

テキスト処理 第6回 (2007-06-12)

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-2007/`

今日の内容

- Proc (lambda)
- ブロック
- レポート

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 などというクロージャ
- コード + 変数などの状態(環境)

λ

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 }'
```

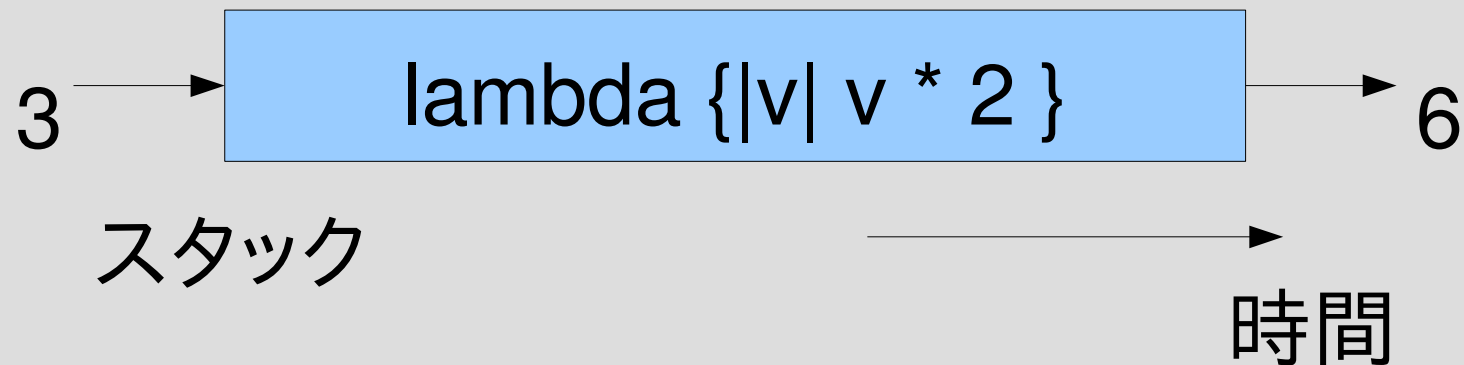
```
#<Proc:0xb7e1c870@-e:1>
```

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
 - メソッド
 - etc.
- 他の言語では Java のクラスとか

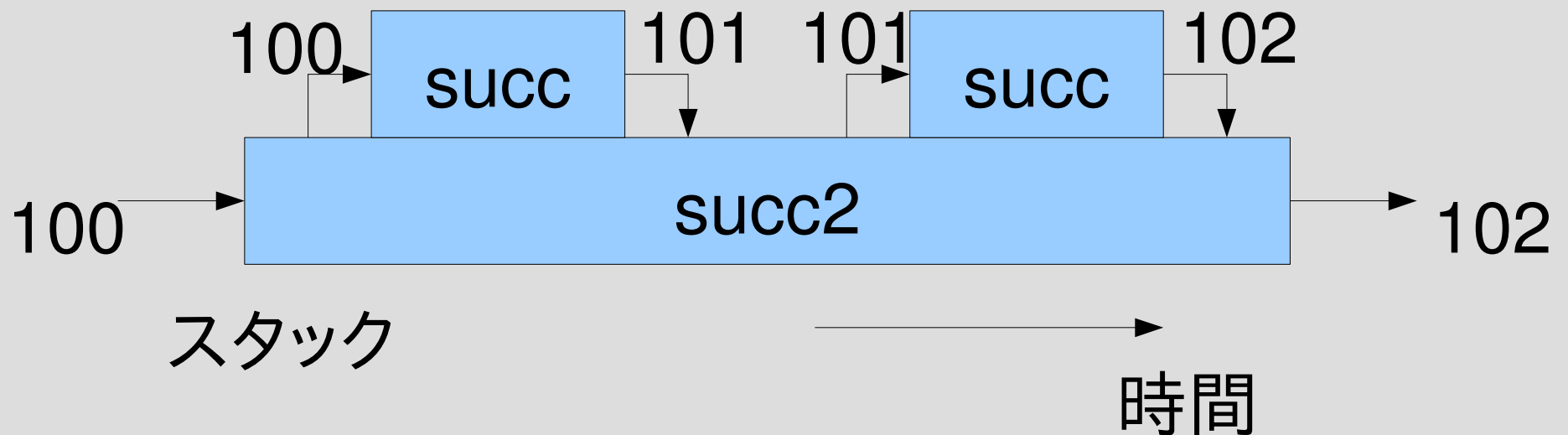
後者関数: succ

- 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`
- `p succ.call(2)` `#=> 3`
- `p succ.call(1000)` `#=> 1001`

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

succ2

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



Proc を使った map

```
def map(ary, f)
```

```
  result = []
```

```
  ary.each {|v|
```

```
    result << f.call(v)
```

```
  }
```

```
  result
```

```
end
```

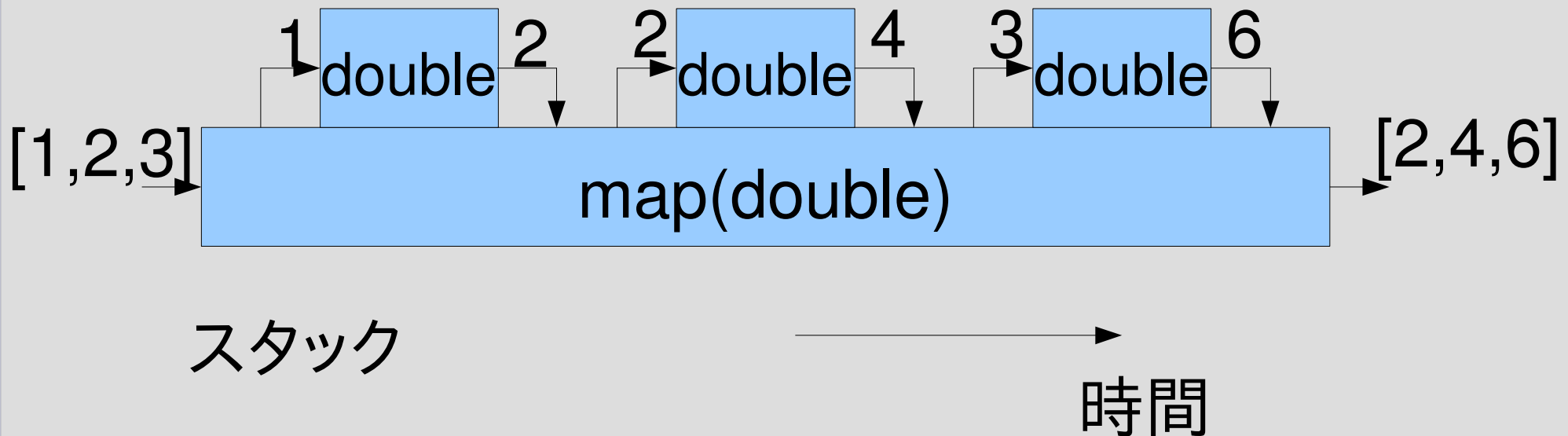
```
p map([1,2,3], lambda {|v| v * 2 })      #=> [2,4,6]
```

