テキスト処理 第7回 (2008-06-03)

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

- Proc (lambda)
- ブロック
- レポート

目標

- ブロック自体の受渡し
  - try_cat(re, str, pos) {|pos2| yield pos2 }
    の {|pos2| yield pos2 } をもっと短く書く
  - {|pos2| yield pos2 } は pos2 を受け取ってそのまま
    yield するという中継
  - 中継せず直接 yield する
- ブロックからの脱出
  - return
  - break

try における {|pos2| yield pos2 }

```
def try(re, str, pos)
  ...
    try_cat(re, str, pos) {|pos2| yield pos2 }
  ...
    try_alt(re, str, pos) {|pos2| yield pos2 }
  ...
    try_rep(re, str, pos) {|pos2| yield pos2 }
  ...
end
```

{|pos2| yield pos2 } をなくす

```
def try(re, str, pos, &b )
  ...
    try_cat(re, str, pos, &b )
  ...
    try_alt(re, str, pos, &b )
  ...
    try_rep(re, str, pos, &b )
  ...
end
```

&をつけた引数を使う

この引数にはProcが渡される

ブロックからの脱出の例

- Array#each のブロック内から return している

```
def ematch_include(r, str)
  enumre(r).each {|s|
    return true if str.include? s
  }
  false
end
```

Proc (lambda)

- 呼び出せるコードを表現したオブジェクト
- ブロックを受け取る他に lambda で生成できる
- Proc#call で呼び出せる
- Lisp, Scheme などでいうクロージャ
- コード + 変数などの状態(環境)

Procの生成: lambda

- lambda による Proc の生成
  lambda {|args| code }
  lambda { code }　　引数がない場合

  % ruby -e 'p lambda {|v| v + 1}'
  #<Proc:0xb7e1c870@-e:1>

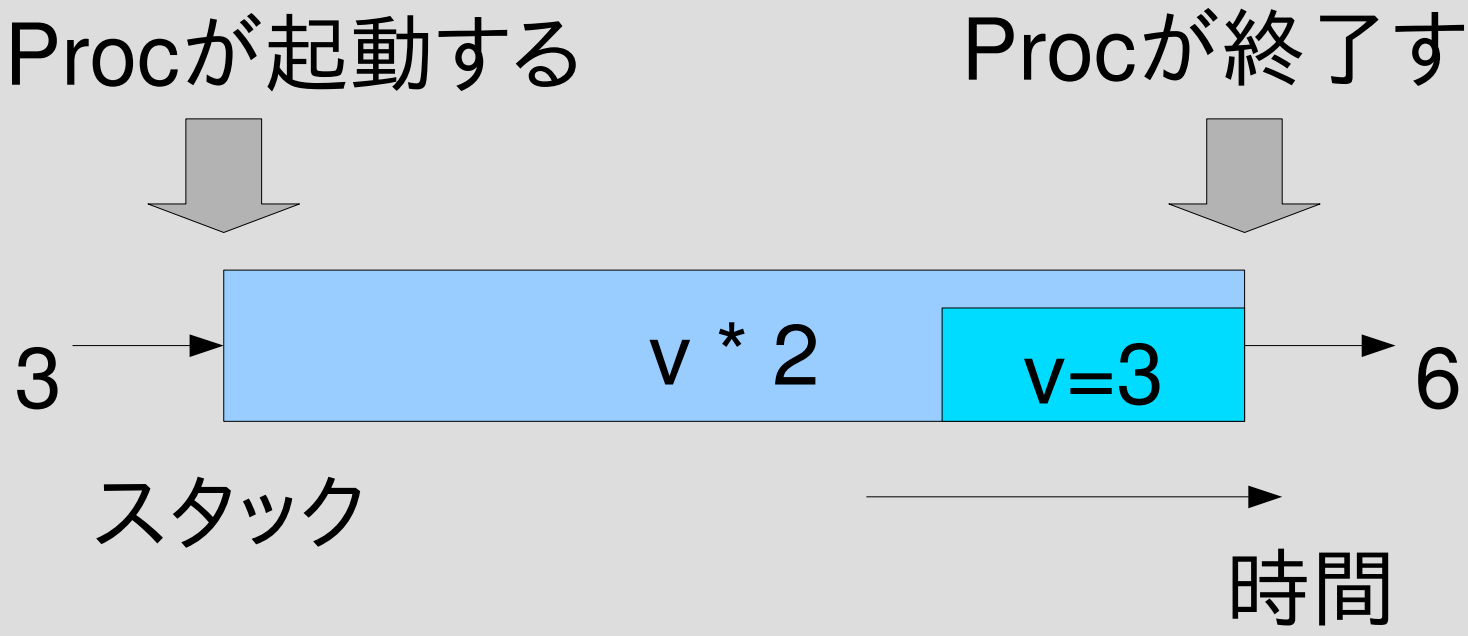
メモリ内の 0xb7e1c870 に存在

-e の引数の 1行めで生成された

Procの呼び出し: Proc#call

- lambda {|v| v * 2 }.call(3)　　#=> 6

- 動作時にはスタックフレームが確保される
- スタックフレームには v の値などが記録される



Procの生成: &引数

- &引数でブロックを Proc として受け取れる

```
% ruby -e '
def m(&block)
  p block
end
m {|v| v + 1}'
#<Proc:0xb7d7acc8@-e:5>
```

lambda もどき

- &引数で lambda のような機能を実現できる

```
% ruby -e '
def lambda2(&block)
  block
end
p lambda2 {|v| v + 1}'
#<Proc:0xb7debcfc@-e:5>
```

first class object

- Proc は first class object (一級オブジェクト)
  - 変数に代入してとっておける
  - 引数に渡せる
  - 返り値にできる
  - 要するに普通の値として扱える

countup2 (Proc版 countup)

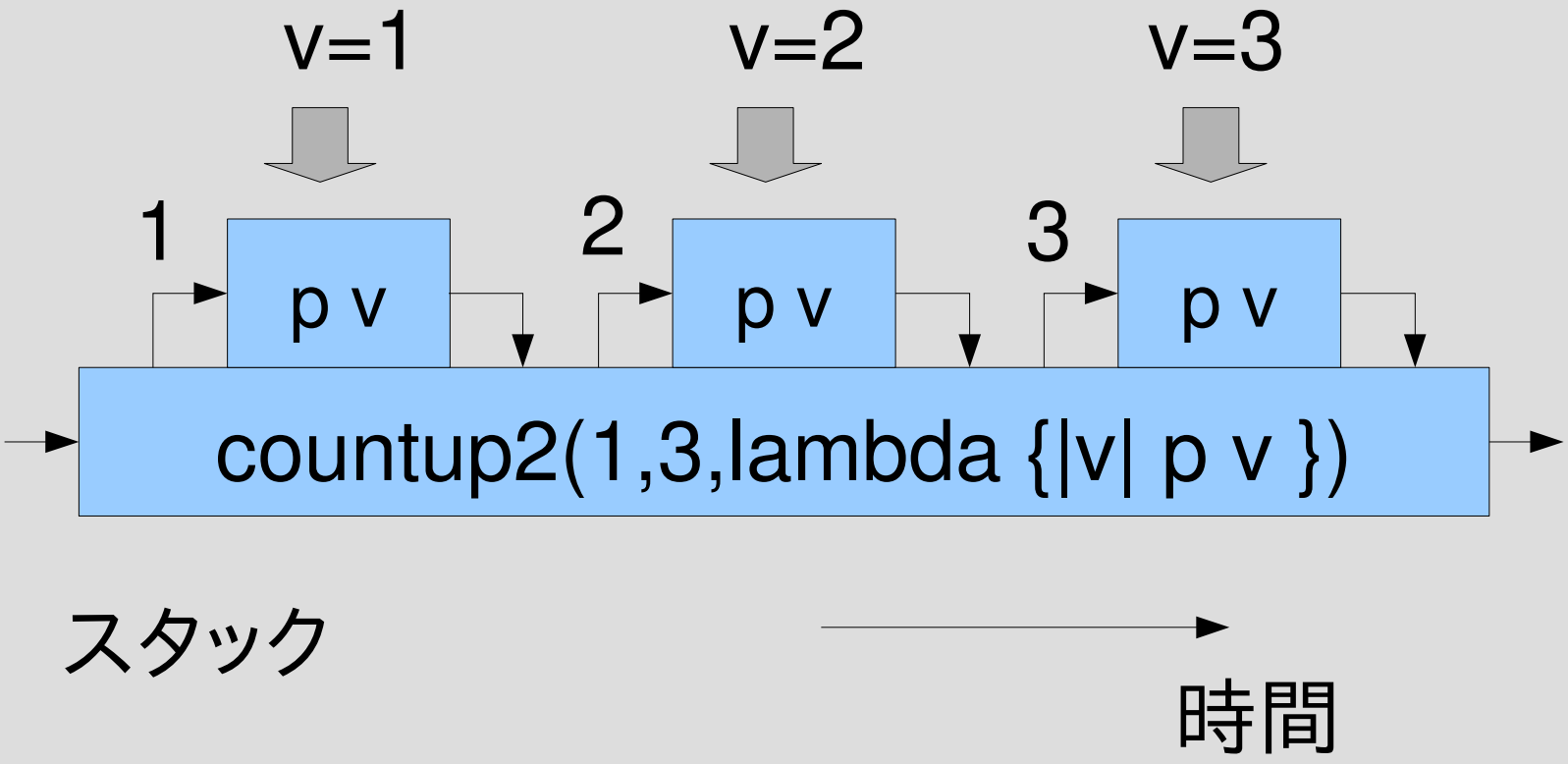
```
def countup2(first, last, pr)
  i = first
  while i <= last
    pr.call(i)
    i += 1
  end
end
countup(1, 3, lambda {|i| p i })
```

