テキスト処理 第9回 (2006-06-20)

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

- 前期試験について
- Proc (lambda)
- ブロック
- レポート

前期試験

- 2006-07-25 (火) 5時限 60分 (16:30 - 17:30)
- 持込: 一切可
- 学生証を携帯すること

- コンピュータに向かわないでプログラムを書くのはなんなので、たぶんなにかを解説させることになる

Proc (lambda)

- 呼び出せるコードを表現したオブジェクト
- メソッド呼び出し引数の & で出てきた奴
- 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 + 環境

Procの生成: lambda

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

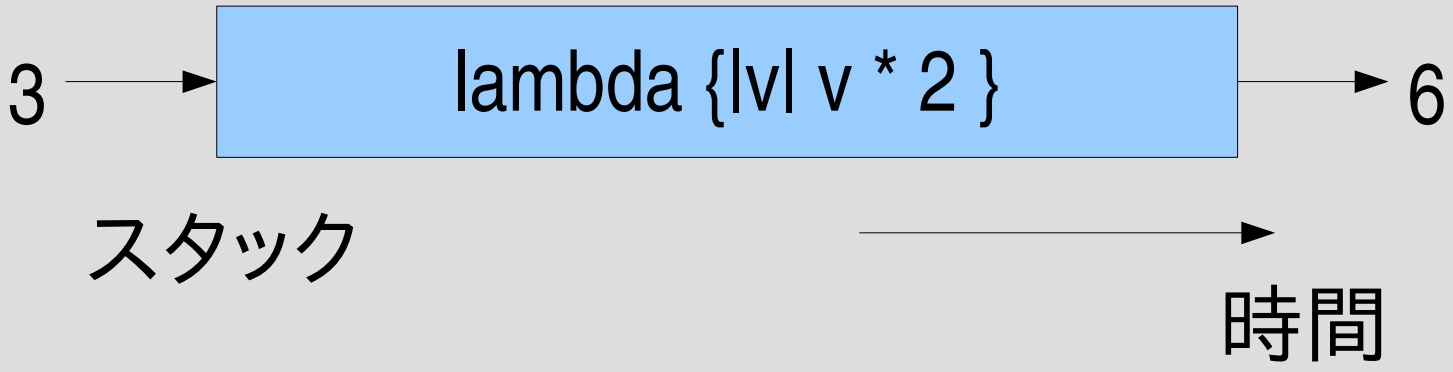
  % ruby -e 'p lambda {|v| v + 1}'
  #<Proc:0xb7e1c870@-e:1>

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
- メソッドと同様に、スタックフレームが確保されて動く



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
- first class object でないもの
  - 変数
  - 式
  - スタック (継続というものはある)
  - etc.

  - 他の言語では Java のクラスとか

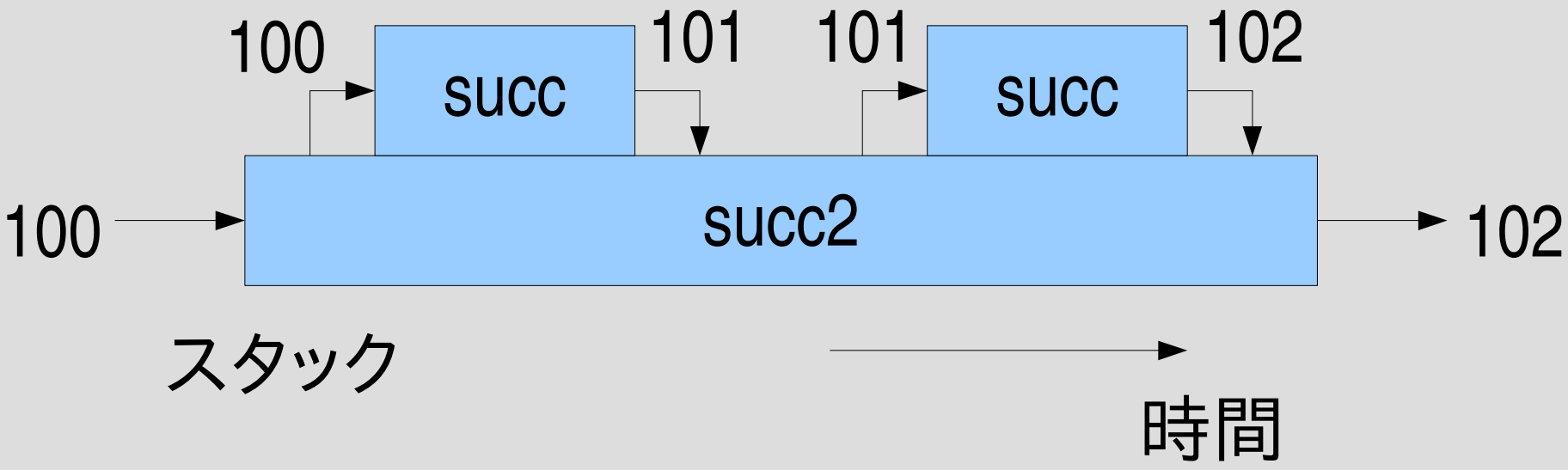
後者関数: **succ**

- 後者関数: (自然数の)次の値を返す関数
- succ = lambda {|v| v + 1 }
- p succ.call(0)　　#=> 1
- p succ.call(1)　　#=> 2
- succ.call(2)　　#=> 3
- succ.call(1000)　　#=> 1001

- def succ(v) v + 1 end　　とほぼ同じ
- p succ(1)　　#=> 2

succ2

- succ2 = lambda {|v| succ.call(succ.call(v)) }
  succ2.call(100) #=> 102

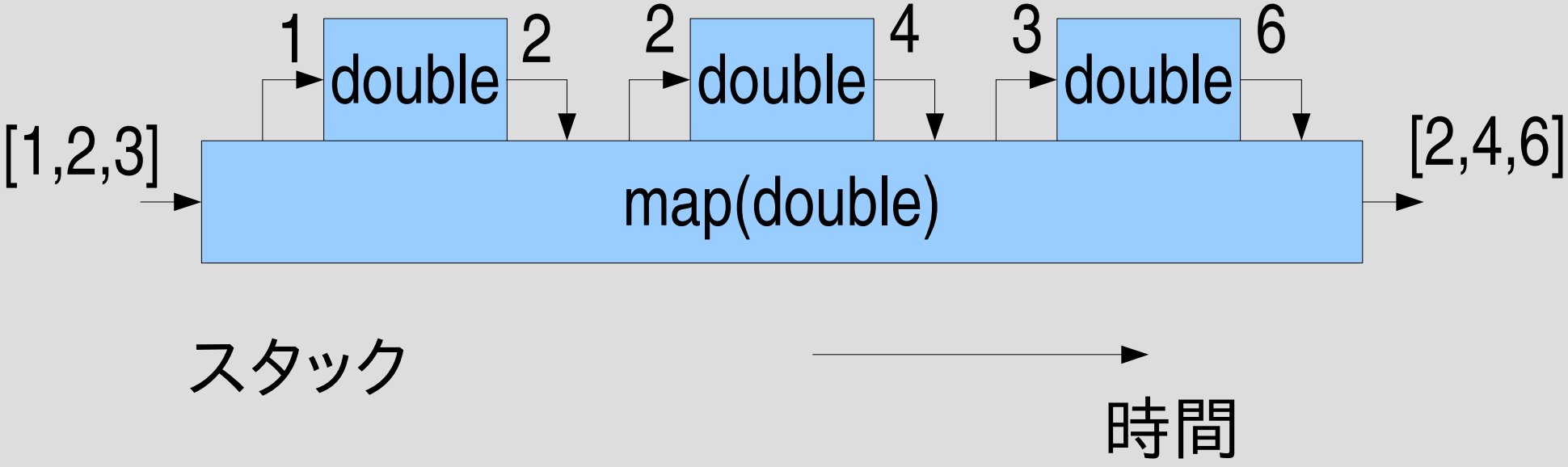


Proc を使った map

```
def map(ary, f)
  result = []
  ary.each {|v|
    result << f.call(v)
  }
  result
end
p map([1,2,3], lambda {|v| v * 2 })      #=> [2,4,6]
double = lambda {|v| v * 2 }
p map([1,2,3], double)                   #=> [2,4,6]
```

map の動作

とりあえず ary.each は無視
したおおざっぱな動作



ブロックによる map との比較

Proc版

```
def map(ary, f)
  result = []
  ary.each {|v|
    result << f.call(v)
  }
  result
end
p map([1,2,3], lambda {|v| v * 2 })
```

