テキスト処理 第11回 (2006-07-04)

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/

今日の内容

- 前回のレポートの説明
- MatchData
- String#sub
- String#gsub
- キャプチャ
- レポート

MatchData (1)

- 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

MatchData (2)

- p /b+/ =~ “abbccc”　#=> 1
  m = $~
  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], $~.begin(0), $~.end(0) の「0」の意味は後述

特殊変数 $&

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

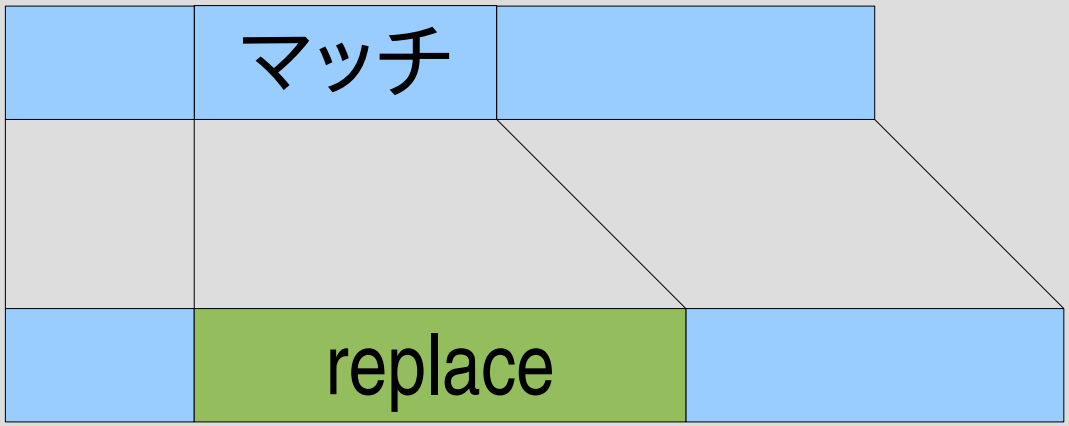
  p $& #=> "bb"

  $~ = nil

  p $& #=> nil

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"

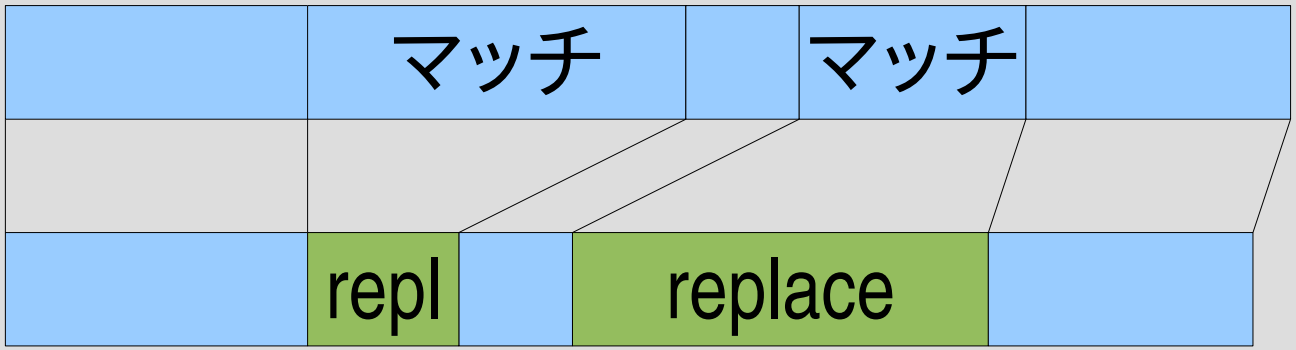


String#sub (cont.)

- replace 内では $~ が使える ($& 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#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 (cont.)