実行結果:

```
1
2
3
```

countup2 の動作

- countup2 の起動中に 3 回 {|v| p v } が起動する



ブロックによる countup との比較

Proc版

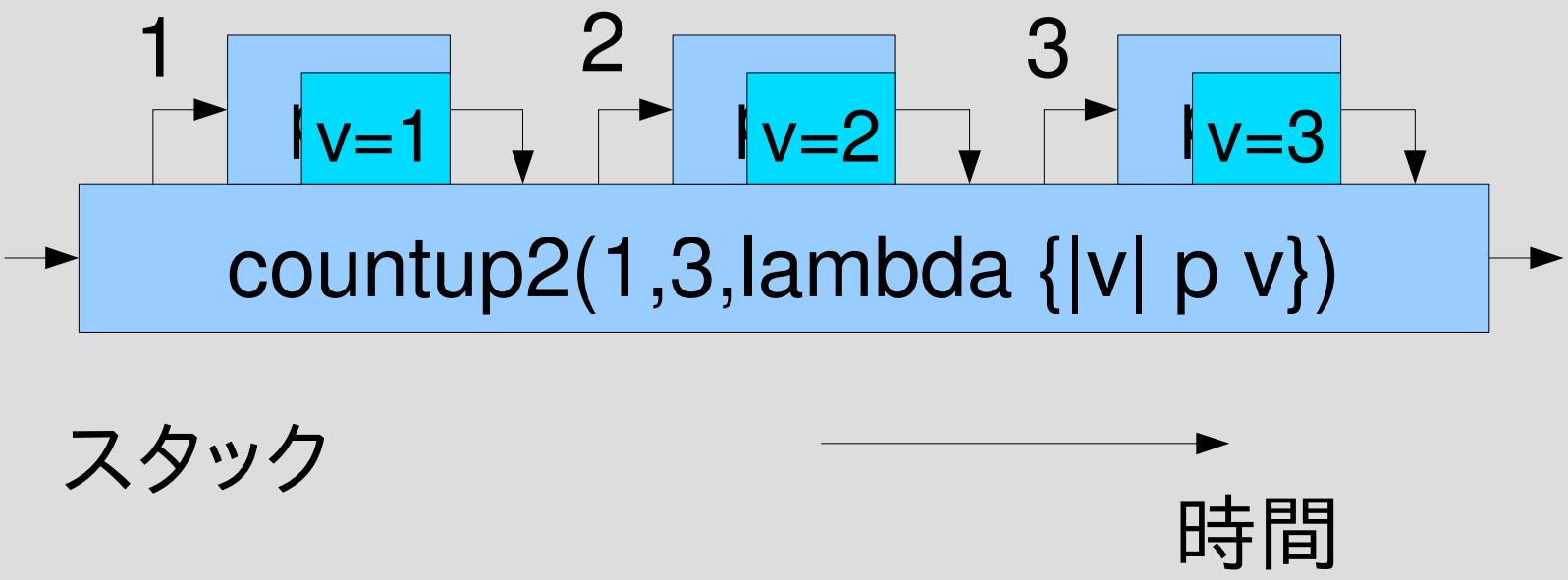
```
def countup2(first, last, pr)
  i = first
  while i < last
    pr.call(i)
    i += 1
  end
end
countup2(1, 3,
  lambda {|v| p v })
```

ブロック版

```
def countup(first,last)
  i = first
  while i < last
    yield i
    i += 1
  end
end
countup(1,3) {|v| p v}
```

ブロックと変数

- ブロック引数はブロックが起動したときにできる
- ブロックが起動したときにできるスタックフレームに記録される
- ブロック内部だけで使う変数も同様

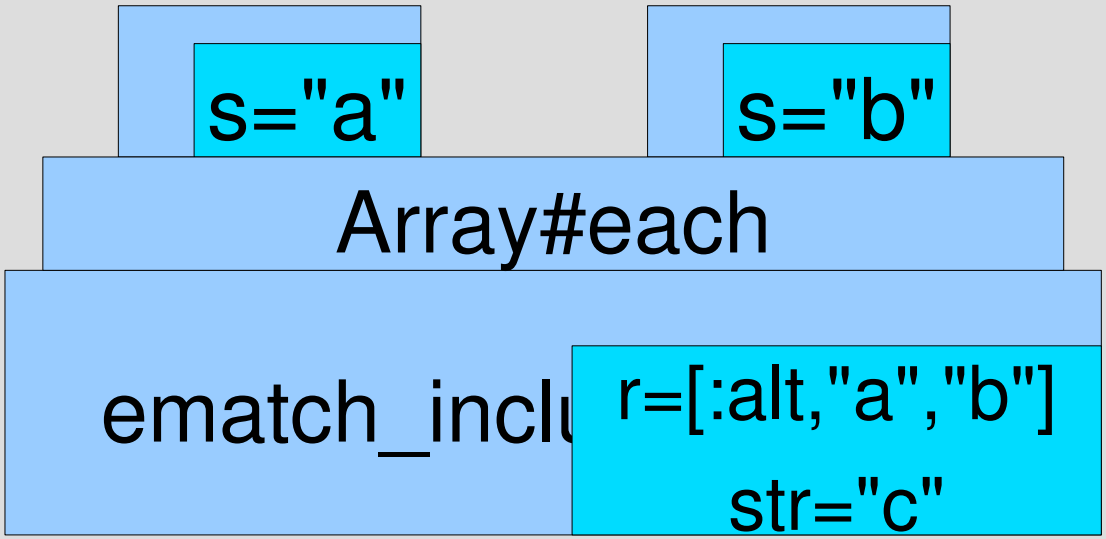


外側の変数をアクセスできる

- Array#each に能えたブロック内で str を参照している

```
def ematch_include(r, str)
  enumre(r).each {|s|
    return true if str.include? s
  }
  false
end
```