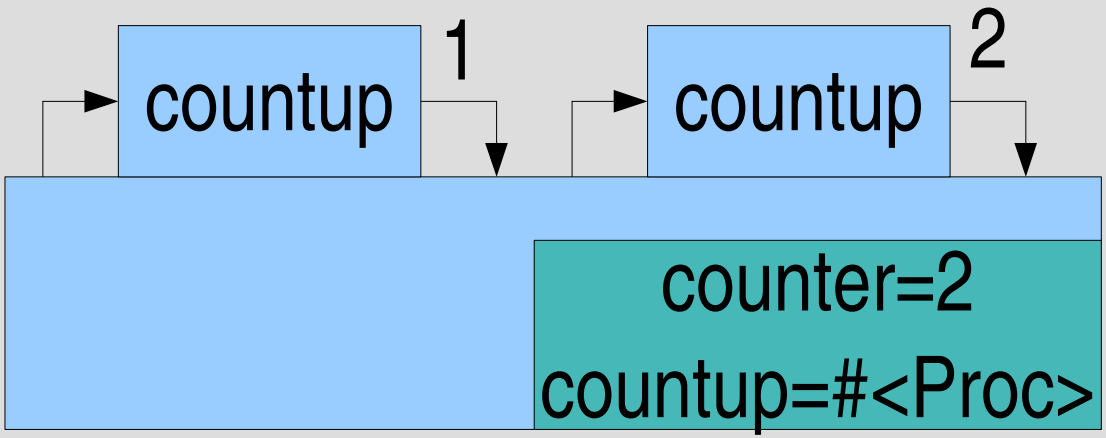
ブロック版

```
def map(ary)
  result = []
  ary.each {|v|
    result << yield(v)
  }
  result
end
p map([1,2,3]) {|v| v * 2 }
```

外部の変数アクセス

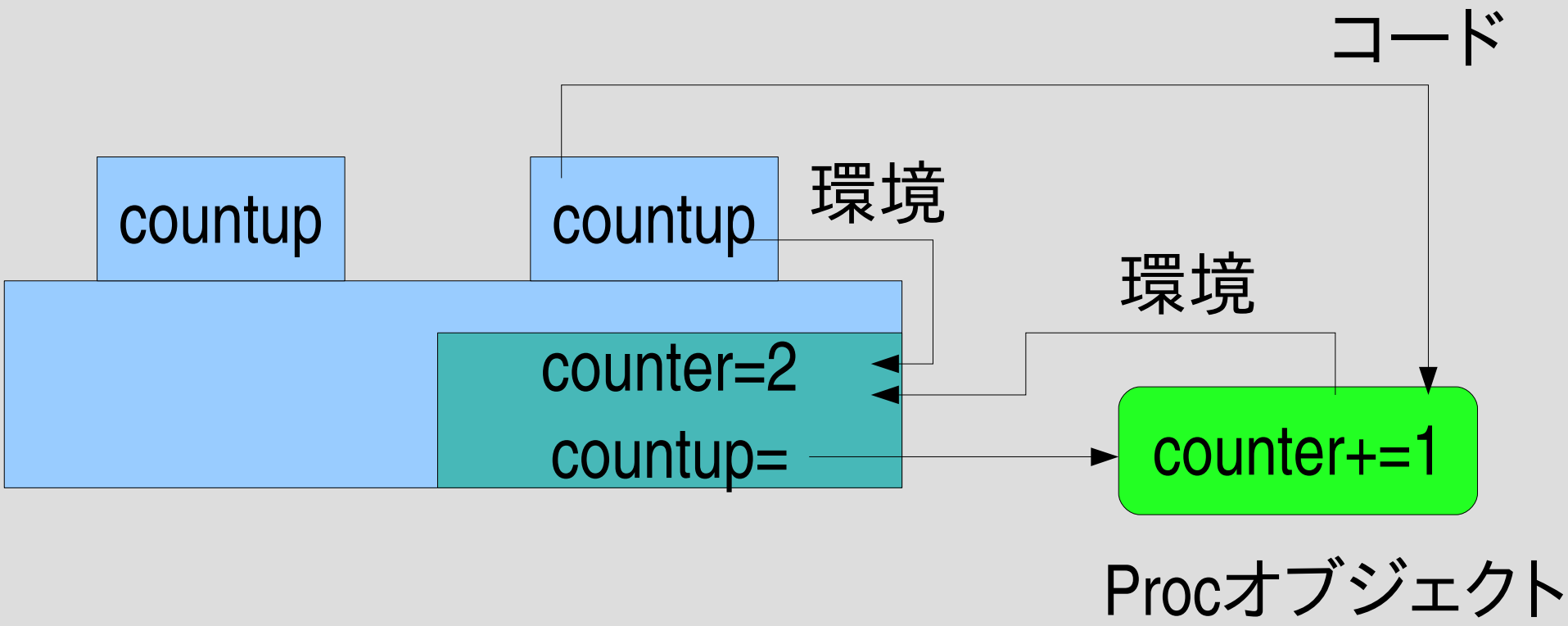
- Proc は外部の変数を参照・変更できる

```
counter = 0
countup = lambda {
  counter += 1
}
p countup.call  #=> 1
p countup.call  #=> 2
p countup.call  #=> 3
```



lambda = コード + 環境

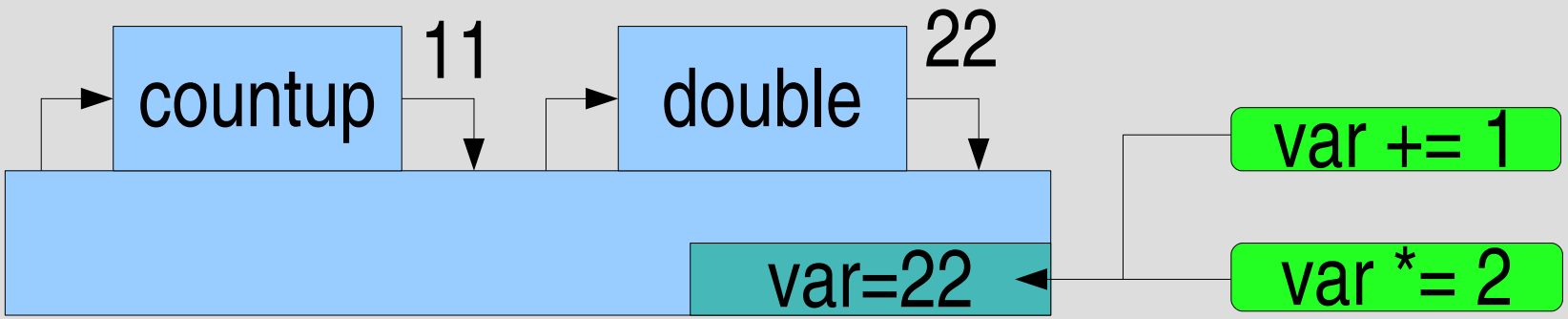
- Proc が評価される文脈を環境という: 変数, self, etc.
- 外部の変数へのアクセスは環境へのアクセス