- $~ 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

- マッチの範囲を見つける: find_match(ary, pat, beg=0)
- 最初のマッチを置き換える: subst(str, pat) { replace }

```
subst("abcabc", [:lit, "b"]) { "X" }    #=> "aXcabc"
```

find_match(ary, pat, beg=0)

- hasmatch に類似
- 返り値が違う:
  マッチした場合、[開始位置, 終了位置] という配列を返す

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

省略可能引数

- 「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| return [s, e] }
  }
  nil
end
```

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

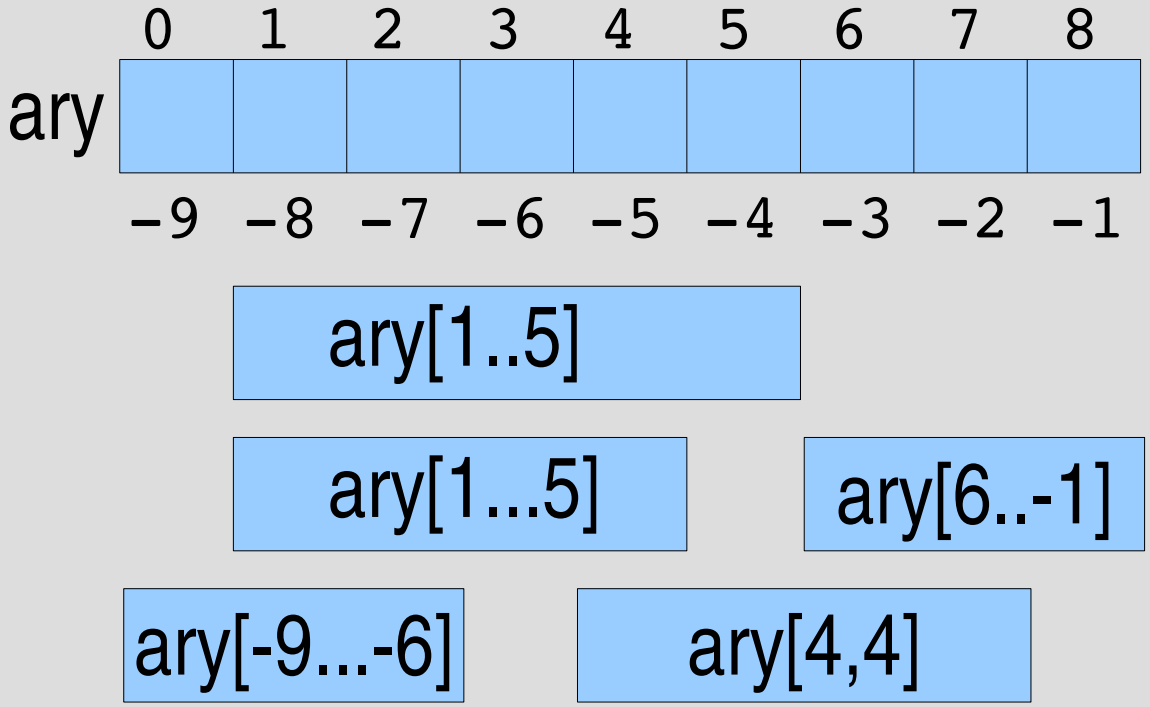
subst(str, pat) { replace }

- マッチが見つかったらその部分を置き換えた文字列を生成する

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

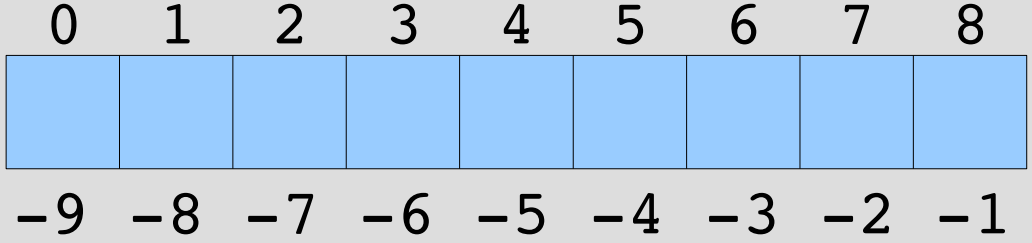
部分配列

- 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



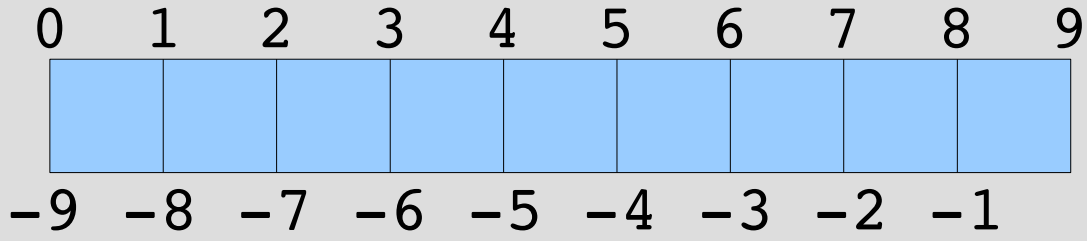
要素を指すか、間を指すか

要素を指す



- a[s..e] に適切

間を指す

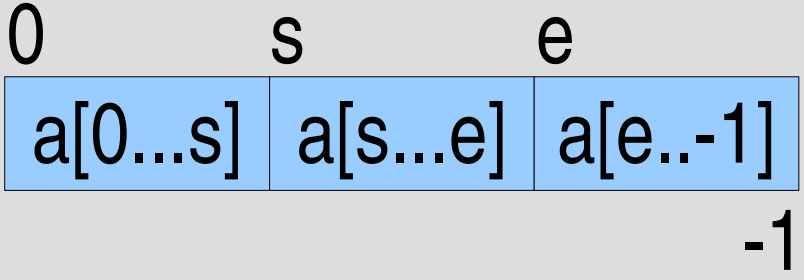


- a[s...e] に適切
- 負で最後を指せない

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




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

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

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

正規表現エンジンで **gsub** を実現

- gsubst(str, pat) { replace }

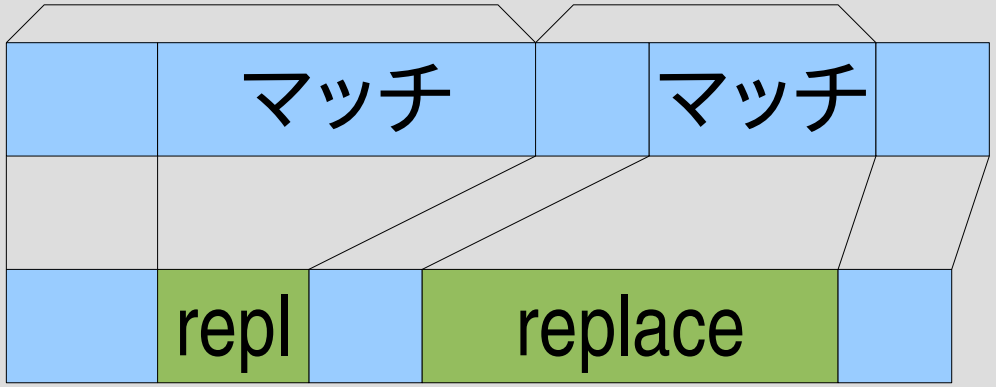
  gsubst("abcabc", [:lit, "b"]) { "X" } #=> "aXcaXc"

gsubst(str, pat) { replace }