r=[:alt, "a","b"]
str="c" の場合



トップレベル

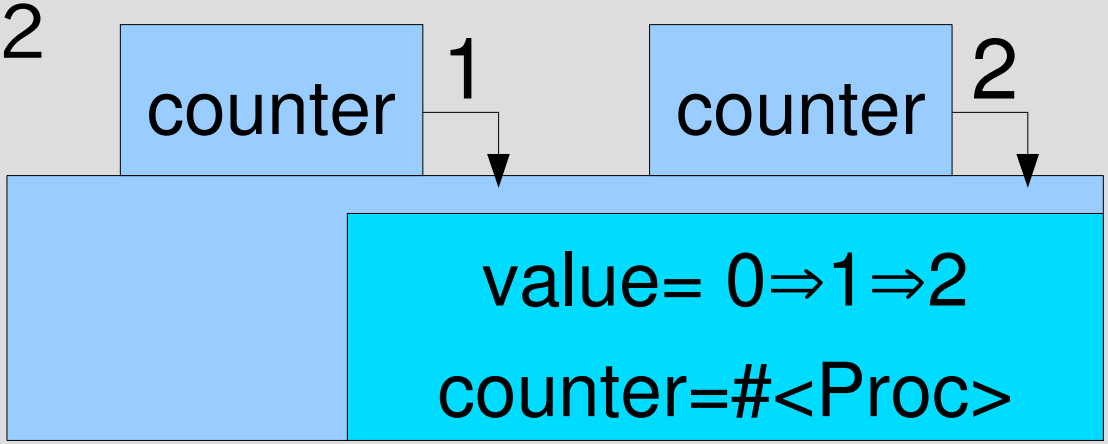
- 一番外側のことをトップレベルという
- トップレベルにも対応するスタックフレームがある
- トップレベルで定義された変数はそこに記録される

value = 0

外側の変数をアクセスする例

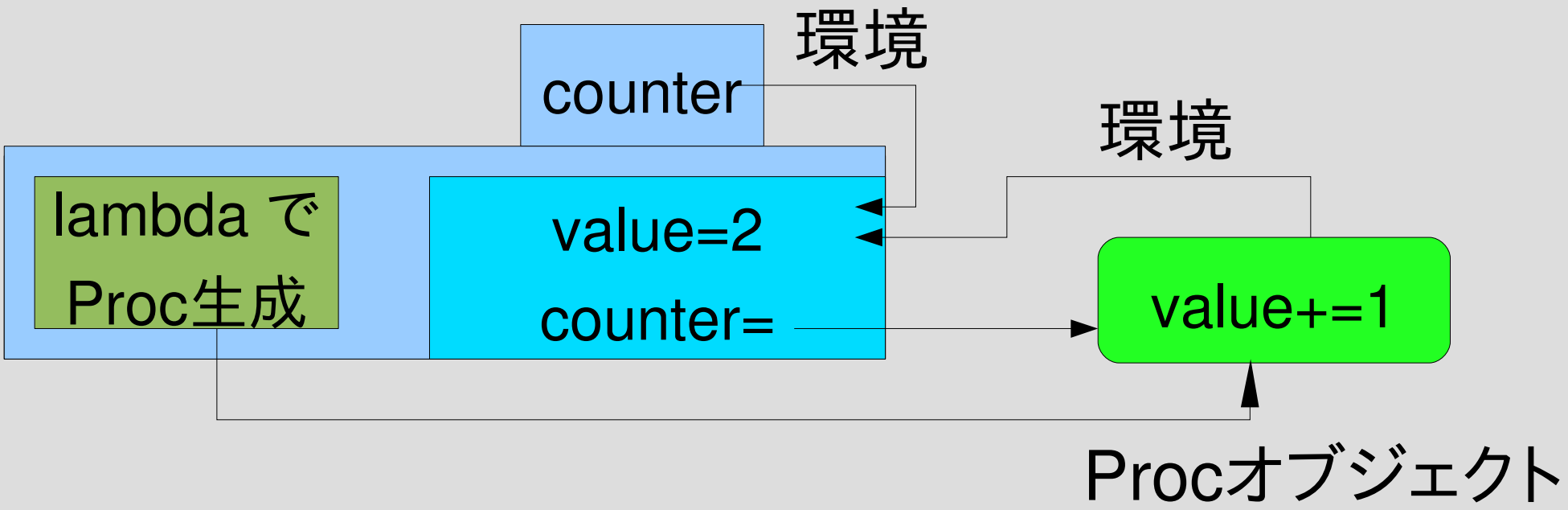
- Proc も外部の変数を参照・変更できる

```
value = 0
counter = lambda {
  value += 1
}
p countur.call    #=> 1
p countur.call    #=> 2
```



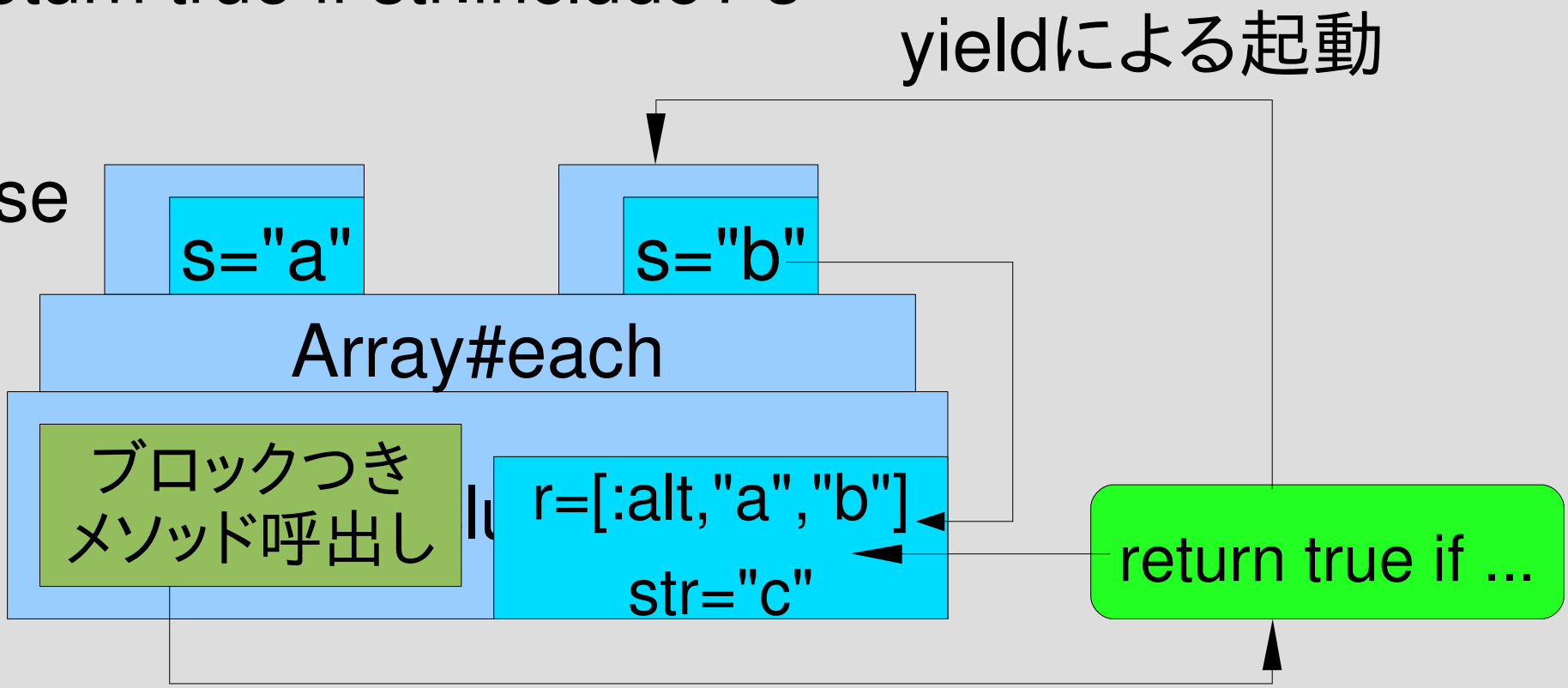
lambda = コード + 環境

- Proc が評価される文脈を環境という: 変数など
- Proc 生成時の環境が記録される
- 外部の変数へのアクセスはその環境へのアクセス



外側の変数をアクセスできる

```
def ematch_include(r, str)
  enumre(r).each {|s|
    return true if str.include? s
  }
  false
end
```