変数の共有

- var = 10
  countup = lambda { var += 1 }
  double = lambda { var *= 2 }
  p countup.call　　#=> 11
  p double.call　　#=> 22

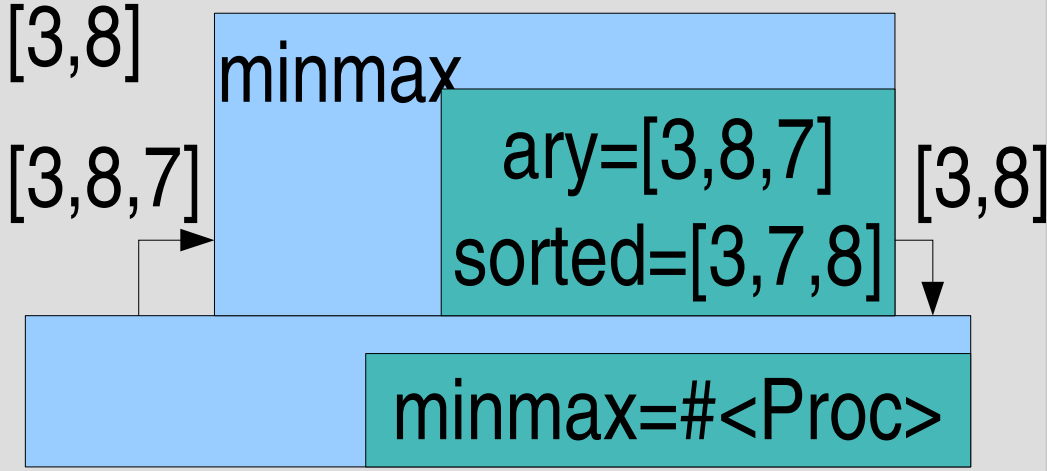
var は
countup からも
double からも
参照・変更できる



ブロックローカル変数

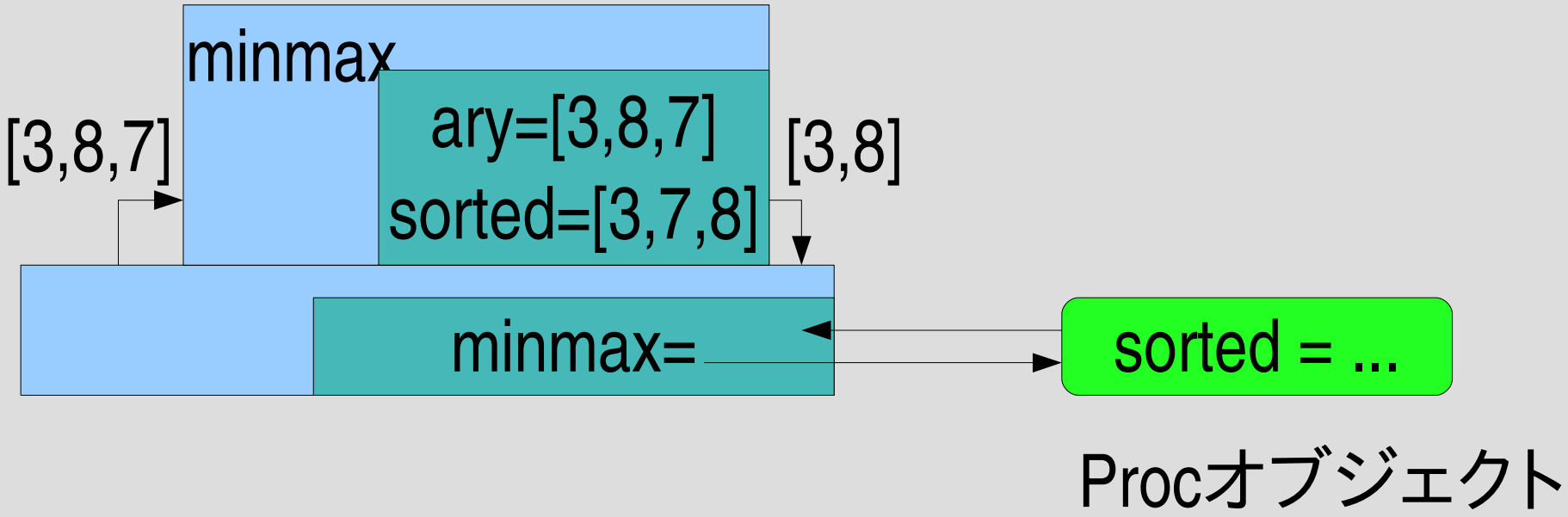
- lambda 内部だけで有効なローカル変数も使える
- ブロック内ではじめて使われたローカル変数、引数

```
minmax = lambda {|ary| # ary はブロックローカル
  sorted = ary.sort        # sorted はブロックローカル
  [sorted[0], sorted[-1]]
}
p minmax.call([3,8,7]) #=> [3,8]
p sorted      #=> エラー
```