```
def gsubst(str, pat)
  ary = str.split(//)
  i = 0
  result = []
  while i <= ary.length &&
        (r = find_match(ary, pat, i))
    s, e = r
    result.concat ary[i...s]
    result << yield(ary[s...e].join)
    if s == e
      result << ary[e]
      e += 1
    end
    i = e
  end
  if i < ary.length
    result.concat ary[i..-1]
  end
  result.join
end
```

gsubst の構造

ループ一回り




```
def gsubst(str, pat)
  初期化
  while マッチするか?
    マッチより前を結果へ
    マッチ自体を置換して結果へ
    次のマッチを試す場所まで進める
  end
  残りを結果へ
  結果を文字列にまとめて返す
end
```

Array への破壊的追加

- ary << obj　　要素をひとつ追加
- ary.concat ary2　　配列の中の要素をすべて追加

- a = []
  a << 1　　# [1] になる
  a.concat [2,3]　　# [1,2,3] になる
  a << [4]　　# [1,2,3,[4]] になる

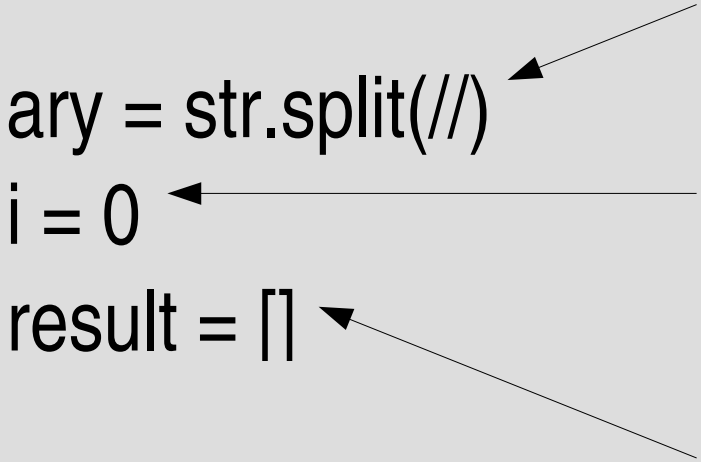
gsubst の <<, concat

