プログラミング言語論第10回
局所状態変数と環境モデル
ー クロージャによる抽象ー

情報工学科 山本修身

代入モデルの復習

関数に引数を与えて評価する過程は代入モデルによって説明した.

```
(define  (sqr x) (* x x)

(define  (sum-of-square x y z)
  (+ (sqr x) (sqr y) (sqr z)))

(define (vec-length x y z)
  (sqrt (sum-of-square x y z)))
```

このモデルでは, 評価するときに出現した変数にその値を代入する. 通常のプログラムであれば, これによって, 動作を理解することができる.

```
(vec-length 1 2 3)
= (sqrt (sum-fo-square 1 2 3))
= (sqrt (+ (sqr 1) (sqr 2) (sqr 3)))
= (sqrt (+ (* 1 1) (* 2 2) (* 3 3)))
= (sqrt (+ 1 4 9))
= (sqrt 14) = 3.7416573867739413
```

局所状態をもつオブジェクト (1)

銀行の口座の状態を表現する関数を生成する関数を考える．以下のようなプログラムを考える．

```
(define (new-account name)
  (let ((balance 0))
    (lambda (message . args)
      (cond ((eq? message 'withdraw)
             (set! balance (- balance (car args)))
             balance)
            ((eq? message 'save)
             (set! balance (+ balance (car args)))
             balance)
            ((eq? message 'balance)
             balance)
            ((eq? message 'name)
             name)
            (else "Unknown command.")))))
```

局所状態をもつオブジェクト (2)

この関数を用いて見かけ上関数となっている実体を作ってみる．後で説明するように代入モデルではここでの動作は説明できない．

```
(define my-acc (new-account 'osami))
(my-acc 'save 200)

(define (disp-account mesg)
  (display mesg)
  (display (my-acc 'balance))
  (newline))

(disp-account "start: ")
(my-acc 'save 340)
(disp-account "step1: ")
(my-acc 'withdraw 100)
(disp-account "step2: ")
(my-acc 'save 50)
(disp-account "step3: ")
(display "owner: ")
(display (my-acc 'name))
(newline)
```

```
OMacBook:yama506> kawa bank.scm
start: 200
step1: 540
step2: 440
step3: 490
owner: osami
OMacBook:yama507>
```

局所状態をもつオブジェクト (3)

前のスライドで示したような動作をするのは，今までの説明を元にして良く考えるとおかしい.

代入モデルで考えると．．．

```
(new-account 'osami)

= (let ((balance 0))

    (lambda (message . args)
      (cond ((eq? message 'withdraw)
             (set! balance (- balance (car args)))
             balance)
            ((eq? message 'save)
             (set! balance (+ balance (car args)))
             balance)
            ((eq? message 'balance)
             balance)
            ((eq? message 'name)
             'osami)
            (else "Unknown command."))))
```

局所状態をもつオブジェクト (4)

balanceを0に置き換えるとプログラムは意味をなさなくなってしまう.

```
(new-account 'osami)

= (lambda (message . args)

      (cond ((eq? message 'withdraw)
             (set! 0 (- 0 (car args)))
             0)
            ((eq? message 'save)
             (set! 0 (+ 0 (car args)))
             balance)
            ((eq? message 'balance)
             balance)
            ((eq? message 'name)
             'osami)
            (else "Unknown command.")))
```

局所状態をもつオブジェクト (5)

オブジェクトとして機能する関数（内部のlambdaを評価したもの）の外側にいくつかの変数があり，この変数の値が確定した状態でlambdaが評価される．ここまでは以前考えた変数の代入モデルで説明することが可能であるが，これらの変数を書き換えて使えることは説明できない．

