テキスト処理 第12回 (2006-07-11)

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
- キャプチャの説明
- キャプチャの実装
- 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*)/ =~ "abbcc"   #=> 0
p $1                          #=> "a"
p $2                          #=> "bb"
p $3                          #=> "ccc"
p /(.*)=(.*)/ =~ "foo=bar"    #=> 0
p [$1, $2]                    #=> ["foo", "bar"]
```

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/

```
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

shy group

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

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

キャプチャの実装

- 番号を数えるのは面倒なので 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
- 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

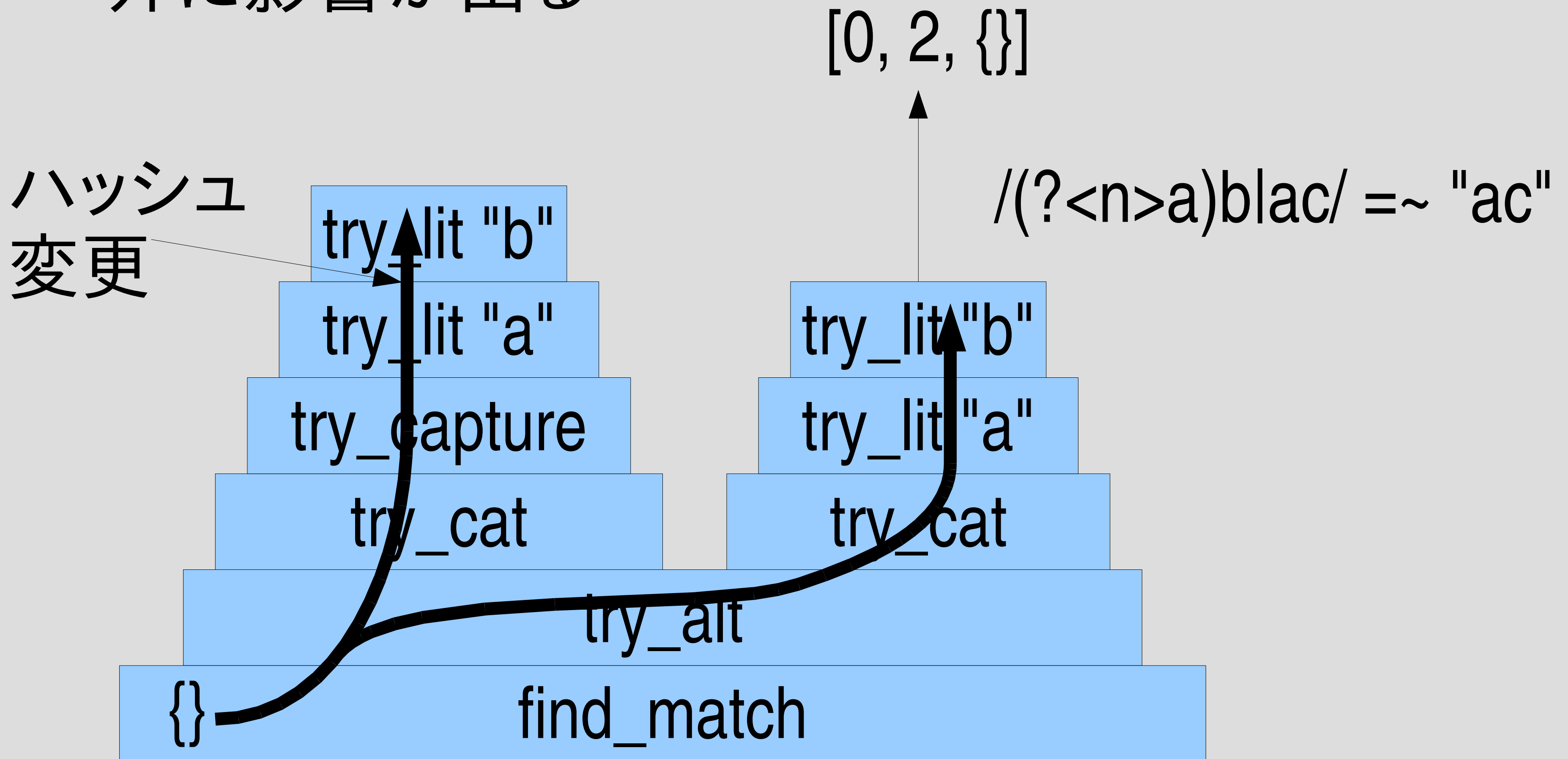
```
def try_capture(n, e, seq, pos, md, &block)
  try(e, seq, pos, md) {|pos2, md2|
    md3 = md2.dup
    md3[n] = pos...pos2
    yield pos2, md3
  }
end
```

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
-
  ```
  s, e, md = find_match(["a","c"],
    [:alt, [:cat, [:capture, :k, [:lit, "a"]], [:lit, "b"]],
           [:cat, [:lit, "a"], [:lit, "c"]]])
  p [s,e,md]    #=> [0, 2, {}]
  ```
- /(?<n>a)blac/ で、ac にマッチして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



try_capture の実行

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

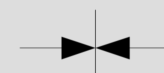
try_capture の実行 (2)

```
try([:cat, [:capture, :key, [:rep, [:anysym]]],
        [:cat, [:lit, "="],
          [:capture, :val, [:rep, [:anysym]]]]],
      ["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}]
```

0	1	2	3	4	5	6	7	8
f	o	o	=	h	o	g	e	

key

val



キャプチャ対応 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
  ```
- try に md として空ハッシュ {} を与える
- try から渡された md を返り値に加える
  ```

find_match の実行

```
seq = ["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"]
```

```
0   1   2   3   4   5   6   7   8
| f | o | o | = | h | o | g | e |
  \____key____/   \_____val_____/
```

キャプチャ対応 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
```

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]
- 順番は不定

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 も渡す

subst の実行

- ```
  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

- gsubst をキャプチャに対応させよ
- 実装したらユニットテストで確認してほしい
- 〆切 2006-07-18 16:20
- IT's class
- 拡張子が txt なテキストファイルがよい

レポートのヒント

- subst のキャプチャ対応に類似

まとめ

- 前回のレポートの説明
- キャプチャの説明
- キャプチャの実装
- レポート: gsubst のキャプチャ対応

- 次回:
  - backreference (\1, \2, ...)
  - 合成数(の長さの文字列)に対するマッチ
  - 試験のこと