テキスト処理 第11回 (2008-07-01)

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

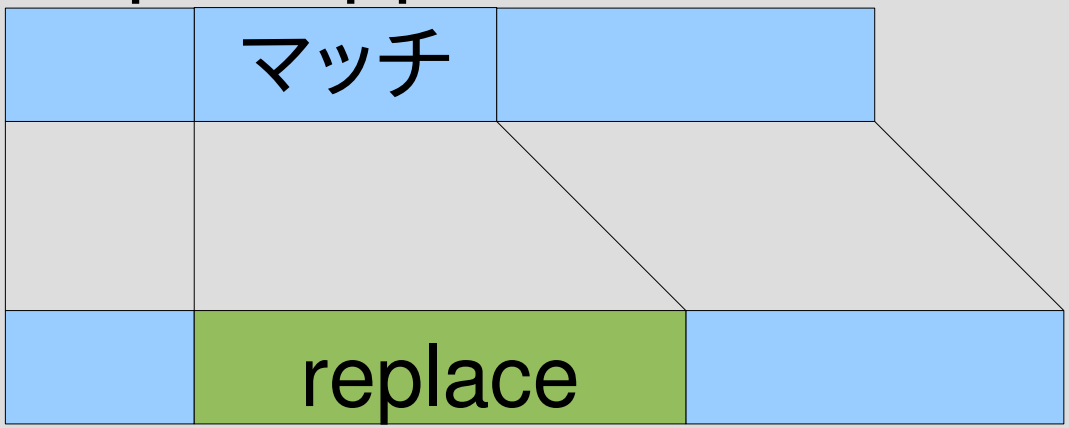
- 前回のレポートの説明
- 文字列置換
- 正規表現の拡張: ¥G
- レポート

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　　すべて置き換える

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)
- "hot".sub(/o/) { "a" }　#=> "hat"
- "prune".sub(/run/) { "ineappl" }
  #=> "pineapple"



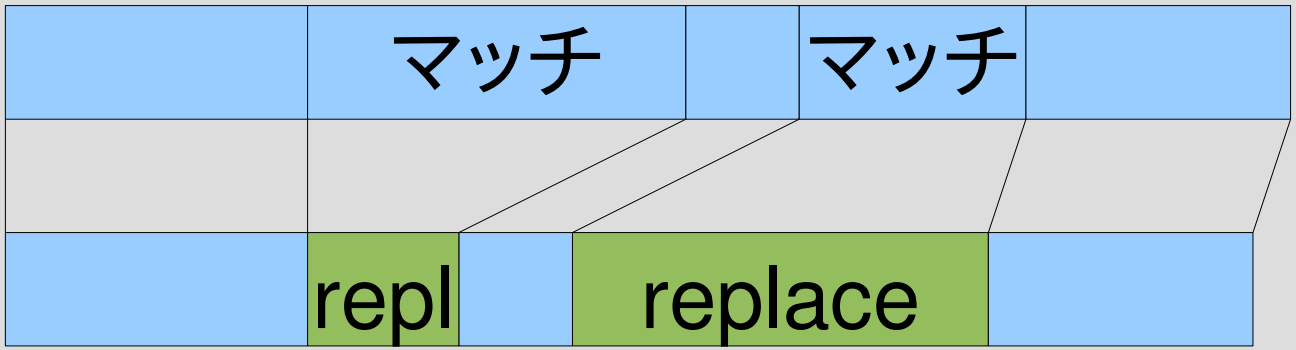
String#sub（続き)

- replace 内では $~ が使える ($& や $1, ... も)
- "hot".sub(/o/) { $& * 4 } #=> "hoooot"
- "hot".sub(/o/) {
    "[#{$~.begin(0)}...#{$~.end(0)}]" }
  #=> "h[1...2]t"
- "this is a pen.".sub(/[a-z]+/) { $&.upcase }
  #=> "THIS is a pen."