オブジェクトの内部状態の変数

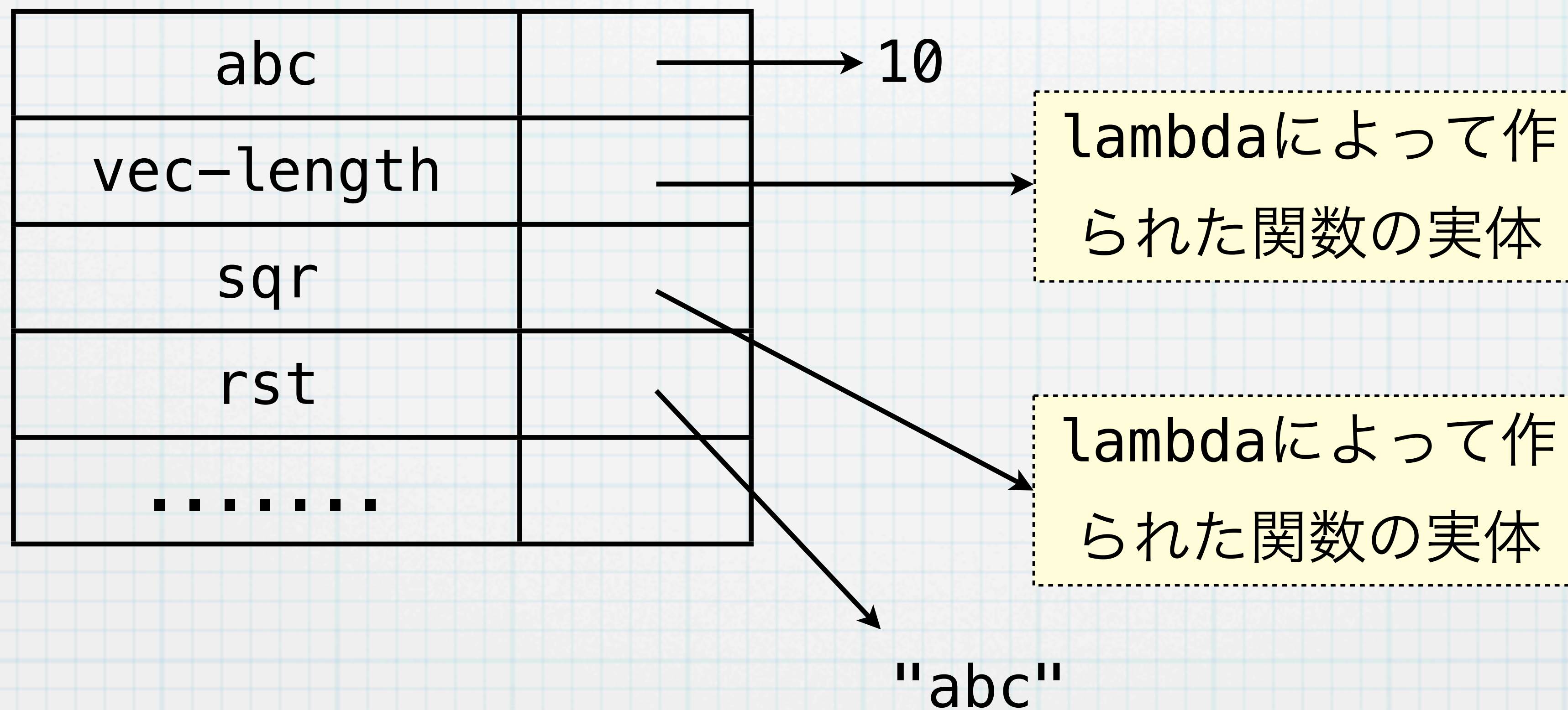
```
(define (new-account name)
  (let ((balance 0))
    (lambda (message . args)
      (cond ((eq? message 'withdraw)
             (set! balance (- balance (car args)))
             balance)
            ((eq? message 'save)
             (set! balance (+ balance (car args)))
             balance)
            ((eq? message 'balance)
             balance)
            ((eq? message 'name)
             name)
            (else "Unknown command.")))))
```

オブジェクトの内部状態の変数

このプログラムが問題なく動くことを説明するには，別考え方を導入する必要がある．

環境モデル (1)