```
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 }
```

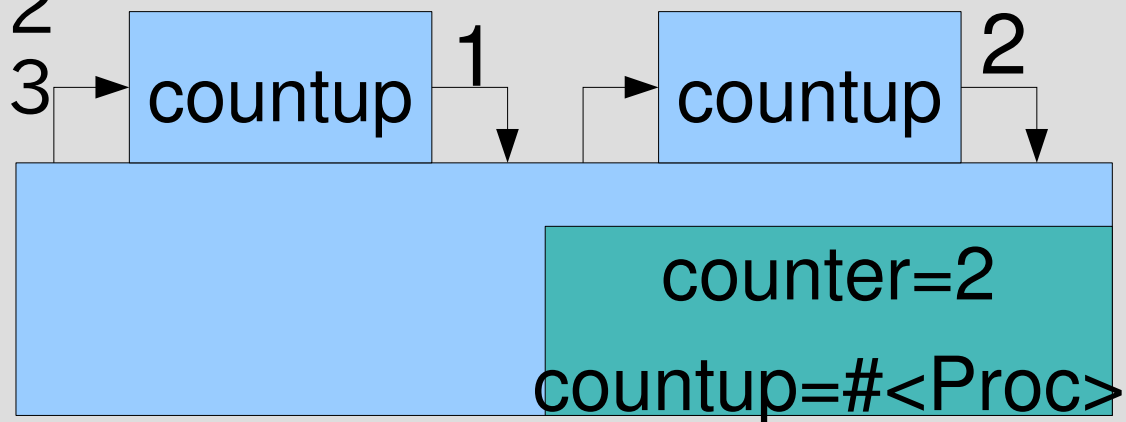
外部の変数アクセス

- Proc は外部の変数を参照・変更できる

```
counter = 0
countup = lambda {
  counter += 1
}
```

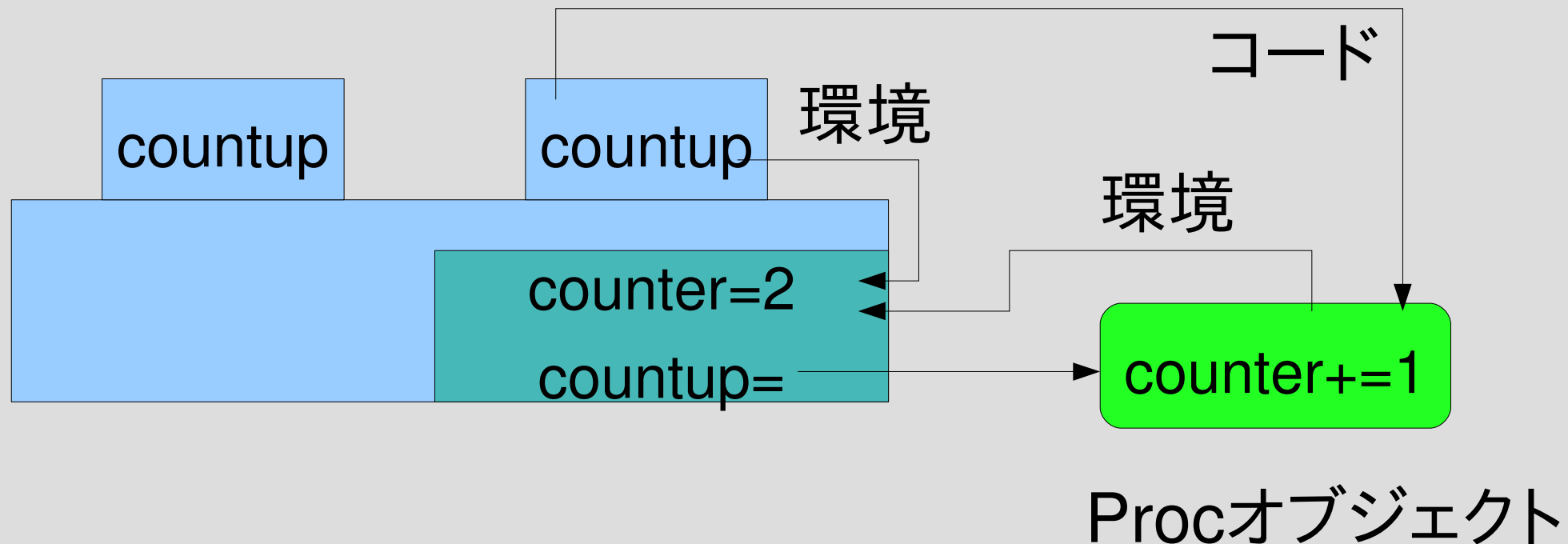
```
p countup.call #=> 1
p countup.call #=> 2
p countup.call #=> 3
```

変数はスタックフレーム
生成時に作成される



lambda = コード + 環境

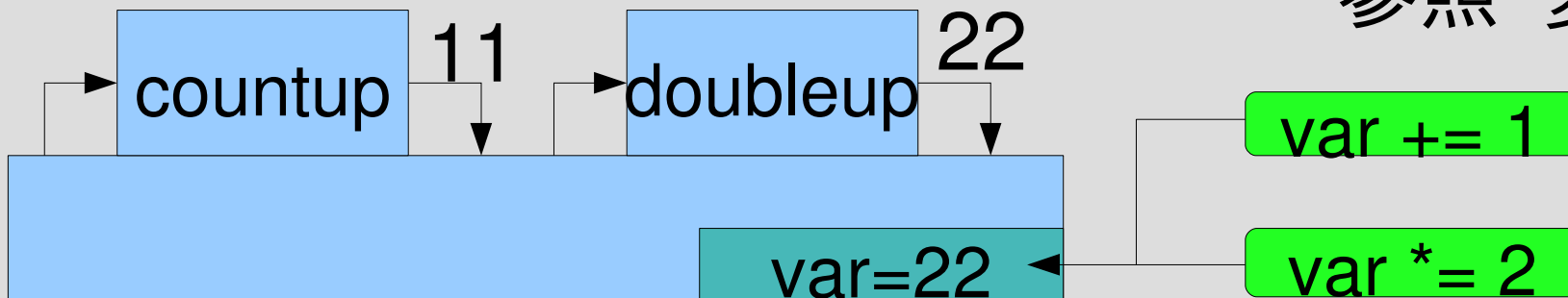
- Proc が評価される文脈を環境という: 変数, self, etc.
- 外部の変数へのアクセスは環境へのアクセス



変数の共有

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

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



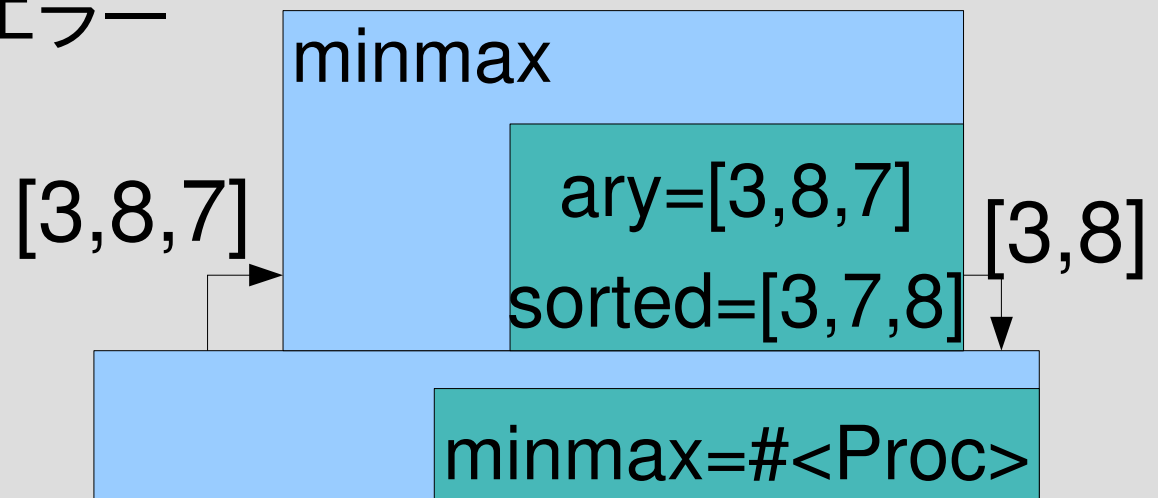
ブロックローカル変数

- 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]]  
}
```