変数の共有

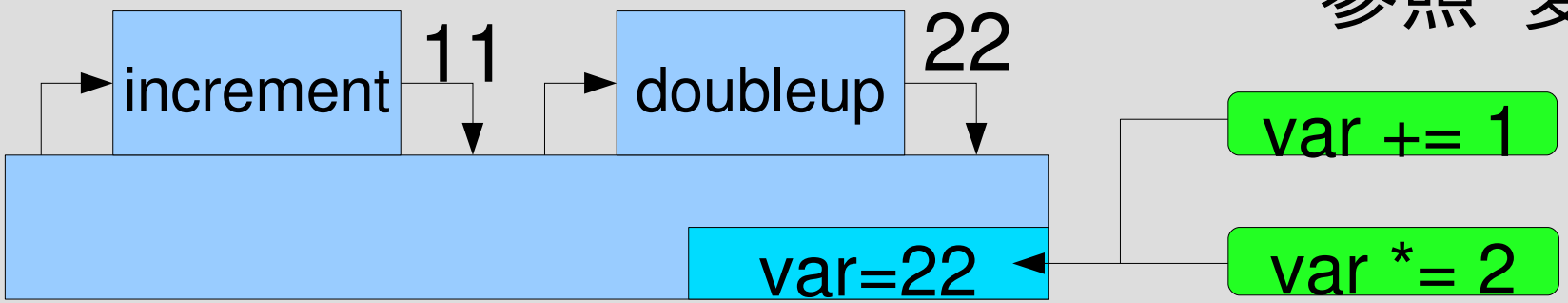
- 外側の変数を複数の Proc で共有できる
- var = 10
  increment = lambda { var += 1 }
  doubleup = lambda { var *= 2 }
  p increment.call　　#=> 11
  p doubleup.call　　#=> 22

var は

increment からも

doubleup からも

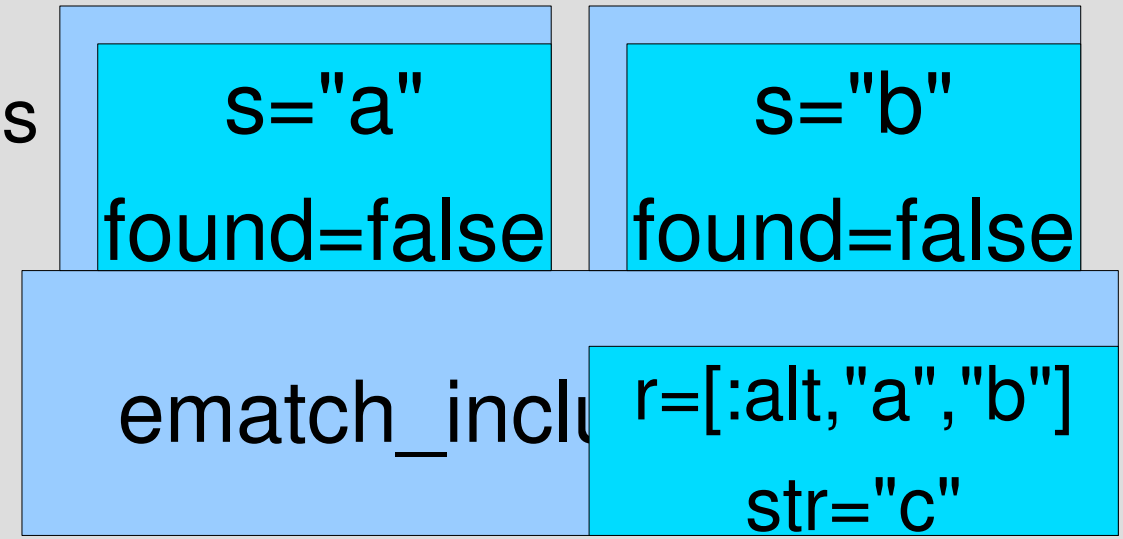
参照・変更できる



ブロックローカル変数

- ブロック内で初めて代入した変数 (ブロックの外では使っていない変数) はそのブロック内だけで有効
- ブロック引数もブロックローカル変数の一種

```
def ematch_include(r, str)
  enumre(r).each {|s|
    found = str.include? s
    return true if found
  }
  false
end
```



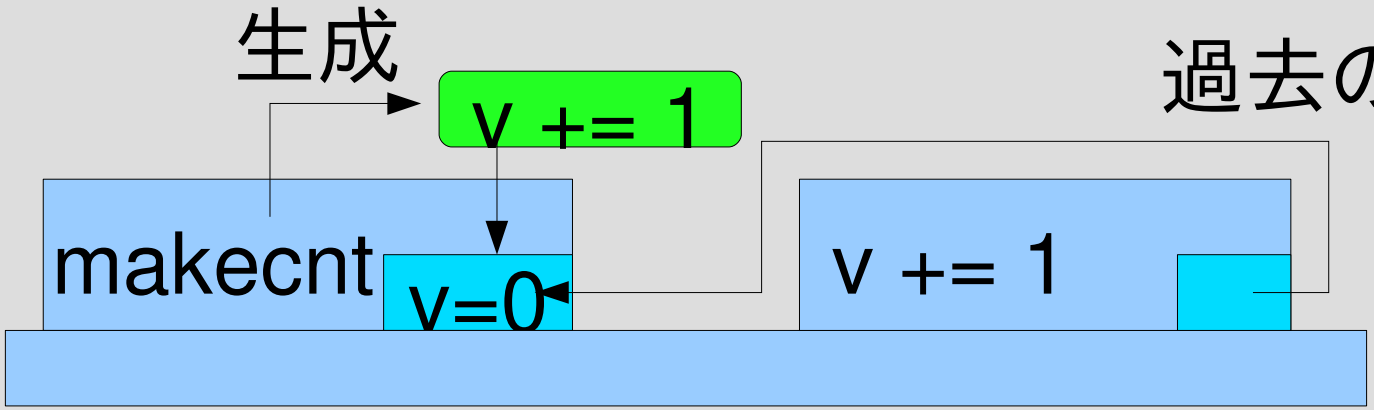
(Array#each は省略)

Proc を返す

- あるメソッドで Proc を生成して返すことができる

```
def makecnt
  v = 0
  lambda { v += 1 }
end
p makecnt.call    #=> 1
```

- v という変数を保持する環境は消えてない?