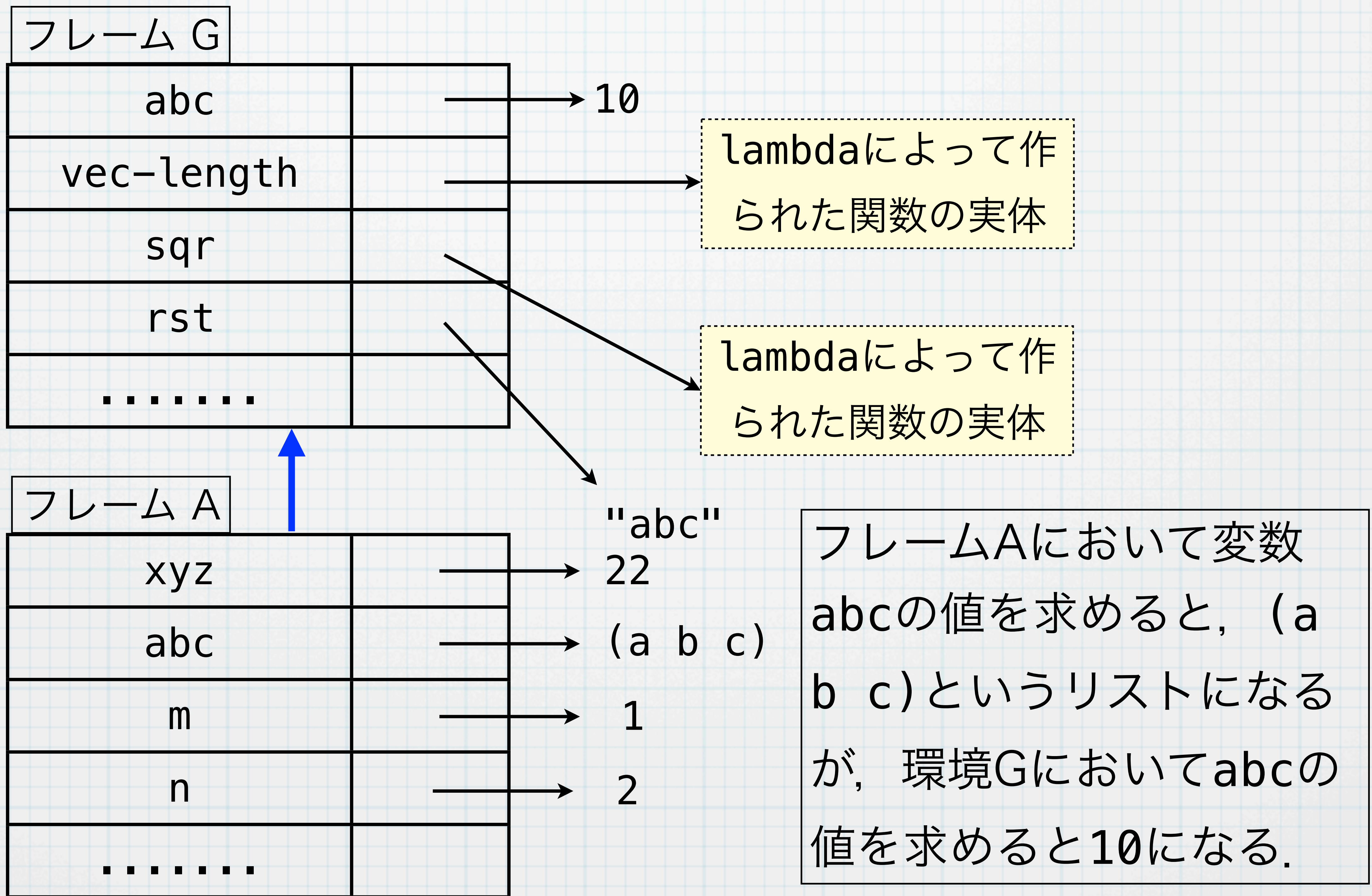
変数を参照する，lambdaを定義する，変数に値を代入するなどの操作を説明するには代入モデルではうまくいかない．まず，「フレーム」を定義する．**フレーム (frame)**とは，変数とその値をペアー（これを**束縛 (binding)**と呼ぶ）にして保存されている表のことである．