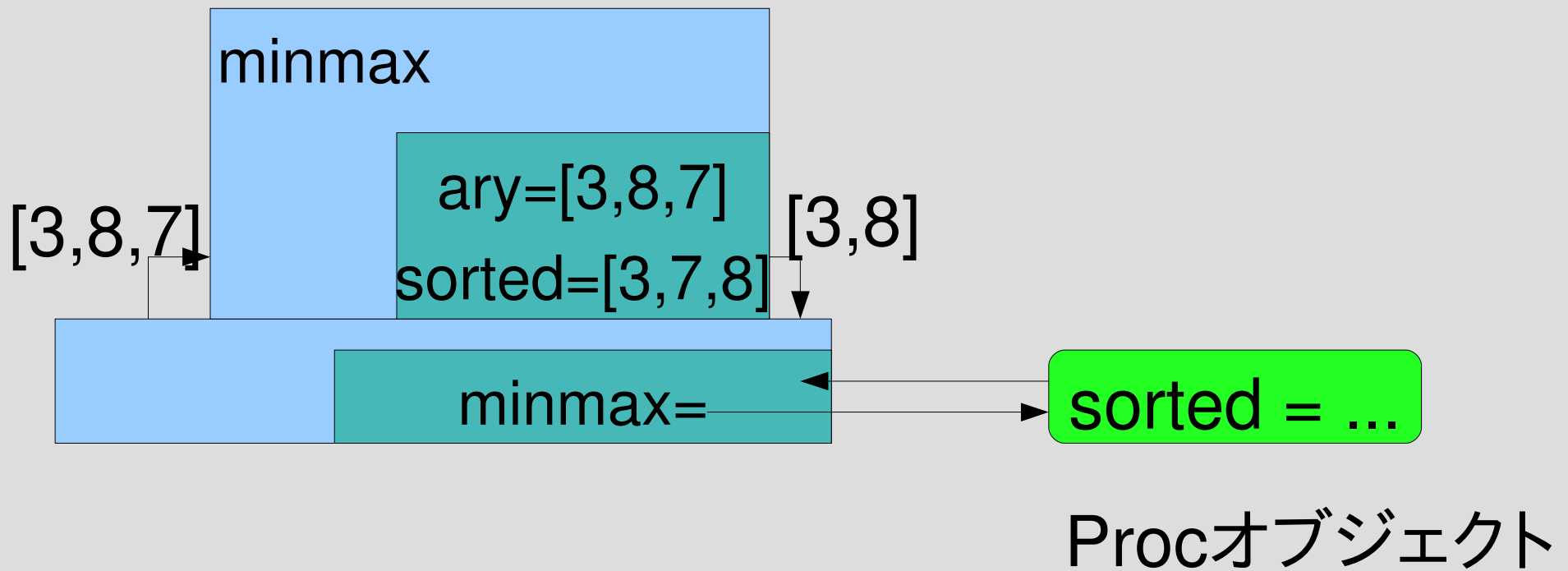
```
p minmax.call([3,8,7]) #=> [3,8]
```

```
p sorted          #=> エラー
```