```
def gsubst(str, pat)
  ary = str.split(//)
  i = 0
  result = []
  while i <= ary.length &&
        (r = find_match(ary, pat, i))
    s, e = r
    result.concat ary[i...s]
    result << yield(ary[s...e].join)
    if s == e
      result << ary[e]
      e += 1
    end
    i = e
  end
  if i < ary.length
    result.concat ary[i..-1]
  end
  result.join
end
```

gsubst の初期化

```
def gsubst(str, pat)
  ary = str.split(//)
  i = 0
  result = []
  while i <= ary.length &&
        (r = find_match(ary, pat, i))
    s, e = r
    result.concat ary[i...s]
    result << yield(ary[s...e].join)
    if s == e
      result << ary[e]
      e += 1
    end
    i = e
  end
  if i < ary.length
    result.concat ary[i..-1]
  end
  result.join
end
```

gsubst の初期化

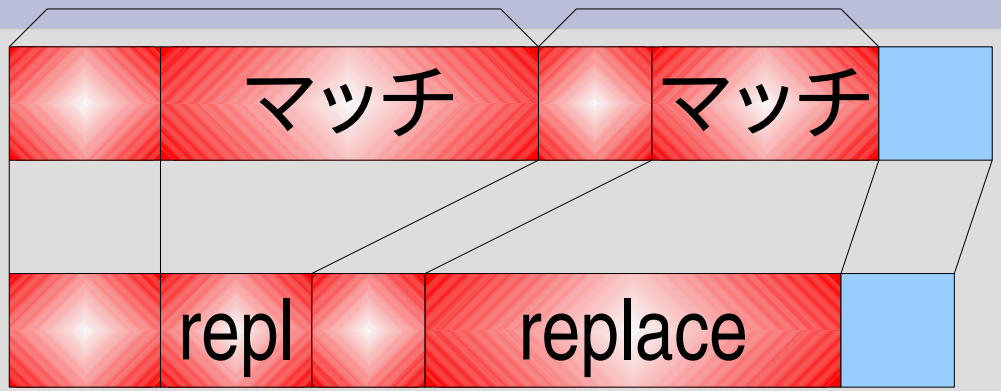


- 受け取った文字列を配列に変換
- どこまで処理したかを示すインデックス
  要素の間を指すイメージ
- 結果を入れておく配列

gsubst の置換

```
def gsubst(str, pat)
  ary = str.split(//)
  i = 0
  result = []
  while i <= ary.length &&
        (r = find_match(ary, pat, i))
    s, e = r
    result.concat ary[i...s]
    result << yield(ary[s...e].join)
    if s == e
      result << ary[e]
      e += 1
    end
    i = e
  end
  if i < ary.length
    result.concat ary[i..-1]
  end
  result.join
end
```

gsubst の置換




```
while i <= ary.length &&
      (r = find_match(ary, pat, i))
  s, e = r
  result.concat ary[i...s]
```

次のマッチを試す
場所まで進める
end

- 空文字列が見つかるかもしれないから
  i == ary.length も許す
- find_match で試す
- マッチ以前を結果へ
- マッチ自体を文字列にしてブロックに渡して結果へ

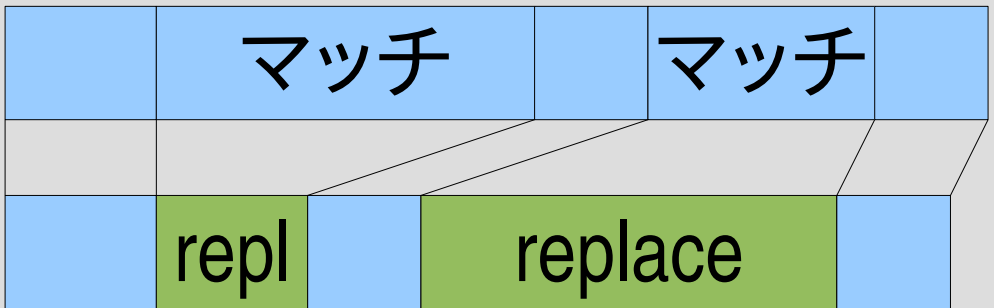
次のマッチを試す所まで進める