String#upcase は大文字に変換した文字列を返す

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)
- "hat".gsub(/a/) { "o" }　　#=> "hot"
- "hat".gsub(/[ht]/) { "X" }　　#=> "XaX"



String#gsub (続き)

- $~ が使える (もちろん $& も)
- "this is a pen.".gsub(/[a-z]+/) { $&.upcase }
  #=> "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
  - キャプチャのハッシュ h
    (位置じゃなくて文字列自体)
- cap_include を使って実現

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

subst の例 (キャプチャ不使用)

- subst("abcde", [:cat, "b", "c"]) { "X" }
  #=> "aXde"
- subst("abcde", [:anychar]) { "X" }
  #=> "Xbcde"
- subst("abc", [:empstr]) { "X" }
  #=> "Xabc"

subst の例 (キャプチャ使用)

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

h[:key] は "foo"

h[:val] は "hoge"

subst の例

- p subst("melon", [:string_start]) {|s, h| "water" }
  #=> "watermelon"

- p subst("watermelon",
  [:cat, "m",
  [:capture, :c, [:anychar]],
  "l"]) {|s, h|
  "l" + h[:c] + "m"
  }
  #=> "waterlemon"

h[:c] は "e" になる
(4...5 ではなく)

subst の実装方針

- cap_include を文字列のところに配列もうけとるようにする
- cap_include でマッチに挑戦
- マッチしなかったら、
  あたえられた文字列をそのまま返す
- マッチしたら、
  ブロックを読んで置換文字列を取得
- マッチ部分の左右を置換文字列の左右に連結して返す

cap_include 改

```
def cap_include(r, str)
  str = str.split(//) if str.respond_to? :to_str
  0.upto(str.length) {|b|
    try(r, str, b, {}) {|e, md|
      md = md.dup
      md[:all] = b...e
      return md
    }
  }
  nil
end
```

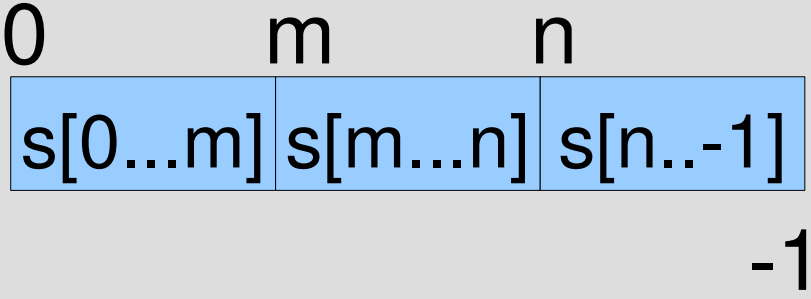
- 文字列じゃなかったらsplit しないでそのまま使う
- 配列を与えればそのまま動く
- 文字列だったら今までどおりの動作

subst

```
def subst(s, r)
  s = s.split(//)
  md = cap_include(r, s)
  return s.join if !md
  h = {}
  md.each {|k, v| h[k] = s[v].join }
  m = md[:all].begin
  n = md[:all].end
  s[0...m].join +
    yield(s[m...n].join, h) +
    s[n..-1].join
end
```

substの部分配列

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




```
m = md[:all].begin
n = md[:all].end
s[0...m].join +
  yield(s[m...n].join, h) +
  s[n..-1].join
```

Array#join

- 配列の要素を連結して文字列として返す
- ["m", "e", "l", "o", "n"].join #=> "melon"

substのArray#join

- s[v] はキャプチャした文字列
- s[0...s].join がマッチより前
- s[s...e].join がマッチした部分
- s[e..-1].join がマッチより後

```
md.each {|k, v| h[k] = s[v].join }
m = md[:all].begin
n = md[:all].end
s[0...m].join +
  yield(s[m...n].join, h) +
  s[n..-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(s, r)
  s = s.split(//)
  md = cap_include(r, s)
  return s if !md
  h = {}
  md.each {|k, v| h[k] = s[v].join }
  m = md[:all].begin
  n = md[:all].end
  s[0...m].join +
    yield(s[m...n].join, h) +
    s[n..-1].join
end
```