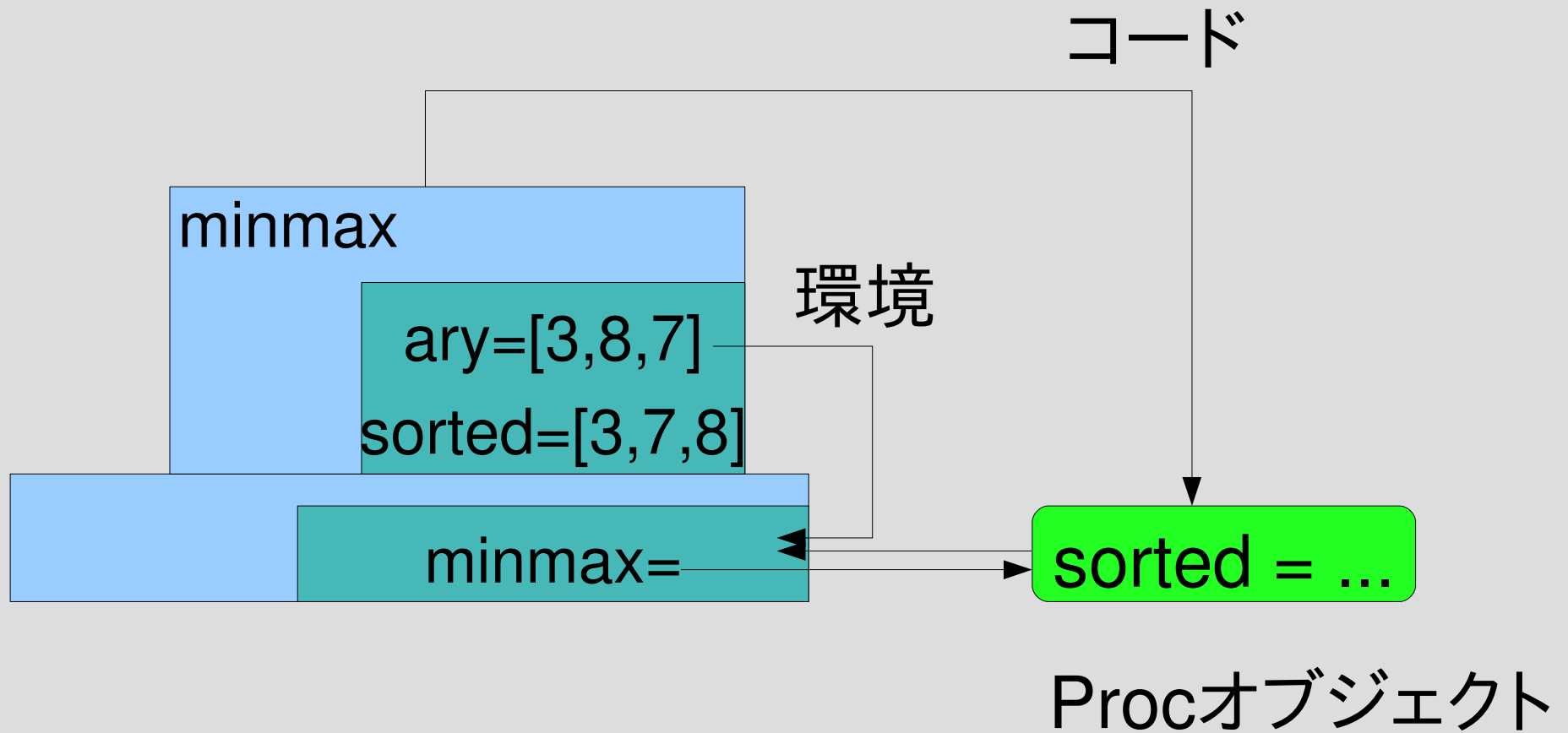


ブロックローカル変数

- 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

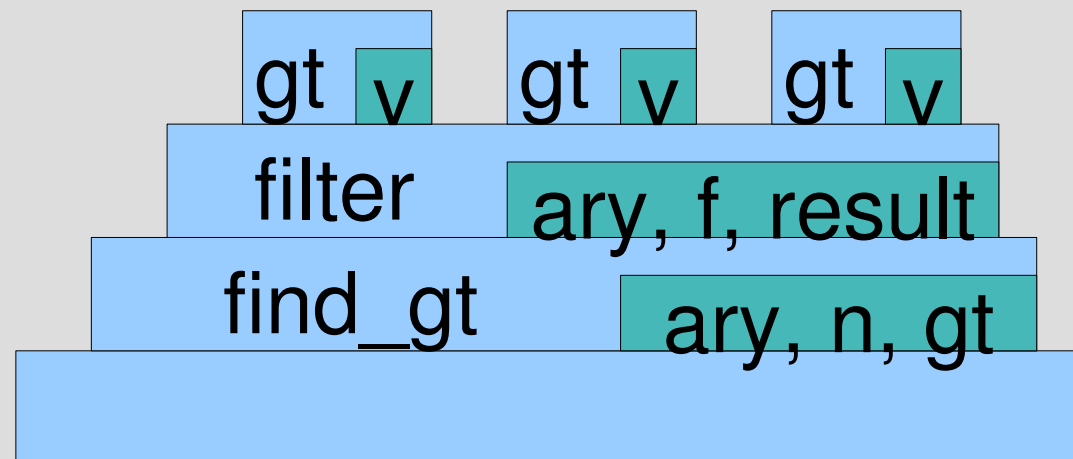
```
def filter(ary, f)
  result = []
  ary.each {|v| result << v if f.call(v) }
  result
end
```

ary から条件に
あったものだけを
取り出す

end 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

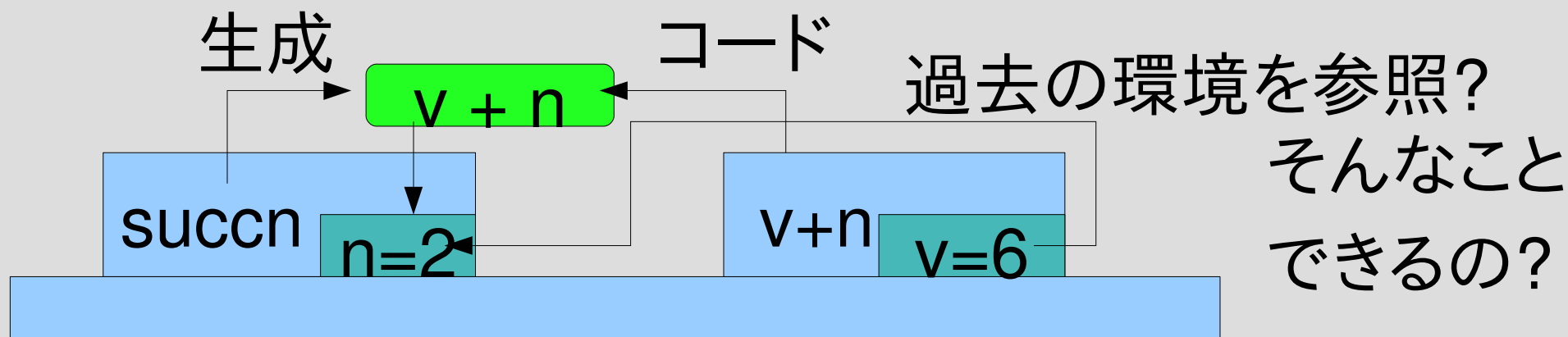
- `gt = lambda { |v| v > n }` は外側の `n` を参照している
- `gt` から参照される外側の環境は `find_gt` のもの
 - `gt` の Proc オブジェクトに記録されていたのが `find_gt` の環境だから
- `filter` の環境は `gt` からは見えない