システム全体に一つの環境のみを用意すると，同じ名前の変数は1つしか定義できないことになる．

環境モデル (2)

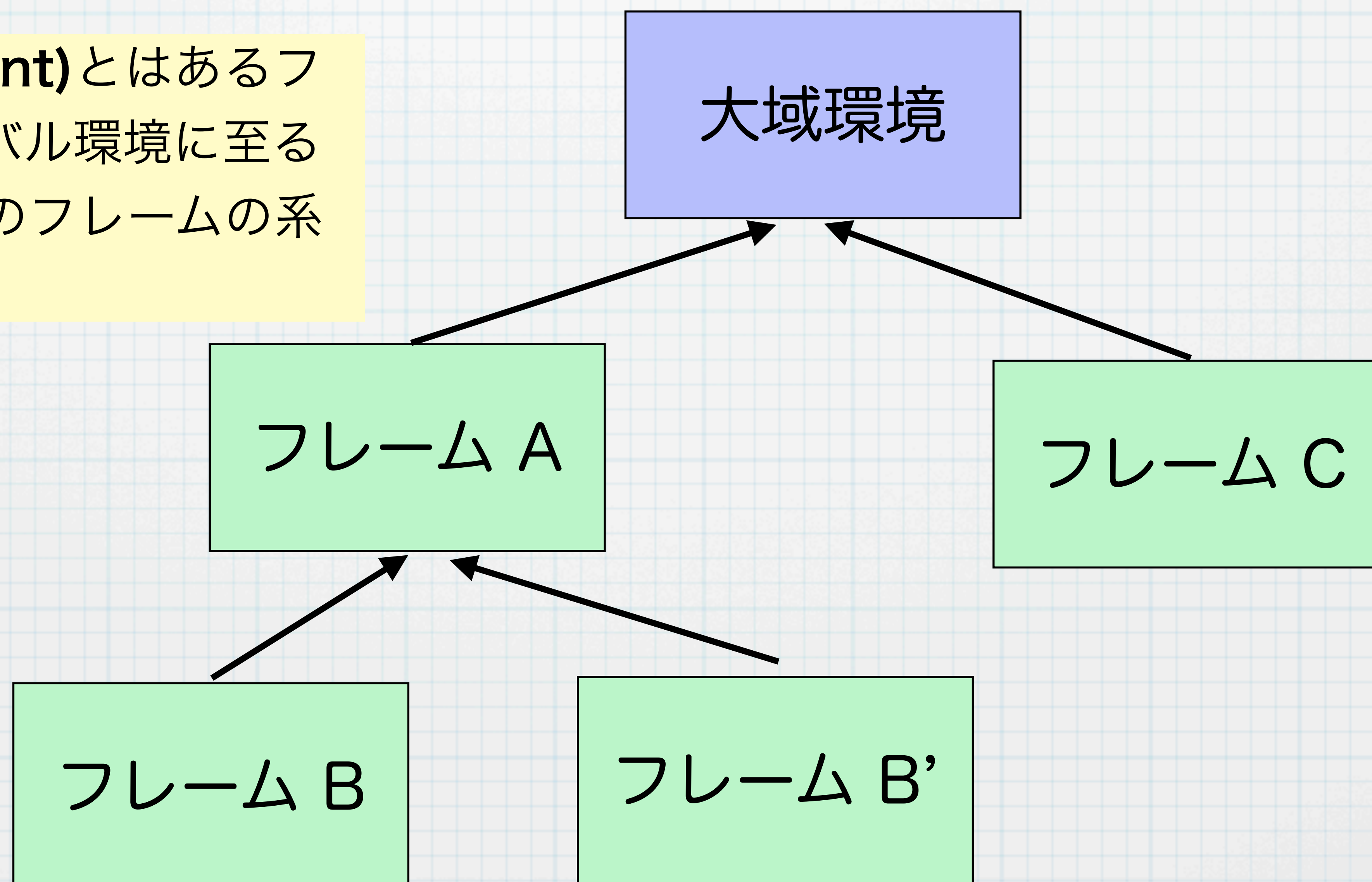
環境に局所性を持たせるために，フレームを数珠つなぎにする方法が考えられる．Schemeではフレームは数珠つなぎにして利用する．