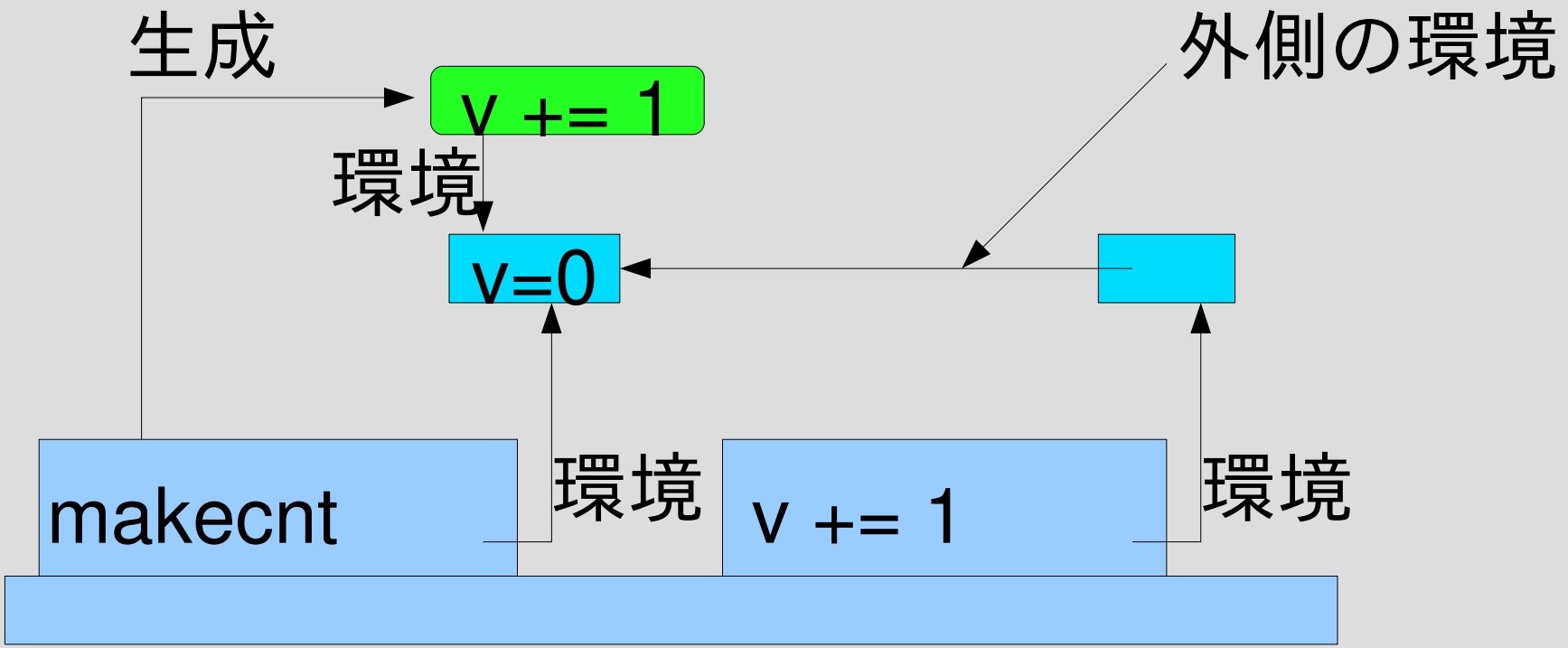
過去の環境を参照?

そんなこと

できるの?

環境は消えない

- そじつは環境はメソッドが終わった後も生き残る
- 参照する Proc が存在する限り存在する
- 環境はスタックの外に保持される



メソッドと Proc

- どちらも呼び出せるコード
  - 本質的な違いはあまりない
- 定義のしかたが違う
  - def m(args) code end
  - lambda {|args| code }
- 呼び出しかたが違う
  - m(args)
  - pr.call(args)
- 外側のローカル変数
  - メソッドは参照できない (でも、定数などの環境はある)
  - Proc は参照できる
- 引数の扱いが微妙に違う

ブロックと Proc

- ブロック = lambda + 脱出先
  - next
  - break
  - return
- ブロックと Proc は相互に変換できる (&引数)

脱出

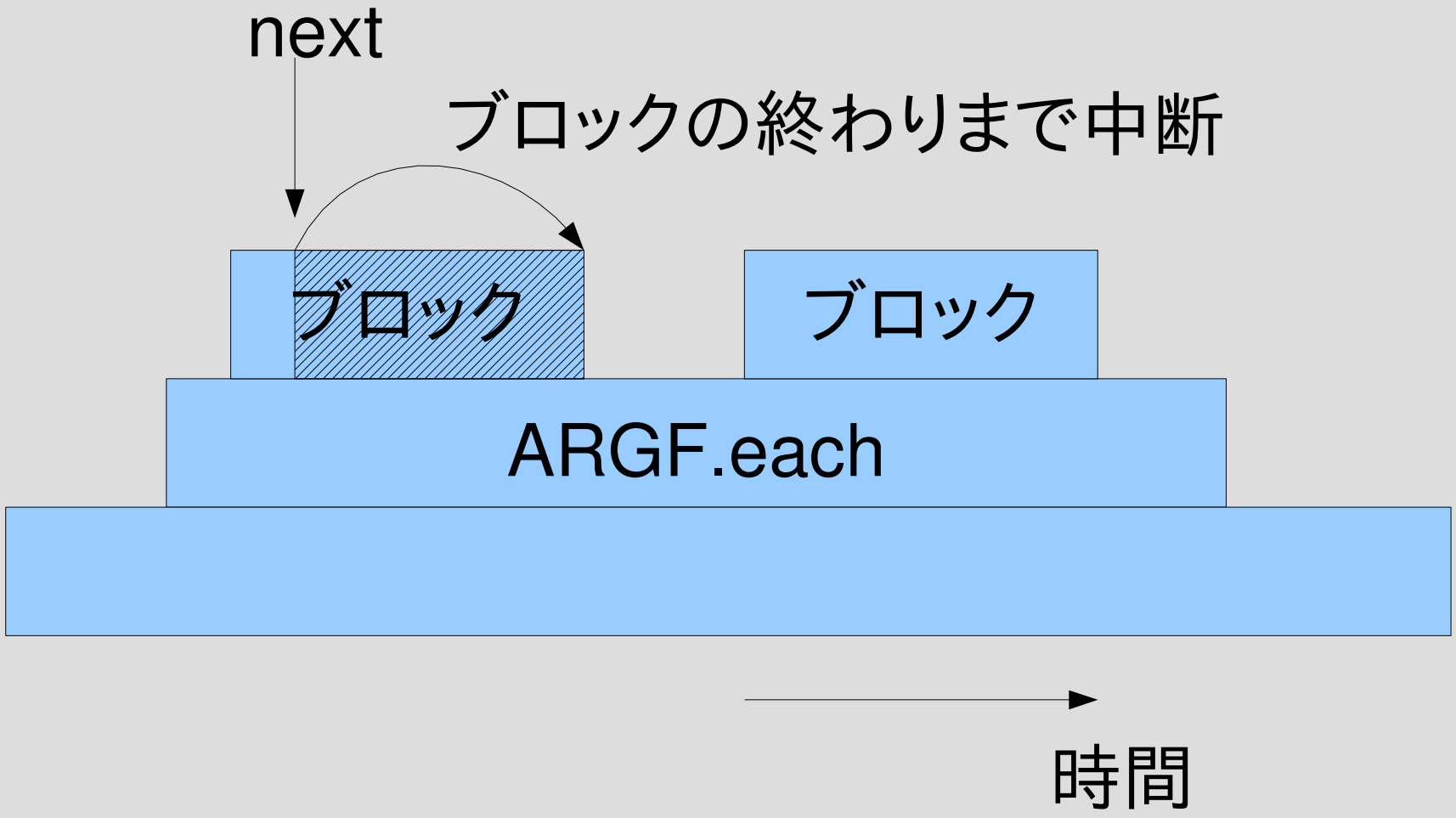
- next　　ブロックからの脱出
- break　　ブロックつき呼び出しからの脱出
- return　　メソッドからの脱出

next

- ブロックからの脱出
- ループの次の繰り返しを始めるのによく使う
- C の continue に近い

- ARGF.each {|line|
  next if /^#/ =~ line　　# コメントだったら次へ
  コメントでない行を処理
}
  ブロックが終わって
  ARGF.each 内の yield が返る所

