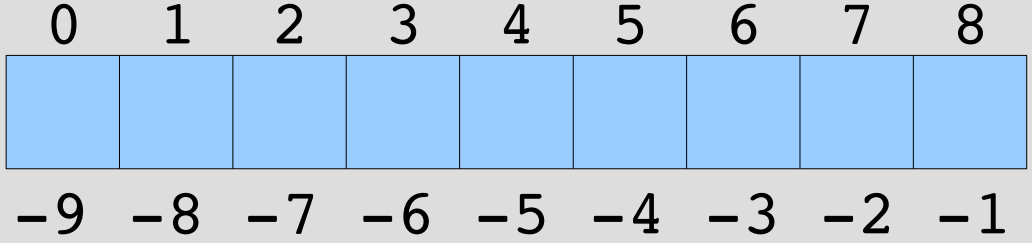
部分配列

- ary[pos1..pos2]　pos1からpos2までの部分配列
- ary[pos1...pos2]　pos1からpos2。pos2 は含まない
- ary[pos,len]　posから長さlen



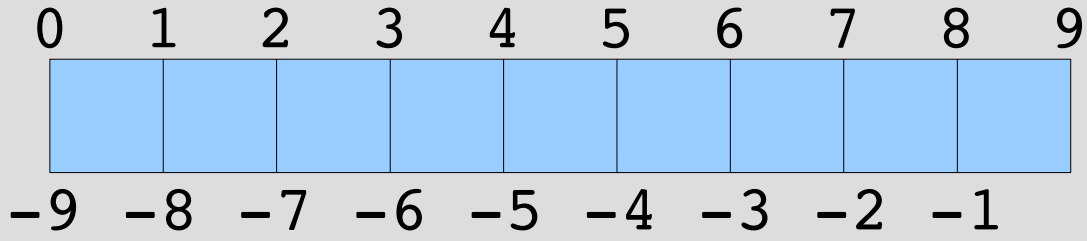
要素を指すか、間を指すか

要素を指す



- a[s..e] に適切

間を指す

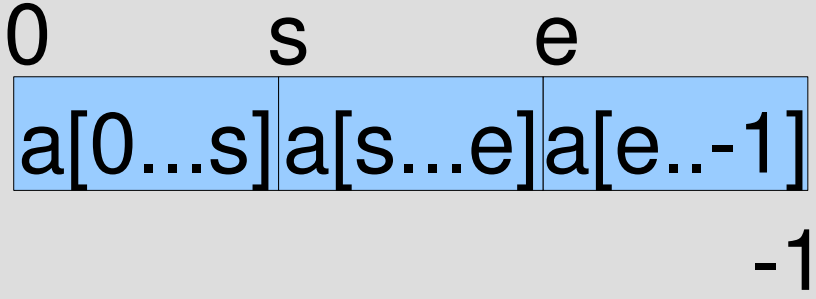


- a[s...e] に適切
- 負で最後を指せない

単に考え方の問題
つじつまが合えば、どちらで考えても良い

substの部分配列

- ary[0...s] がマッチより前
- ary[s...e] がマッチした部分
- ary[e..-1] がマッチより後




```
r = find_match(ary, pat)
return str if !r
s, e, md = r
h = {}
md.each {|k, r| h[k] = ary[r].join }
ary[0...s].join +
  yield(ary[s...e].join, h) +
 ary[e..-1].join
```

Array#join

- 配列の要素を連結して文字列として返す
- ["a", "b", "c"].join #=> "abc"

substのArray#join

- ary[0...s].join がマッチより前
- ary[s...e].join がマッチした部分
- ary[e..-1].join がマッチより後

```
r = find_match(ary, pat)
return str if !r
s, e, md = r
h = {}
md.each {|k, r| h[k] = ary[r].join }
ary[0...s].join +
  yield(ary[s...e].join, h) +
  ary[e..-1].join
```

Hash#each

- 鍵と値のペアそれぞれに対する繰り返し
- hash.each {|k, v| ... }
- h = {"one"=>1, "two"=>2}
  h.each {|k, v| p [k, v] }
  #=>
  ["two", 2]
  ["one", 1]

- 順序は不定

subst の Hash#each

- def subst(str, pat)
  ary = str.split(//)
  r = find_match(ary, pat)
  return str if !r
  s, e, md = r
  h = {}
  md.each {|k, r| h[k] = ary[r].join }
  ary[0...s].join +
  yield(ary[s...e].join, h) +
  ary[e..-1].join
  end

{:key=>0...3} から
{:key=>"foo"} を生成

ブロックに h を渡す

subst の例 (キャプチャ不使用)

- subst("abcde", [:cat, [:lit, "b"], [:lit, "c"]]) {
  "X" }
  #=> "aXde"

- subst("abcde", [:anysym]) { "X" }
  #=> "Xbcde"

- subst("abc", [:empseq]) { "X" }
  #=> "Xabc"

subst の例 (キャプチャ使用)

- p subst("foo=hoge",
  [:cat, [:capture, :key, [:rep, [:anysym]]],
  [:cat, [:lit, "="],
  [:capture, :val, [:rep, [:anysym]]]]]) {|s, h|
  "#{h[:key]}=#{h[:val].reverse}"
  }
  #=> "foo=egoh"

レポート

- 文字列中にダブルクォートで括られた部分があるとして、その括りをシングルクォートに変えるメソッド dq2sq を実装せよ
- dq2sq(str)
- 実装したらユニットテストで確認してほしい
- ∠切 2007-07-24 16:20
- HIPLUS
- 拡張子が txt なテキストファイルがよい

動作例

- dq2sq('abc"def"ghi')　#=> "abc'def'ghi"
- dq2sq('a"b"c"d"e')　#=> "a'b'c¥"d¥"e"
- dq2sq('abc')　#=> "abc"

ヒント

- subst を一回使う
- ダブルクォートで括ってあるものがふたつ以上あっても最初のひとつだけ変える
- ダブルクォートで括ってあるものがひとつもなければなにも変えずに返す

まとめ

- 前回のレポートの説明
- キャプチャの説明
- MatchData
- キャプチャの実装
- 文字列置換: subst
- レポート

- 次回:
  - ルックアラウンドアサーション
  - 試験のこと