Proc を返す

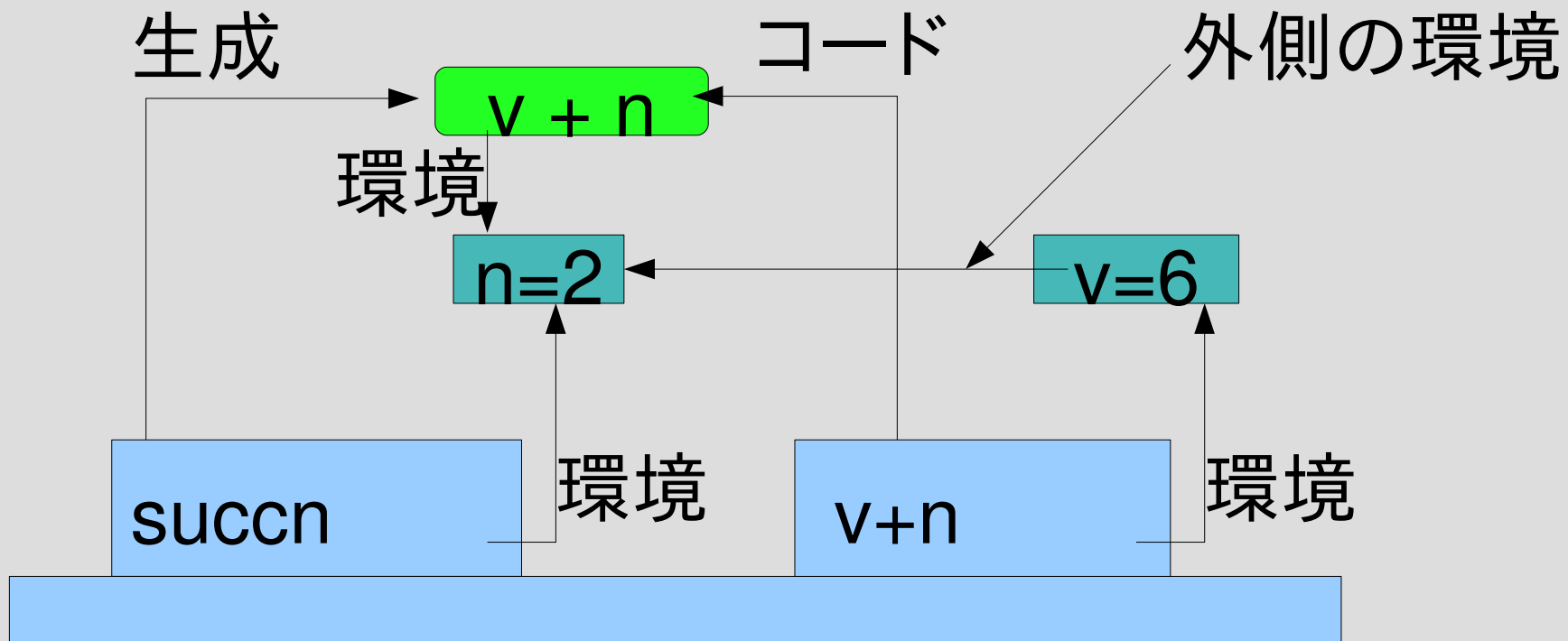
- あるメソッドで Proc を生成して返すことができる

```
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

ブロックと Proc

- ブロックと Proc は相互に変換できる (後述)
- ブロック = lambda + 脱出先
- 脱出には next, break, return の 3種類ある

Proc を使った map

```
def map(ary, f)
```

```
  result = []
```

```
  ary.each {|v|
```

```
    result << f.call(v)
```

```
  }
```

```
  result
```

```
end
```

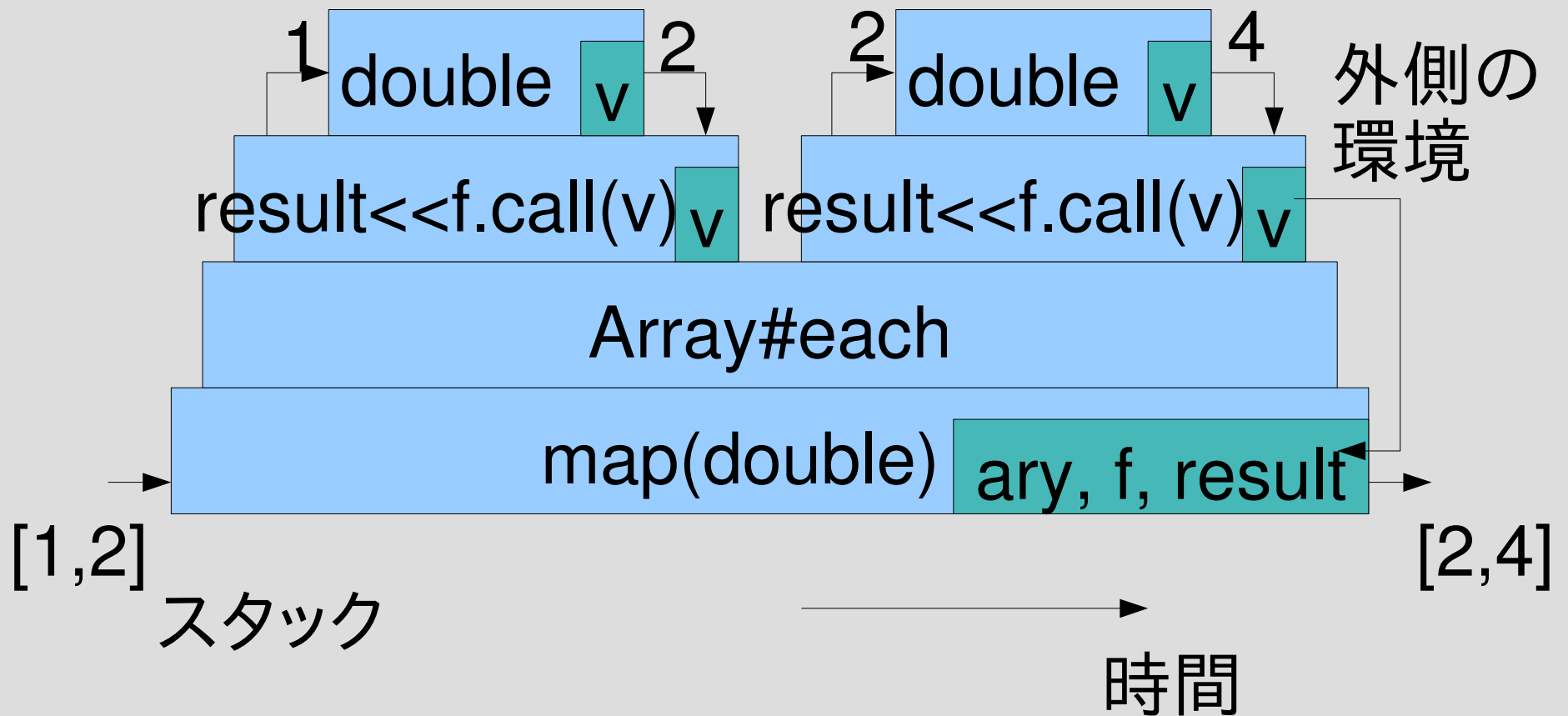
```
double = lambda {|v| v * 2 }
```