環境モデル (3)

フレームは何重にも数珠つなぎにすることができる．また，あるフレームを複数のフレームが親フレームとして指すことができる．ただし，すべてのフレームについてその親フレームは一つしかない．また，最終的な親フレームのことを**グローバル環境（大域環境）**と呼ぶ．Schemeの場合，グローバル環境は一つしかない．

環境 (environment)とはあるフレームからグローバル環境に至るパスを辿ったときのフレームの系列のことである．



環境を操作するための関数

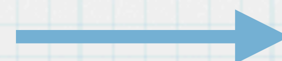
define: defineは現在式を評価しているフレームに新たなエントリーを作ってそこに値を結びつける．たとえば，

```
(define abc 234)
```

は現在のフレームに `(abc . 234)` という束縛を作る

set!: この関数は環境リストを辿って，与えられた名前の最初に見つかった束縛の値を書き換える．

```
(let ((a 234))
  (let ((b 345))
    (set! a 222))
  (display a)
  (newline))
```



```
222
```

※ letは新たな環境を生成する

lambda式が評価されるときに起こること (1)

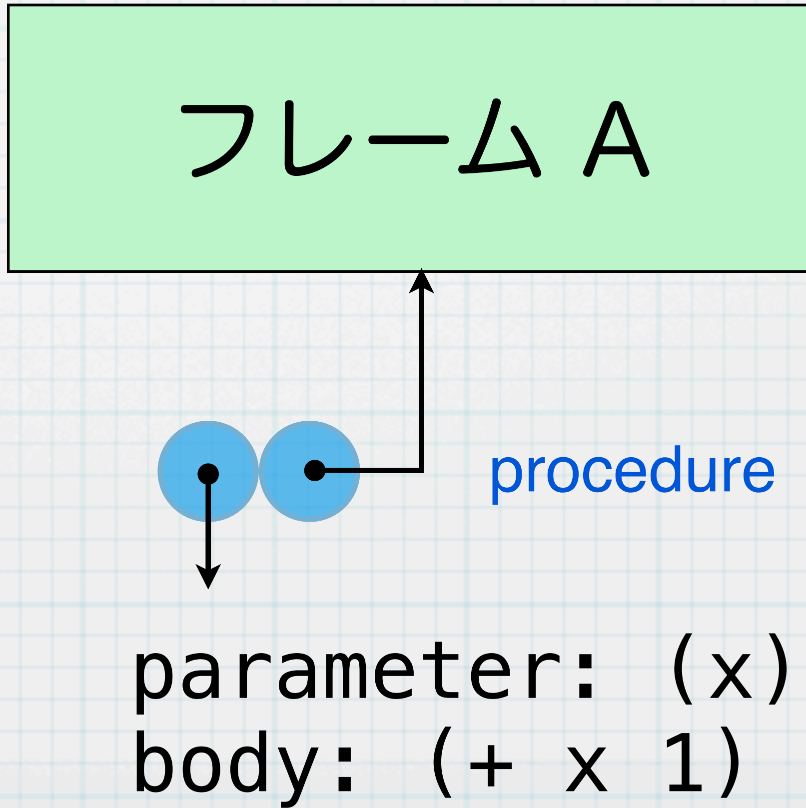
適当な環境において関数を定義する場合，結局その環境でlambda式が評価されることになる．評価されたlambda式の値は，値としての手続き (procedure) である．手続きは以下の値を内部にもつ．