{:key=>0...3} から
{:key=>"foo"} を生成

ブロックに h を渡す

正規表現エンジンで gsub を実現

- すべてのマッチを置き換える:
  gsubst(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
  - キャプチャのハッシュ h
- cap_include を使って実現

```
gsubst("abcabc", "b") {|s, h| "[#{s}]" }
#=> "a[b]ca[b]c"

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

gsubst の例

- p gsubst("watermelon", "e") {|s, h| s*2 }
  #=> "wateermeelon"

gsubst の実装方針

- cap_include で開始位置を指定できるようにする
- 開始位置 0 からマッチを探す
- 見つかったらブロックを呼んで置換文字列を得る
- マッチの終端を開始位置として見つからなくなるまで繰り返す

cap_include 改2

```
def cap_include(r, str, beg=0)
  str = str.split(//) if str.respond_to? :to_str
  beg.upto(str.length) {|b|
    try(r, str, b, {}) {|e, md|
      md = md.dup
      md[:all] = b...e
      return md
    }
  }
  nil
end
```

- 省略可能引数begを加える
- 省略時の値は 0
- 0 からでなく beg から探す
- 省略したときの動作は今までどおり

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]
```

gsubst

```
def gsubst(s, r)
  s = s.split(//)
  beg = 0
  result = []
  while md = cap_include(r, s, beg)
    m = md[:all].begin; n = md[:all].end
    h = {}; md.each {|k, v| h[k] = s[v].join }
    result += s[beg...m]
    result << yield(s[m...n].join, h)
    beg = n
  end
  result += s[beg..-1]
  result.join
end
```

初期化

ループ

後始末

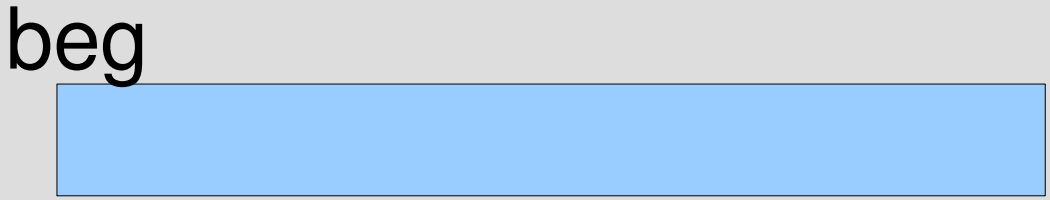
gsubst: 初期化

```
def gsubst(s, r)
  s = s.split(//)
  beg = 0
  result = []
```

s を文字列から配列へ

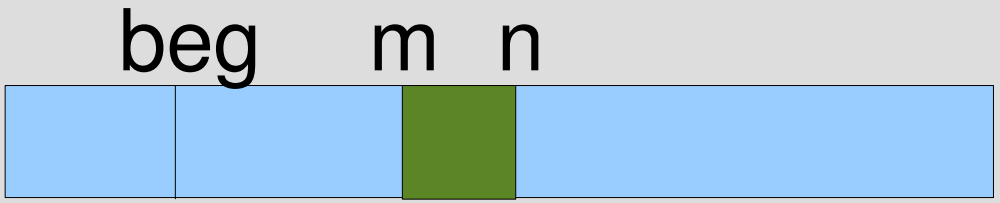
beg は文字列の左端

result は空



gsubst: ループ

```
while md = cap_include(r, s, beg) マッチする間ループ
  m = md[:all].begin
  n = md[:all].end
  h = {}                    subst と同様に h を生成
  md.each {|k, v| h[k] = s[v].join }
  result += s[beg...m]        マッチ前を result に追加
  result << yield(s[m...n].join, h) マッチした部分を yield
  beg = n                   次の開始位置はマッチ終端
end
```

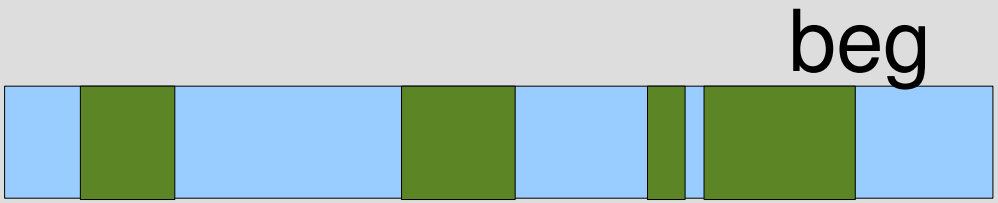


gsubst: 後始末

```
  result += s[beg..-1]
  result.join
end
```

最後のマッチの後を追加

文字列にして返す



gsubst の無限ループ

- gsubst("orange", [:empstr]) {|s, h| "" }
  終了しない
- gsubst("apple¥nbanana¥n",
  [:line_start]) {|s, h| "> " }
  終了しない