```
p map([1,2], double)
```

Array#each で
ブロックを使っている

#=> [2,4]

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

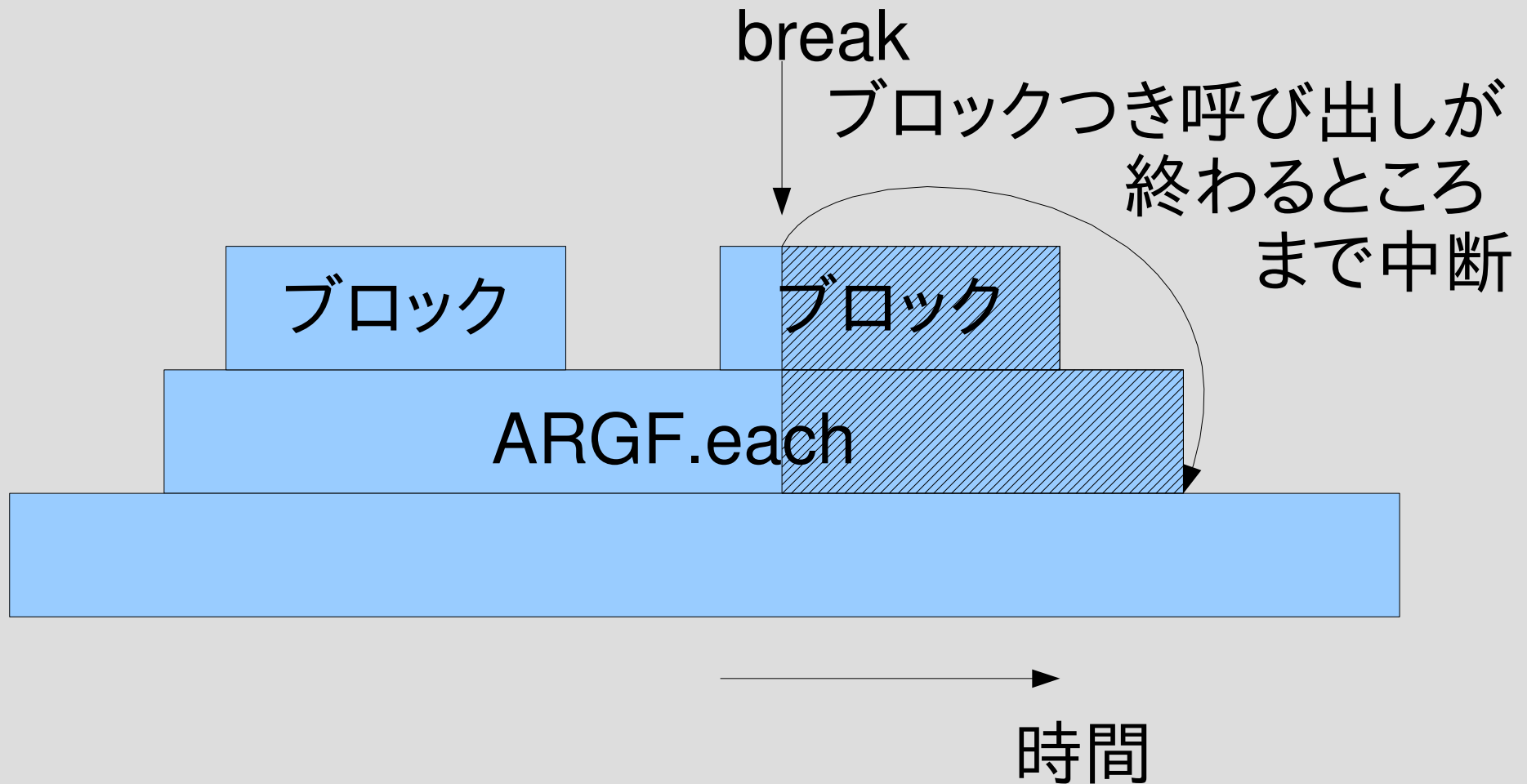
- break はブロックつき呼び出しからの脱出
- ループを止めるのによく使う
- C の break に対応する

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

}

↓
ARGF.each が終わる所

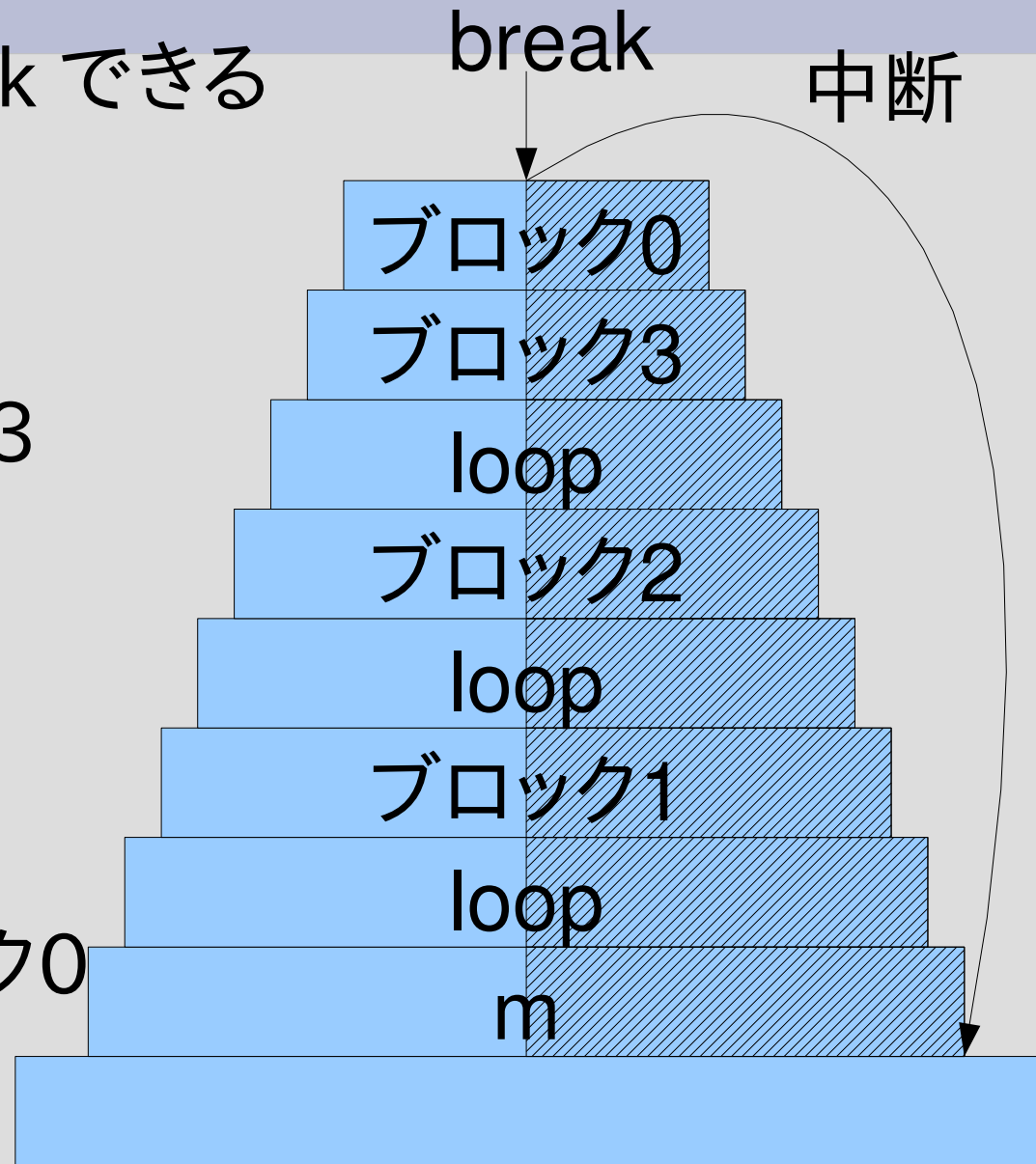
break の動作



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
```