```
def gsubst(str, pat)
  ary = str.split(//)
  i = 0
  result = []
  while i <= ary.length &&
        (r = find_match(ary, pat, i))
    s, e = r
    result.concat ary[i...s]
    result << yield(ary[s...e].join)
    if s == e
      result << ary[e]
      e += 1
    end
    i = e
  end
  if i < ary.length
    result.concat ary[i..-1]
  end
  result.join
end
```

次のマッチを試す所まで進める

- マッチの終わりは e なので i = e が基本
- でも、空文字列にマッチしていたら、無限ループになってしまうので無理矢理一文字進める
  - 空文字列にマッチした判断
  - 一文字を結果へ
  - 一文字 e を進める

```
if s == e
  result << ary[e]
  e += 1
end
i = e
```

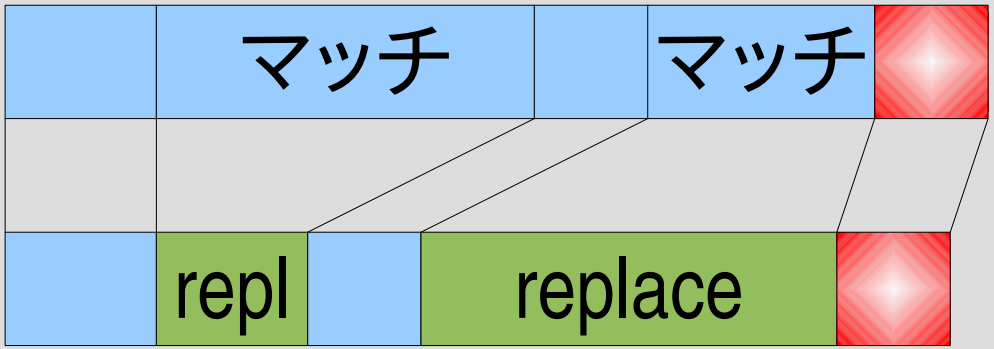


残りを結果へ

```
def gsubst(str, pat)
  ary = str.split(//)
  i = 0
  result = []
  while i <= ary.length &&
        (r = find_match(ary, pat, i))
    s, e = r
    result.concat ary[i...s]
    result << yield(ary[s...e].join)
    if s == e
      result << ary[e]
      e += 1
    end
    i = e
  end
  if i < ary.length
    result.concat ary[i..-1]
  end
  result.join
end
```

残りを結果へ

- 一文字以上残っていたら、と言う判断
  - i は最大 ary.length + 1 になり、そのとき ary[i..-1] は nil
  - result.concat nil はエラーになるので、避ける
- 残りを結果へ




```
if i < ary.length
  result.concat ary[i..-1]
```

結果を文字列にして返す

```
def gsubst(str, pat)
  ary = str.split(//)
  i = 0
  result = []
  while i <= ary.length &&
        (r = find_match(ary, pat, i))
    s, e = r
    result.concat ary[i...s]
    result << yield(ary[s...e].join)
    if s == e
      result << ary[e]
      e += 1
    end
    i = e
  end
  if i < ary.length
    result.concat ary[i..-1]
  end
  result.join
end
```

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

```
/(a*)(b*)(c*)/ =~ "abbcc"
p $1                    #=> "a"
p $2                    #=> "bb"
p $3                    #=> "ccc"
```

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
- /(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、左括弧の順番で決まる
- 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"
```