next の動作



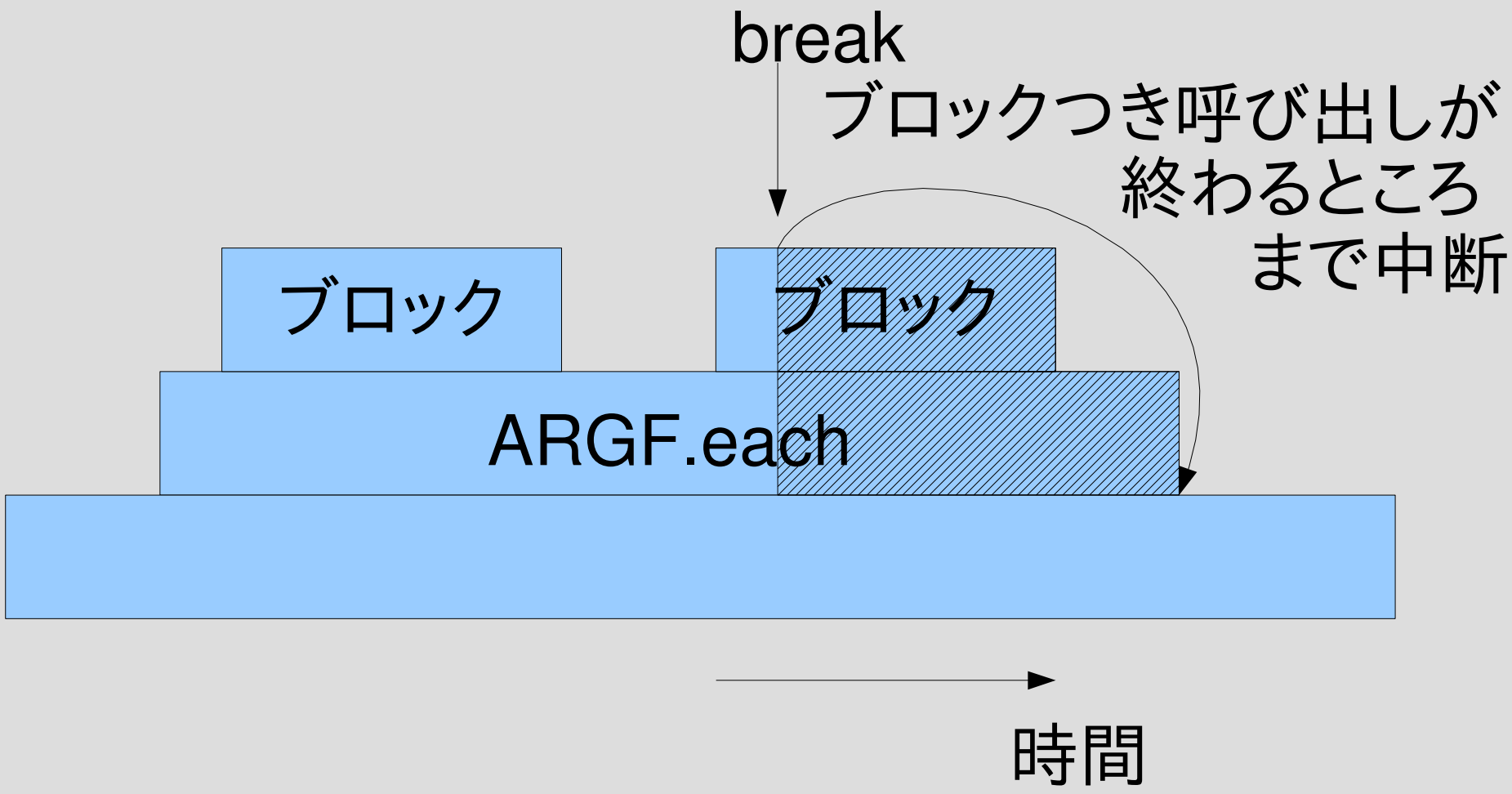
break

- break はブロックつき呼び出しからの脱出
- ループを止めるのによく使う
- C の break に対応する

- ARGF.each {|line|
  break if /^__END__$/ =~ line
  __END__ 以前の行を処理
  }

ARGF.each が終わる所

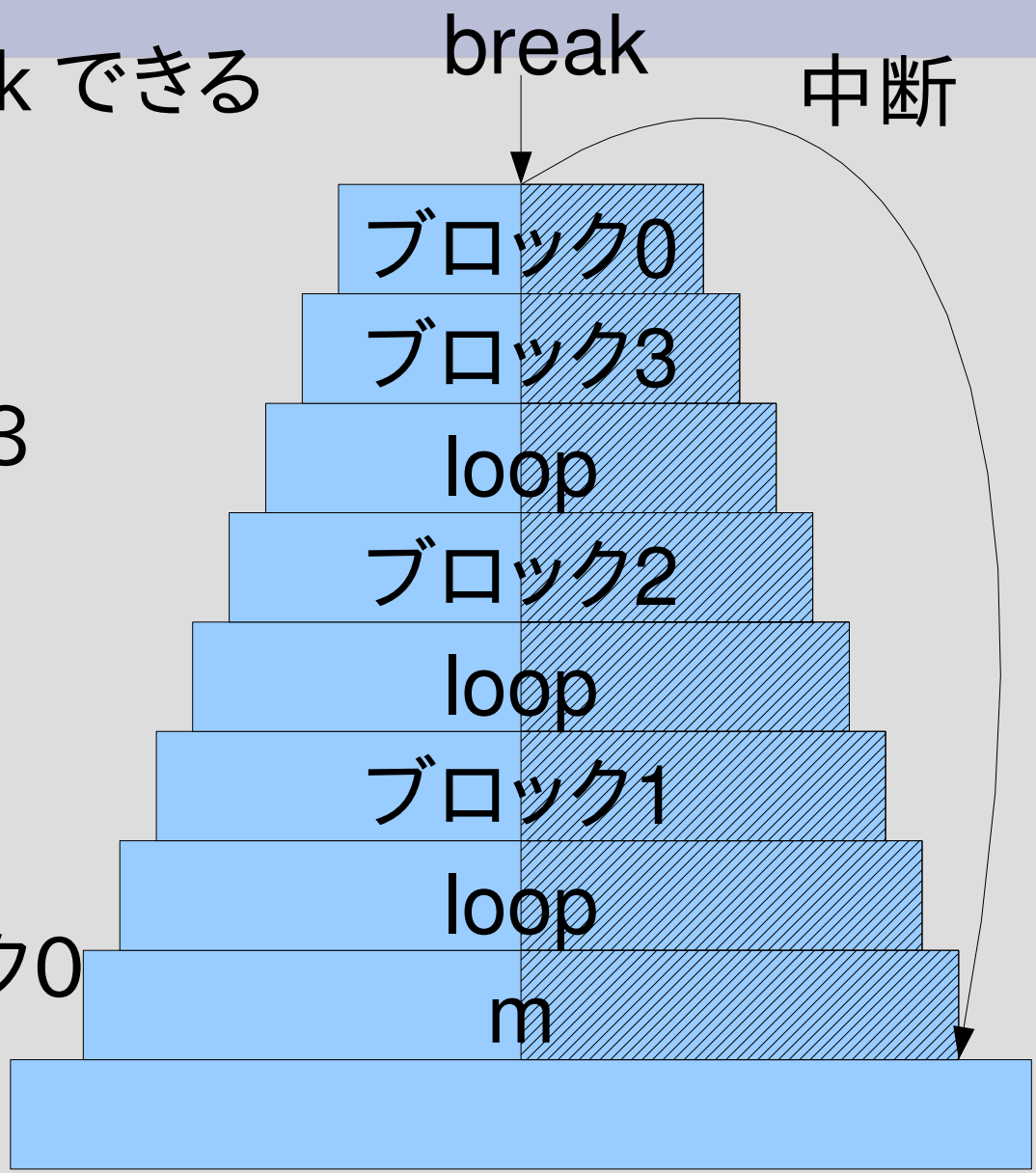
break の動作



ネストの中から break

- 深い所からでも break できる

```
def m
  loop { # ブロック1
    loop { # ブロック2
      loop { # ブロック3
        yield
      }
    }
  }
end
m { break } # ブロック0
```