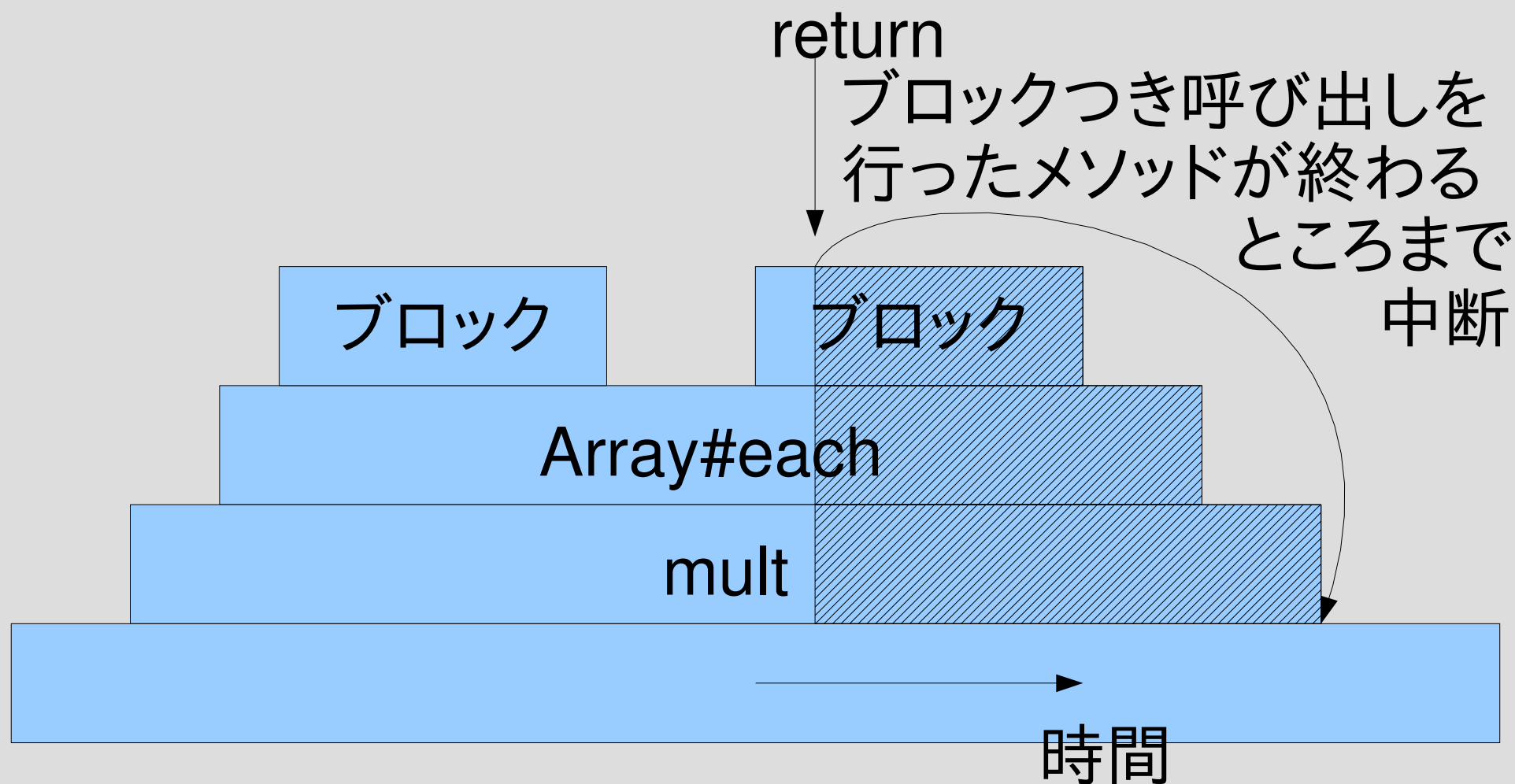
```
  }
```

```
  result
```

```
end
```