MatchData

- $1, $2, $3, ... は $~[1], $~[2], $~[3], ... に対応する
- $~.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
```

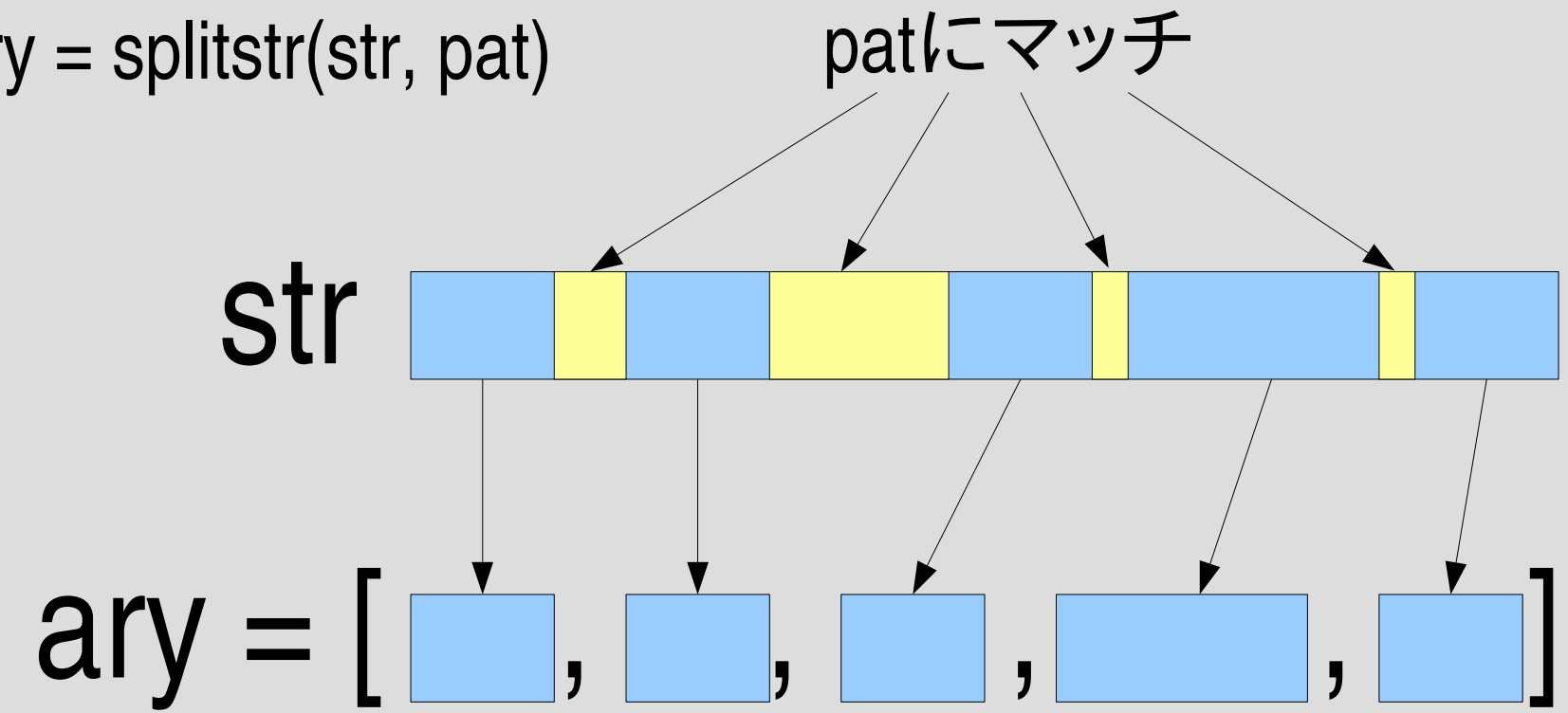
- $~[0], $~.begin(0) の「0」はマッチ全体を表している
- 注意: マッチした部分全体は $& で、$0 ではない

レポート

- splitstr(str, pat) を実装し解説せよ
- splitstr(str, pat) は str を正規表現抽象構文木 pat がマッチする部分で分割し、マッチしない部分を要素とする配列を生成して返す
- 実装したらユニットテストで確認してほしい
- ユニットテストでも不明な挙動があれば String#split で確認する
- ∠切 2006-07-11 16:20
- IT's class
- 拡張子が txt なテキストファイルがよい

レポート

- ary = splitstr(str, pat)



- 注意: マッチの終端の空文字列にマッチした場合、その間の空文字列は配列の要素にならない

レポートのヒント

- ヒント1: find_match を使う
- ヒント2: 今回は再帰不要
- ヒント3: まず空文字列のことを気にせずに書いてみる

まとめ

- 前回のレポートの説明
- MatchData
- String#sub
  - 解説、実装
- String#gsub
  - 解説、実装
- キャプチャ
  - 解説
  - 実装するのは次回
- レポート