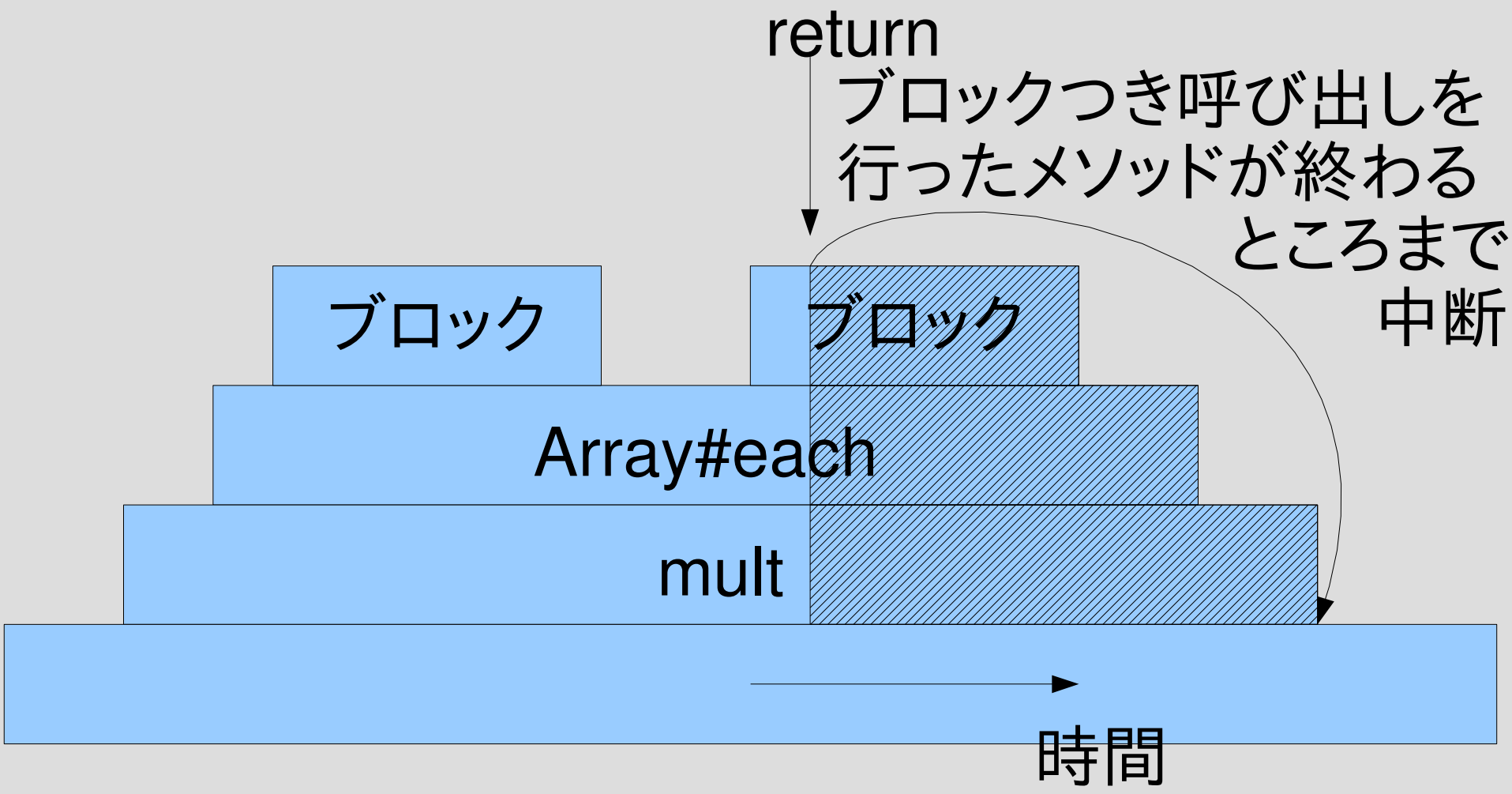
return

- return はメソッドからの脱出
- メソッドの途中で結果が判明したときによく使う
- def mult(ary)　# ary の要素をぜんぶ掛けて返す

```
  result = 1
  ary.each {|v|
    return 0 if v == 0    # 0 があったら結果は 0
    result *= v
  }
  result
end
```

メソッドが終わる所

return の動作

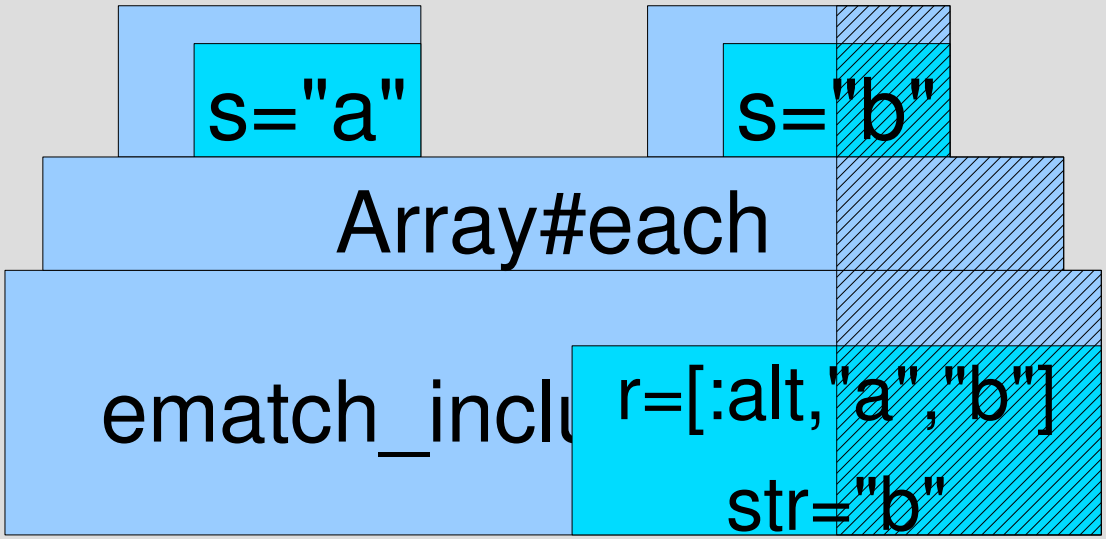

return
ブロックつき呼び出しを
行ったメソッドが終わる
ところまで
中断
ブロック
ブロック
Array#each
mult
時間

ematch_include の return

- Array#each のブロック内から return している

```
def ematch_include(r, str)
  enumre(r).each {|s|
    return true if str.include? s
  }
  false
end
```

r=[:alt, "a","b"]
str="b" の場合



ブロックの Proc 化

- 仮引数の最後の & つき引数で、ブロックとして渡したものを Proc として受け取れる
- こうして作った Proc の中には脱出先も入っている

-
```
def m(&b)
  p b.call(10)
end
m {|v| v + 2 }     #=> 12
```

-
```
def m
  p yield(10)
end
m {|v| v + 2 } #=> 12
```

Proc のブロック化

- 実引数の最後の & つき引数で、Proc をブロックとして渡せる

- ```
  def m
    p yield(20)
  end
  add1 = lambda {|v|
    v + 1
  }
  m(&add1)    #=> 21
  ```

- ```
  def m
    p yield(20)
  end
  m {|v| v + 1 }  #=> 21
  ```