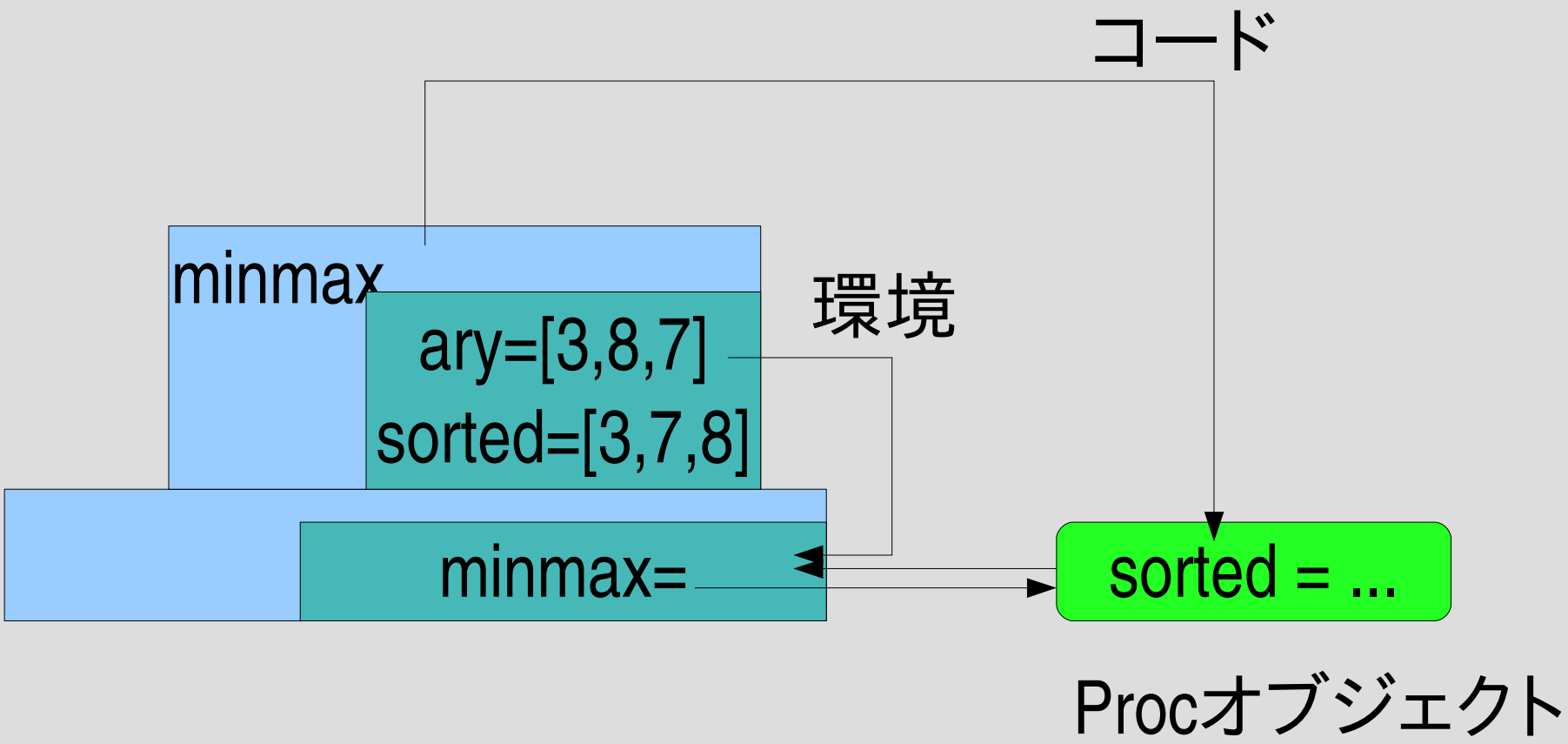


ブロックローカル変数

- ブロックローカル変数は呼び出されるたびに作られる
- ary, sortedの環境はProcオブジェクトに含まれていない



ブロックローカル変数



ブロックローカルでないブロック引数

- v = 1
  p lambda {|v| v * 2 }.call(2) #=> 4
  p v   #=> 2

- v は lambda よりも前に表れているのでブロックローカルではない
- call した時点で外側の v に代入される
- Scheme など (Ruby 以外) では v は lambda にローカルになるので差異に注意

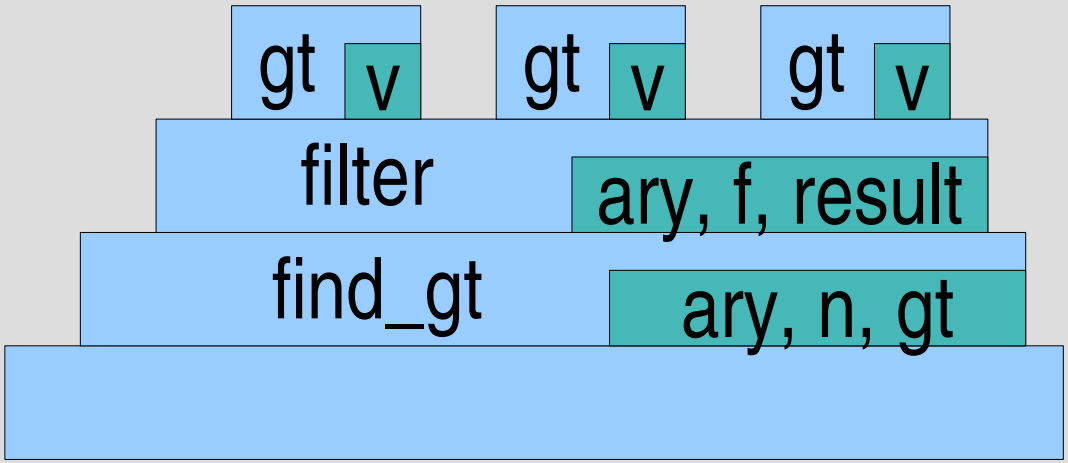
filter

```
def filter(ary, f)
  result = []
  ary.each {|v| result << v if f.call(v) }
  result
end
```

ary から条件に
あったものだけを
取り出す

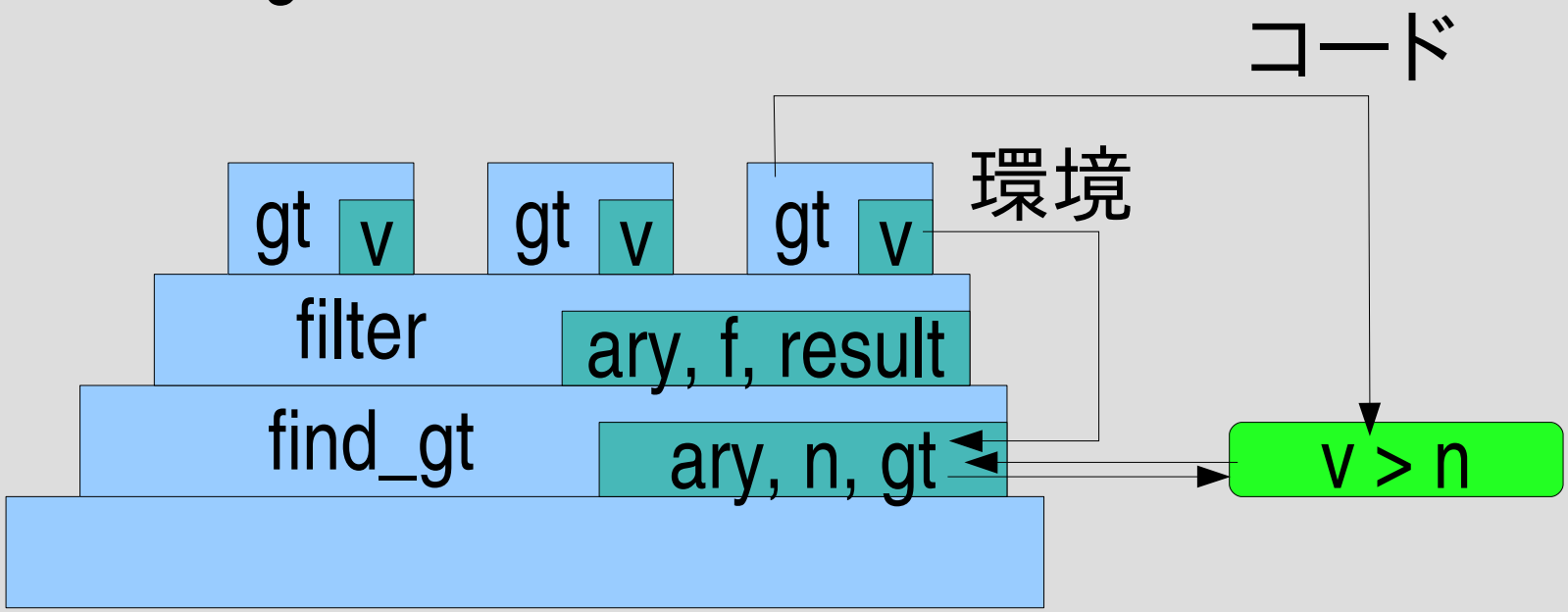
nより大きいのを取り出す

```
def find_gt(n, ary)
  gt = lambda {|v| v > n }
  filter(ary, gt)
end
find_gt(7, [3,8,7])  #=> [8]
```



環境の連鎖

- gt = lambda {|v| v > n } は外側の n を参照している
- gt から参照される外側の環境は find_gt のもの
  - gt の Proc オブジェクトに記録されてたのが find_gt の環境だから
- filter の環境は gt からは見えない