メソッドが終わる所

return の動作



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

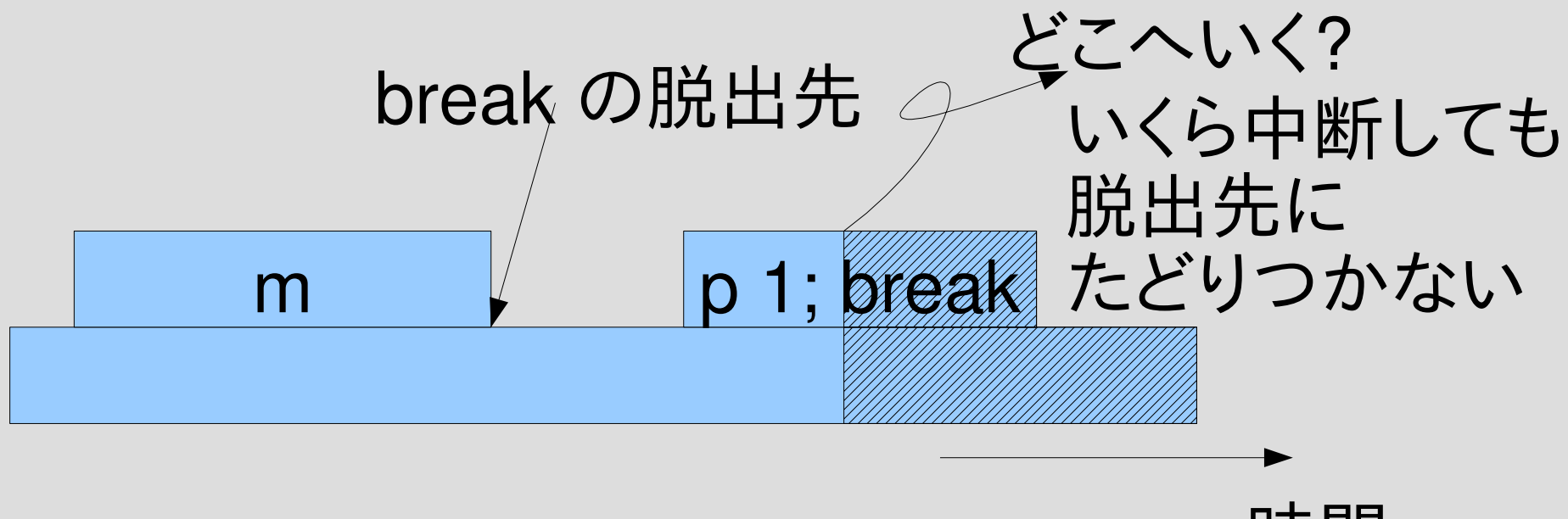
- & を使って、与えられたブロックを他のメソッドにそのまま渡すことができる
- ```
def m
  p yield(10)
end
def n(&b)
  m(&b)
end
n { |v| v * 2 }      #=> 20
```

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 の脱出先を乗り越えることができる
- 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

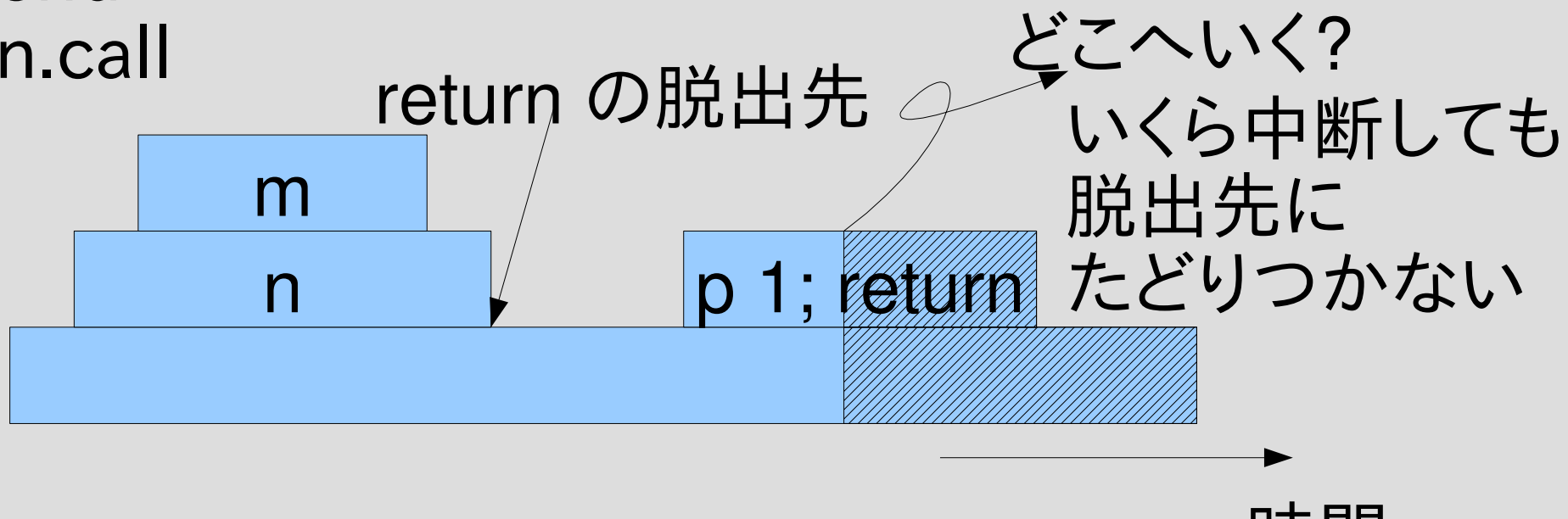
break 先が終わっている

- ```
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

- ネストしているかもしれない配列を受け取り、各要素についてブロックを適用し、真を返した最初の要素を返す `treefind` を作れ。真を返す要素がなければ `nil` を返すものとする
- `def treefind(ary) ... end`
- 再帰をつかうこと。
- `flatten` の類は禁止
- ✖切 2007-06-19 16:20
- HIPLUS
- 拡張子が `txt` なプレーンテキストで

# treefindの動作例

- `treefind([1,2,3,4]) { |v| v % 3 == 2 } #=> 2`
- `treefind([[1,2],[3,[4,5]],6]) { |v| v % 3 == 2 }  
#=> 2`
- `treefind(  
 ["ruby", "perl", "python", "php", "lisp"]) { |v|  
 /^p/ =~ v }  
#=> "perl"`

# 制限事項

- flatten の類は使わない
- 再帰する
- 例えば以下のような実装は禁止

```
def treefind(ary)
 a = ary.flatten.find_all {|v| yield v }
 a.empty? ? nil : a[0]
end
```

# ユニットテストを配布する

- 動作例で示したようなものを自動的に検査するための機構 (詳細はリファレンスマニュアル参照)
- test-treefind.rb
- test-treefind.rb に treefind の定義を書き加えて実行し、失敗が 0 になるまでがんばる

```
% ruby test-treefind.rb
```

```
Loaded suite test-treefind
```

```
Started
```

```
..
```

```
Finished in 0.000517 seconds.
```

```
2 tests, 4 assertions, 0 failures, 0 errors
```

# まとめ

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