ブロックを引きわたす

- & を使って、与えられたブロックを他のメソッドにそのまま渡すことができる

-
```
def m
  p yield(10)
end
def n(&b)
  m(&b)
end
n {|v| v * 2 }      #=> 20
```

-
```
def m
  p yield(10)
end
def n
  m {|x| yield x}
end
n {|v| v * 2 }    #=> 20
```

try でブロックを引きわたす

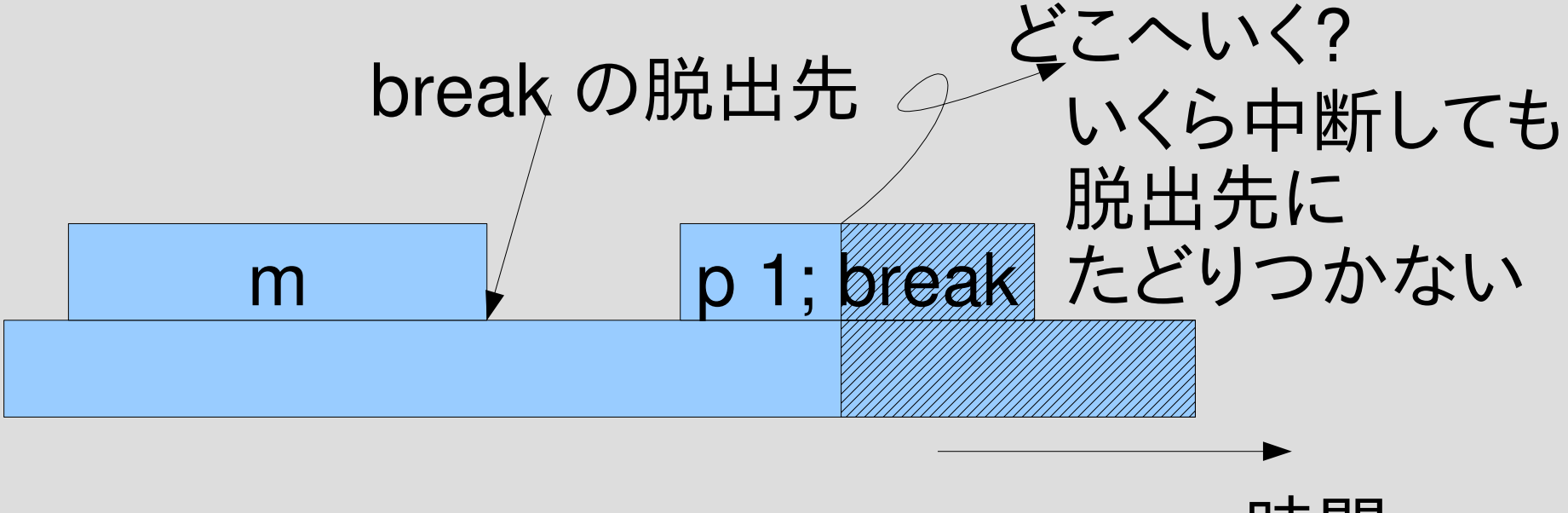
```
def try(re, str, pos, &b )
  ...
    try_cat(re, str, pos, &b )
  ...
    try_alt(re, str, pos, &b )
  ...
    try_rep(re, str, pos, &b )
  ...
end
```

注意: 脱出先が すでに終わっている場合

- ブロックを Proc 化して、その Proc を値として返すと、break, return の脱出先を通り越すことができる
- そのようなときには break, return してはならない
- なおnextの脱出先は常に存在するので問題は生じない

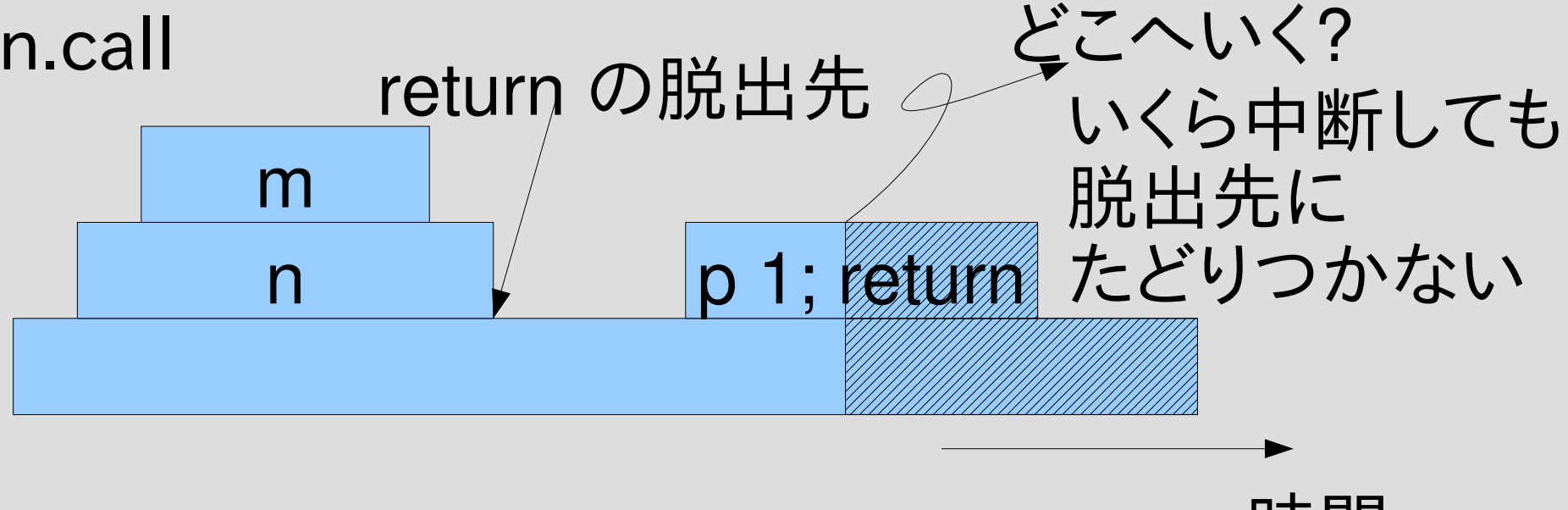
break 先が終わっている

- def m(&b)
  b
  end
  m { p 1; break }.call



return 先が終わっている

- def m(&b)
  b
  end
  def n
  m { p 1; return }
  end
  n.call



脱出先が存在しないときの挙動

- break, return は LocalJumpError になる?
- Ruby のバージョン依存かも? 不安定?
- このあたりの挙動に依存してはならない

レポート

- try を使って ematch_exact と同じ動作をする rx_exact を書く
- try を使って ematch_include と同じ動作をする rx_include を書く

- ユニットテストを配布するので確かめること
- 〆切 2008-06-10 12:00
- RENANDI
- 拡張子が txt なプレインテキストで

関連資料

- try の説明は第6回の資料に説明がある
- ematch_exact, ematch_include は第5回の資料に説明がある

rx_exact の実行例

- rx_exact("a", "a") #=> true
- rx_exact("a", "b") #=> false
- rx_exact(
  [:alt, [:cat, "c", [:cat, "a", "t"]],
  [:cat, "d", [:cat, "o", "g"]]], "dog")
  #=> true

rx_exact の実行例 (2)

- try を使うので繰り返しも扱える
- rx_exact([:rep, "a"], "aaaa") #=> true
- rx_exact([:rep, "a"], "aaaab") #=> false
- rx_exact([:cat, [:rep, "a"], "b"], "b") #=> true

String#split による文字列の分割

- rx_exact, rx_include の第2引数は文字列
- try の第2引数は文字の配列
- 文字列を文字単位に分割して配列にするには String#split を使う
- str.split(//)
- "apple".split(//)　#=> ["a","p","p","l","e"]

rx_include の実行例

- rx_include("a", "abc") #=> true
- rx_include("z", "abc") #=> false
- rx_include(
  [:alt, [:cat, "c", [:cat, "a", "t"]],
  [:cat, "d", [:cat, "o", "g"]]], "education")
  #=> true

rx_include の実行例 (2)

- try を使うので繰り返しも扱える
- rx_exact([:rep, "a"], "aaaa") #=> true
- rx_exact([:rep, "a"], "aaaab") #=> true
- rx_exact([:cat, [:rep, "a"], "b"], "b") #=> true

注意

- enumre は使わない
- 可能なら、途中で結果が分かったら中断する

- 日本語による解説もつけること

まとめ

- 前回のレポートの解説
- lambda = コード + 環境
- ブロック = lambda + 脱出先
- レポートを出した