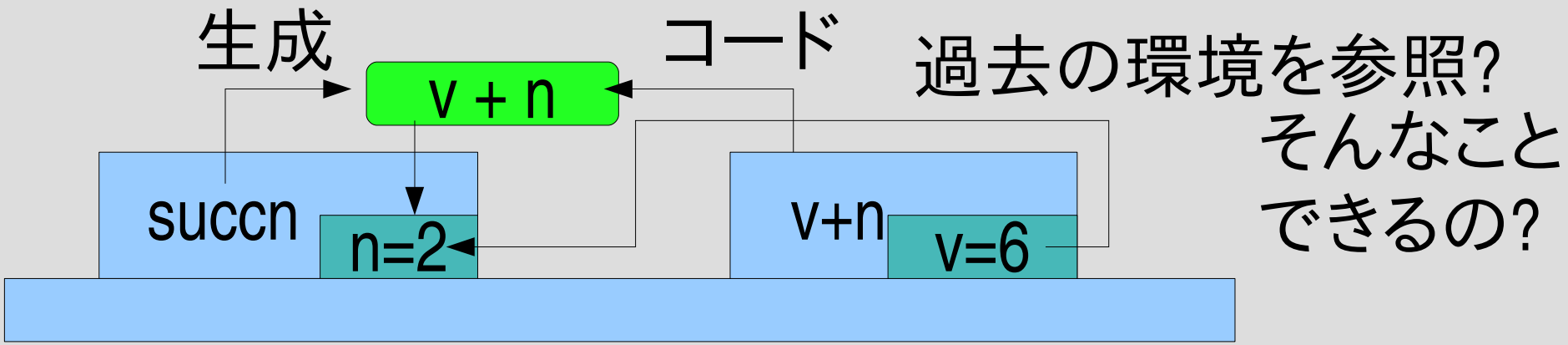


Proc を返す

- あるメソッドで Proc を生成して返すことができる

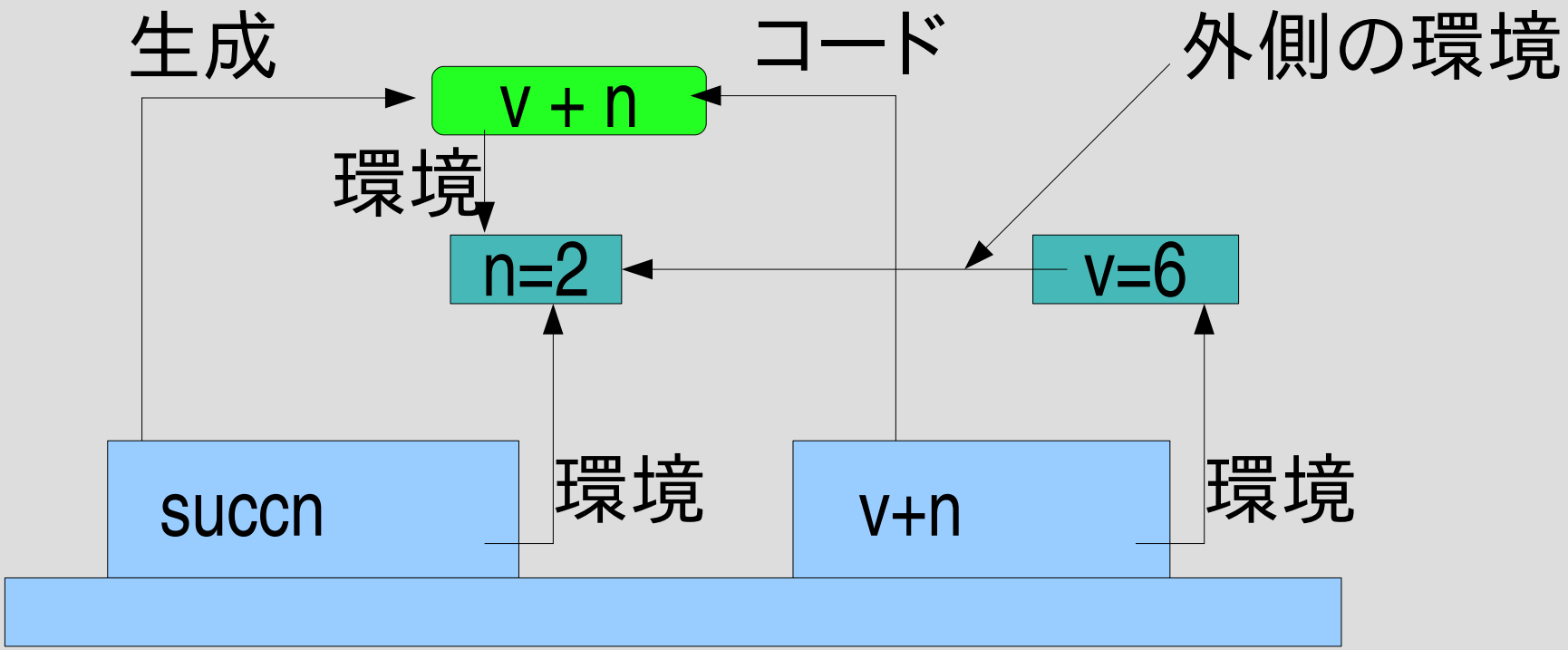
```
def succn(n)
  lambda {|v| v + n }
end
p succn(2).call(6)  #=> 8
```

- n という変数を保持する環境は消えてない?



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)
  - proc.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