- beg が 0 のとき、
  cap_include(r, s, beg) は 0...0 でマッチ
  マッチ終端 n は 0 なので
  beg = n で beg が進まない

無限ループ防止

- ループ内で、
  空文字列にマッチしたら、
  次はその終端から一文字
  進めた場所から始める
- 文字列終端で進められな
  ければ、ループを終了する
- 進めたときはスキップした
  文字を result に追加して
  おく

```
beg = n
if m == n
  if beg == s.length
    break
  else
    result << s[m]
    beg += 1
  end
end
```

無限ループを防止した例

- p gsubst("orange", [:empstr]) {|s, h| "|" }
  #=> "|o|r|a|n|g|e|"
- p gsubst2("apple¥nbanana¥n",
  [:line_start]) {|s, h| "> " }
  #=> "> apple¥n> banana¥n"

¥G

- 探索を始めた位置にマッチする
- ¥A などと同様に ¥G 自身は文字を消費しない
- String#gsub で利用可能
- マッチとマッチの間に空きができないようにさせられる

- 例
- p "aardvark".gsub(/¥Ga/) { "A" }
  #=> "AArdvark"
- p "aardvark".gsub(/a/) { "A" }
  #=> "AArdvArk"

¥G の実装方針

- 配列表現は [:search_start]
- cap_include で md に探索を始めた位置をいれておく
- :search_start の処理では現在位置とその位置を比較

cap_include 改3

```
def cap_include(r, str, beg=0)
  str = str.split(//) if str.respond_to? :to_str
  md0 = { :search_start => beg...beg }
  beg.upto(str.length) {|b|
    try(r, str, b, md0) {|e, md|
      md = md.dup; md[:all] = b...e
      return md
    }
  }
  nil
end
```

:search_start の処理

```
when :search_start
  yield pos, md if pos == md[:search_start].begin
```

実行例

- p gsubst("aardvark",
  [:cat, [:search_start], "a"]) {|s, h| "A" }
  #=> "AArdvark"

無限ループ防止と ¥G

- 無限ループ防止により、1文字スキップすることがある
- その場合、¥G を使っても空きが出来る

- p gsubst("aardvark",
  [:cat, [:search_start], [:opt, "a"]]) {|s, h| "A" }
  #=> "AAArAdAvAArAkA"

- p "aardvark".gsub(/¥Ga?/) { "A" }
  #=> "AAArAdAvAArAkA"

¥G と前回のマッチ終端

- ¥G は前回のマッチ終端にマッチすると説明されることがある
- Ruby では、1文字スキップを考慮するとこれは正しくない

- ただし、正規表現エンジンによっては正しく前回のマッチ終端にマッチするものもある
- Perl はそう振る舞う
  % perl -e '$_ = "aardvark"; s/¥Ga?/A/g; print $_, "¥n"'
  AAArdvark
- このようにするには md0 を gsubst で用意する

レポート

- scanstr(s, r) を実装して解説せよ

- 実装したらユニットテストで確認してほしい
- 〆切 2008-07-08 12:00
- RENANDI
- 拡張子が txt なテキストファイルがよい

scanstr(s, r)

- String#scan を部分的に真似したもの
- 文字列 s 中の r にマッチする部分文字列をすべて調べ、配列として返す

- 実行例
- p scanstr("banana", [:anychar])
  #=> ["b", "a", "n", "a", "n", "a"]
- p scanstr("banana", [:alt, "b", "n"])
  #=> ["b", "n", "n"]
- p scanstr("banana", [:cat, "n", "a"])
  #=> ["na", "na"]

まとめ

- 前回のレポートの説明
- 文字列置換
  - subst
  - gsubst
- 探索開始位置 ¥G
- レポート