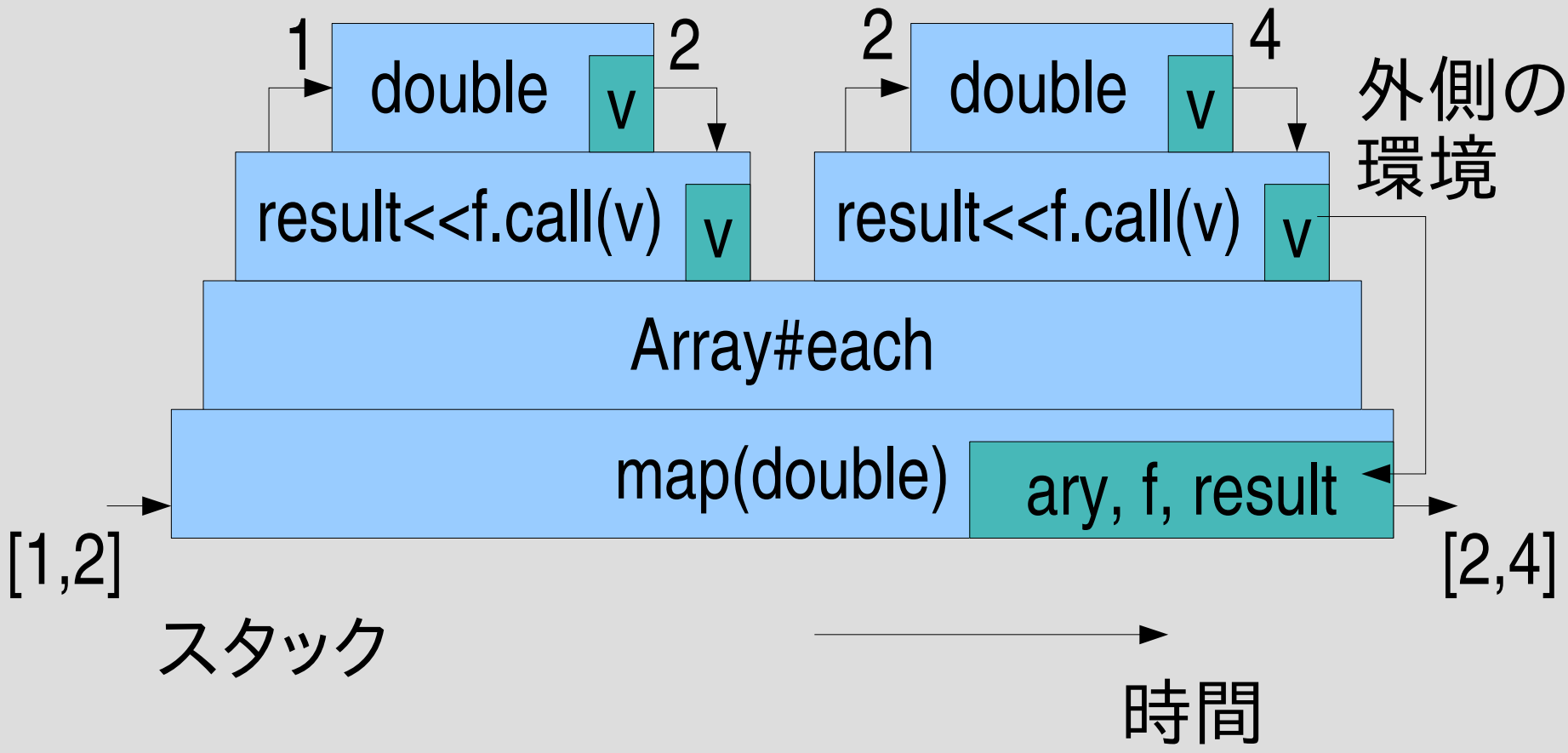
- meth(args) {|bargs| code } をブロックつき呼び出しという
- {|bargs| code } の部分をブロックという
- ブロック = lambda + 脱出先
- 脱出先のことを除けば、& で指定される特殊な引数で渡される Proc とほぼ等しい
  - meth(args) {|bargs| code } は
    meth(args, &lambda {|bargs| code }) とほぼ等しい
  - def meth(args) ... yield bargs ... end は
    def meth(args, &b) ... b.call(bargs) ... end とほぼ等しい

Proc を使った map

```
def map(ary, f)
  result = []
  ary.each {|v|
    result << f.call(v)
  }
  result
end
double = lambda {|v| v * 2 }
p map([1,2], double)                    #=> [2,4]
```

Array#each で
ブロックを使っている

map の動作
Array#each とブロックも含めて



脱出

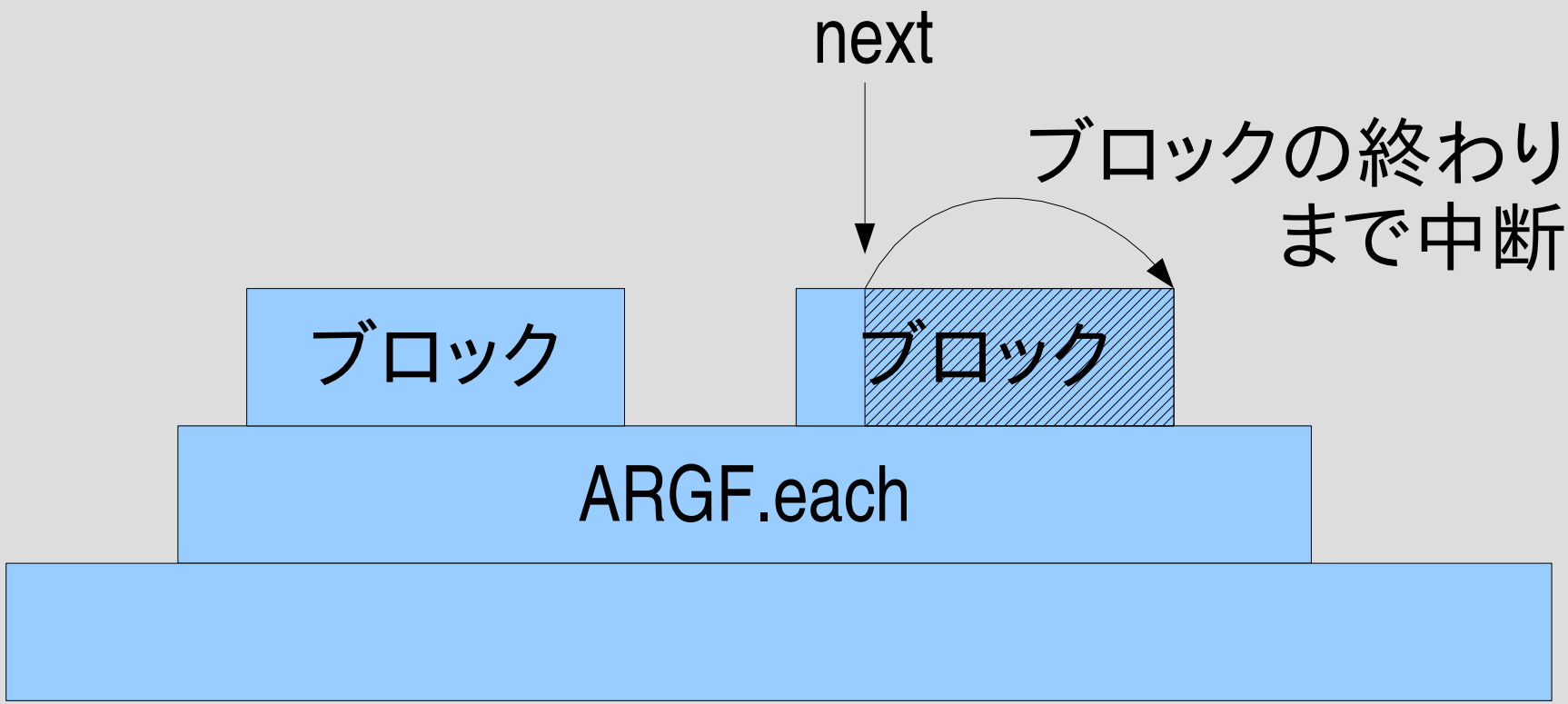
- next　　ブロックからの脱出
- break　　ブロックつき呼び出しからの脱出
- return　　メソッドからの脱出

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



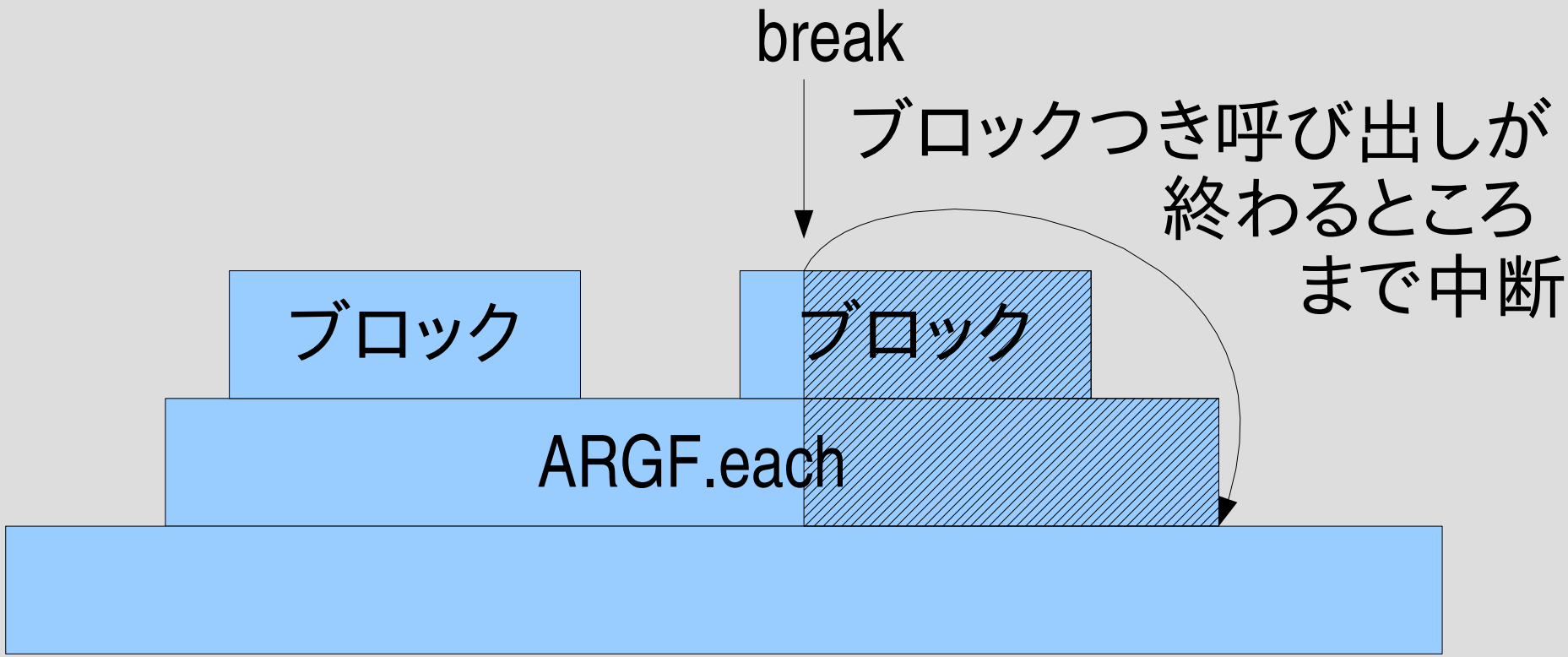
break

- 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

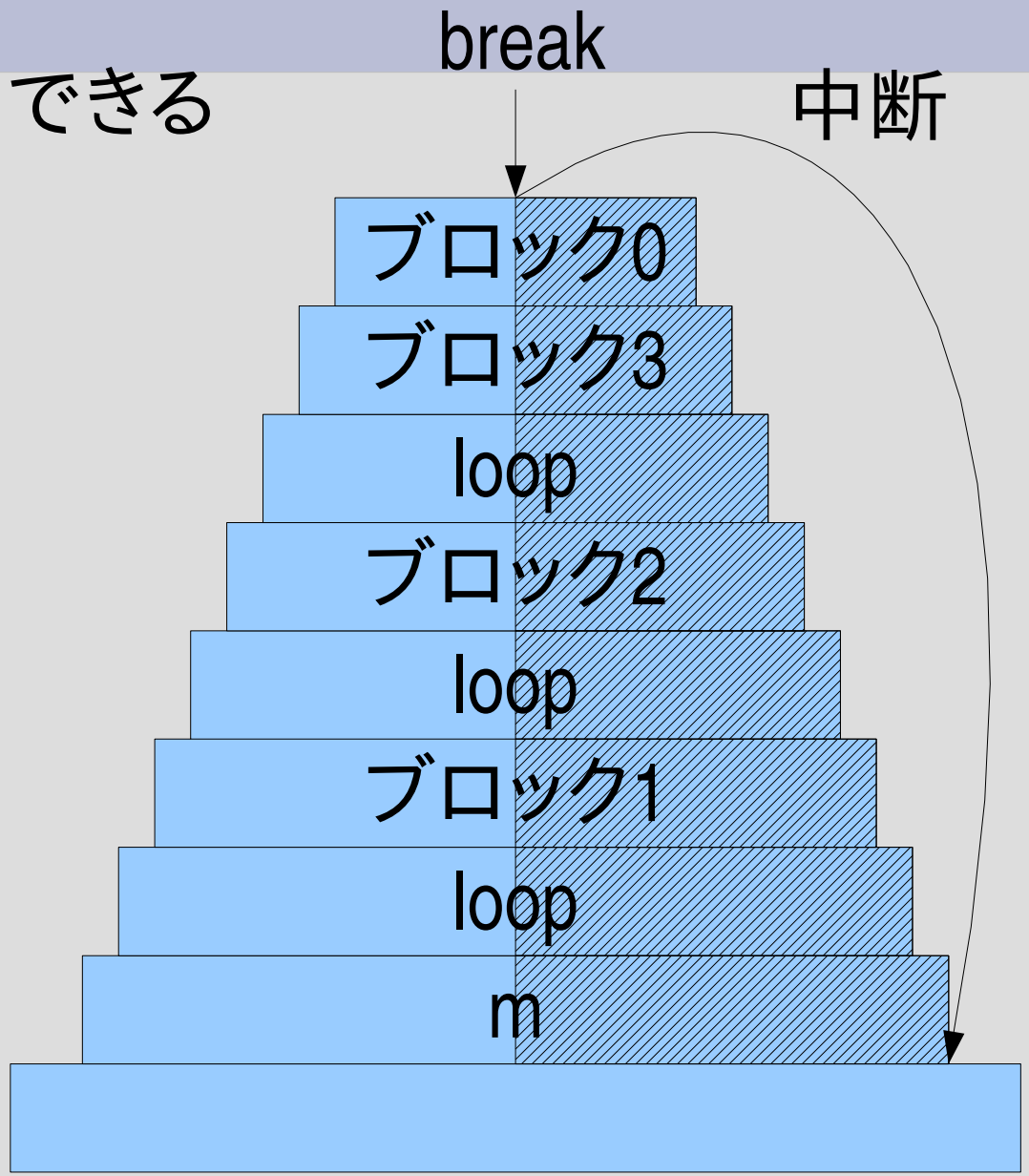
break の動作



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
```



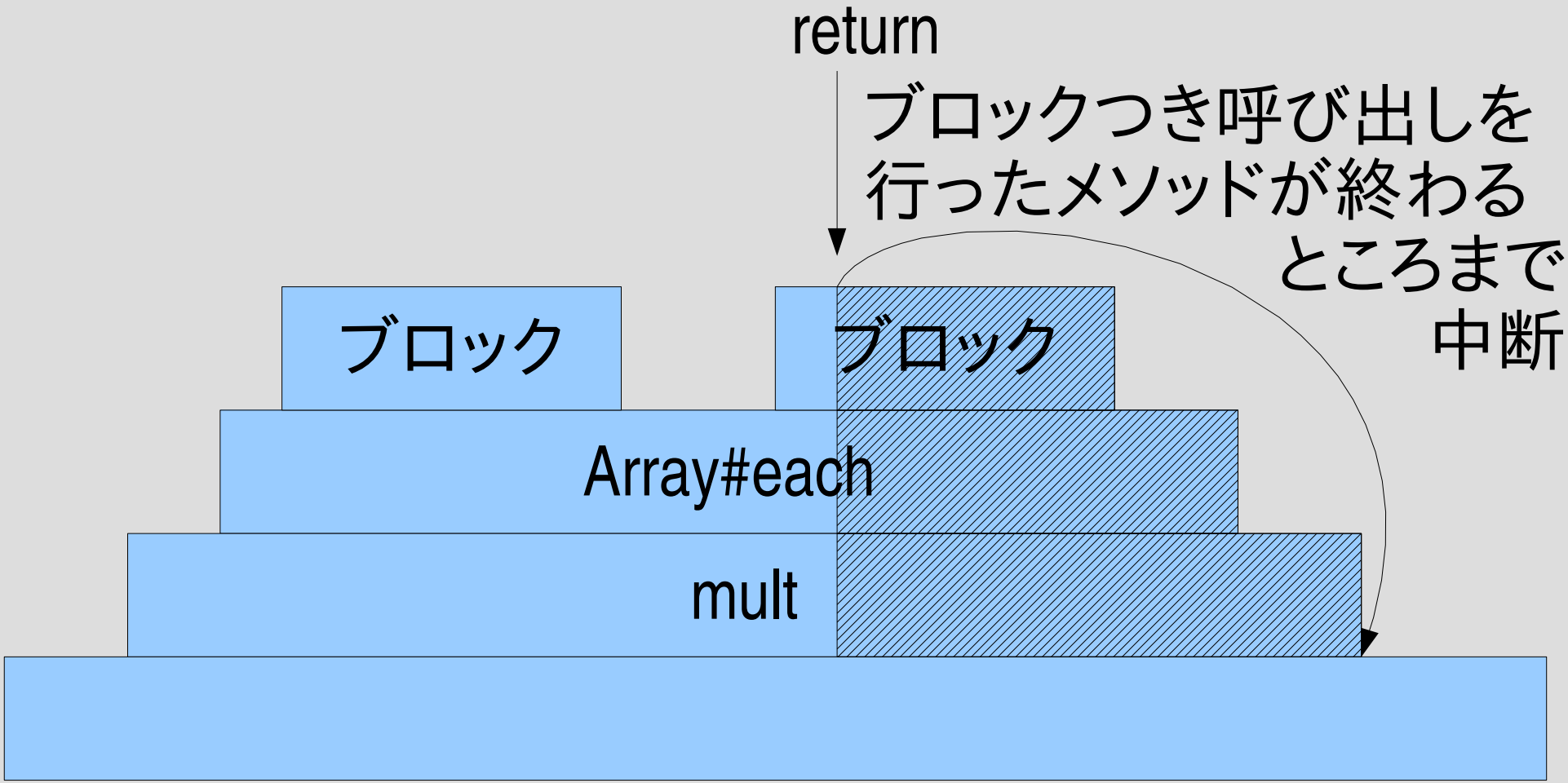
return

- 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

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
  ```
  ```

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
  succ = lambda {|v| v + 1 }
  m(&succ)          #=> 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

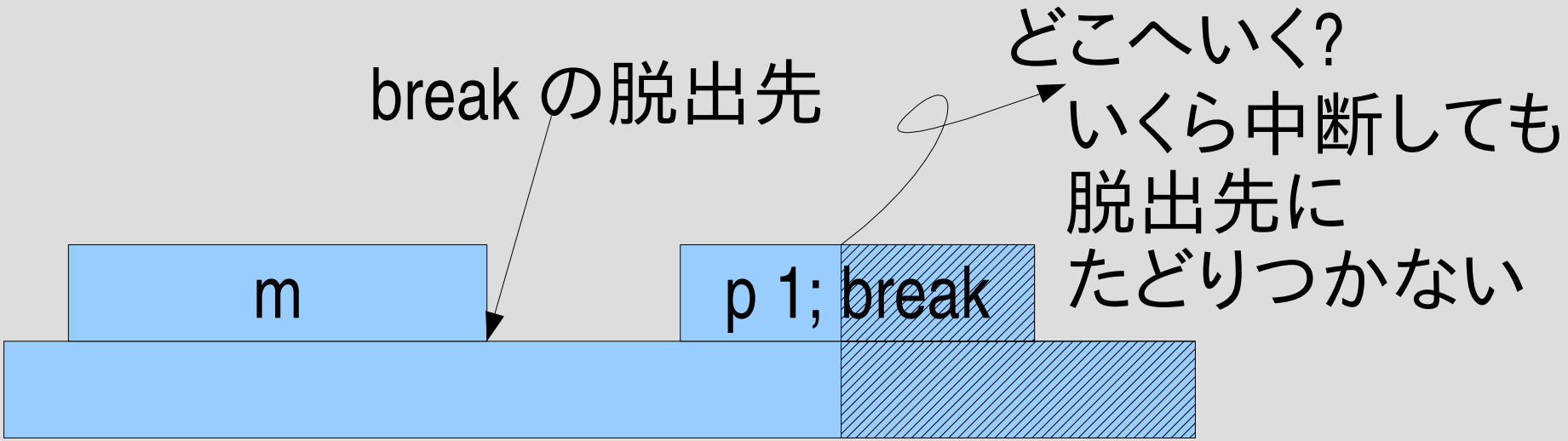
- ```
  def m
    p yield(10)
  end
  def n(&b)
    m(&b)
  end
  n {|v| v * 2 }      #=> 20
  ```
  ```

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

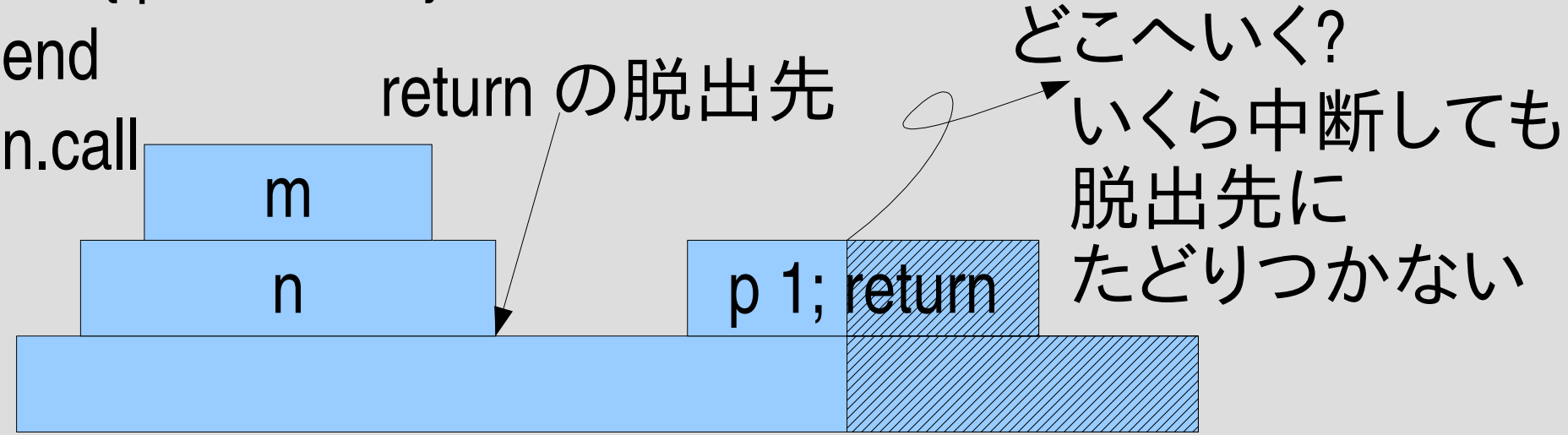
break 先が終わっている

- 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

- map と同様に要素にブロックを適用するが、フラットとは限らずネストしているかもしれない配列を扱うメソッド treemap を定義せよ
- def treemap(obj) ... end
- ユニットテストを提供するので、実装したらテストして確認すること
- ∠切 2006-06-27 16:20
- IT's class
- 拡張子が txt なテキストファイルを望む

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