- その手続きの引数列
- 手続きの本体（式）
- lambda式が評価されたときの環境

フレーム（環境） A で

```
(lambda (x) (+ x 1))
```

を評価する．



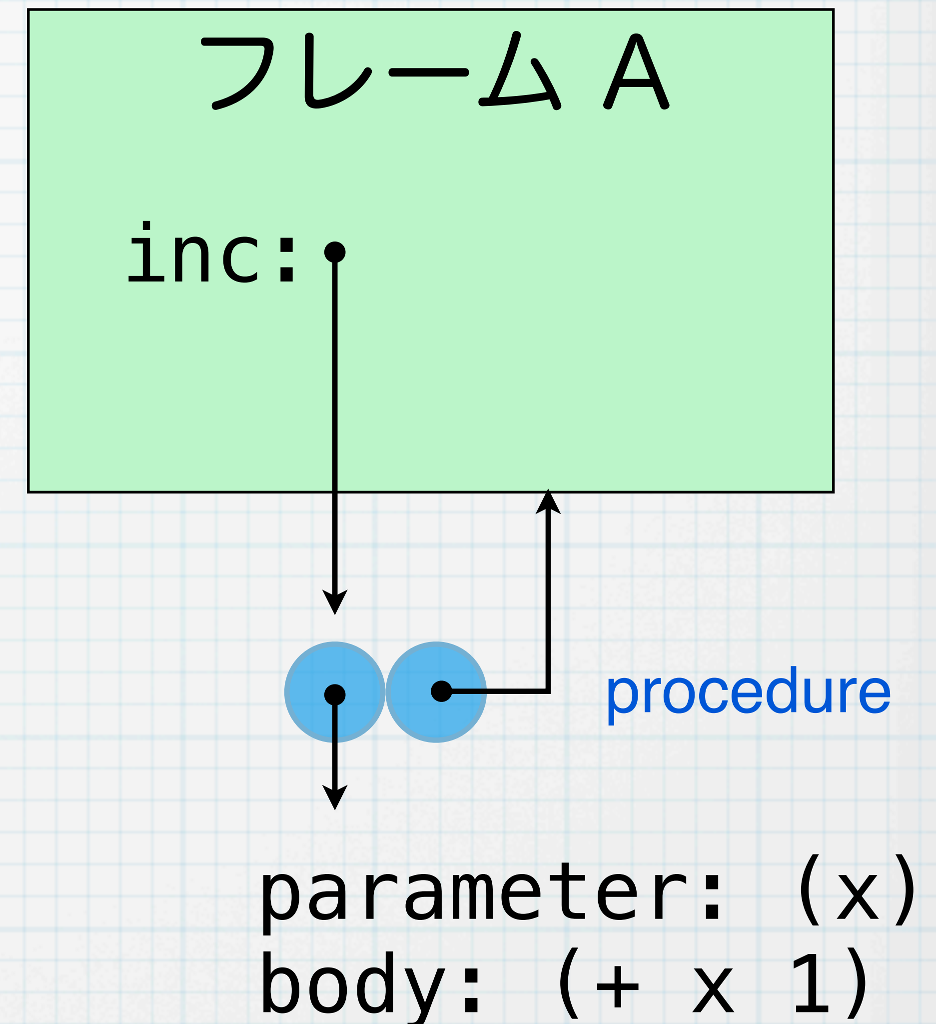
lambda式が評価されるときに起こること (2)

ある環境である関数を定義すると，以下のような構造になる．

```
(define (inc x) (+ x 1))
```

以下の式と等価

```
(define inc
        (lambda (x) (+ x 1)))
```



ここで重要なのは，lambdaを評価したとき，手続き内部に記録されるのは，lambdaが評価された環境（フレーム）であるということ．

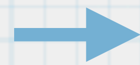
環境の下で式を評価する (1)

適当な環境 A 下で式を評価する (eval) ことは，以下のような行為である．ここでは限定された種類の式のみ考える．

与えられた式 exp が

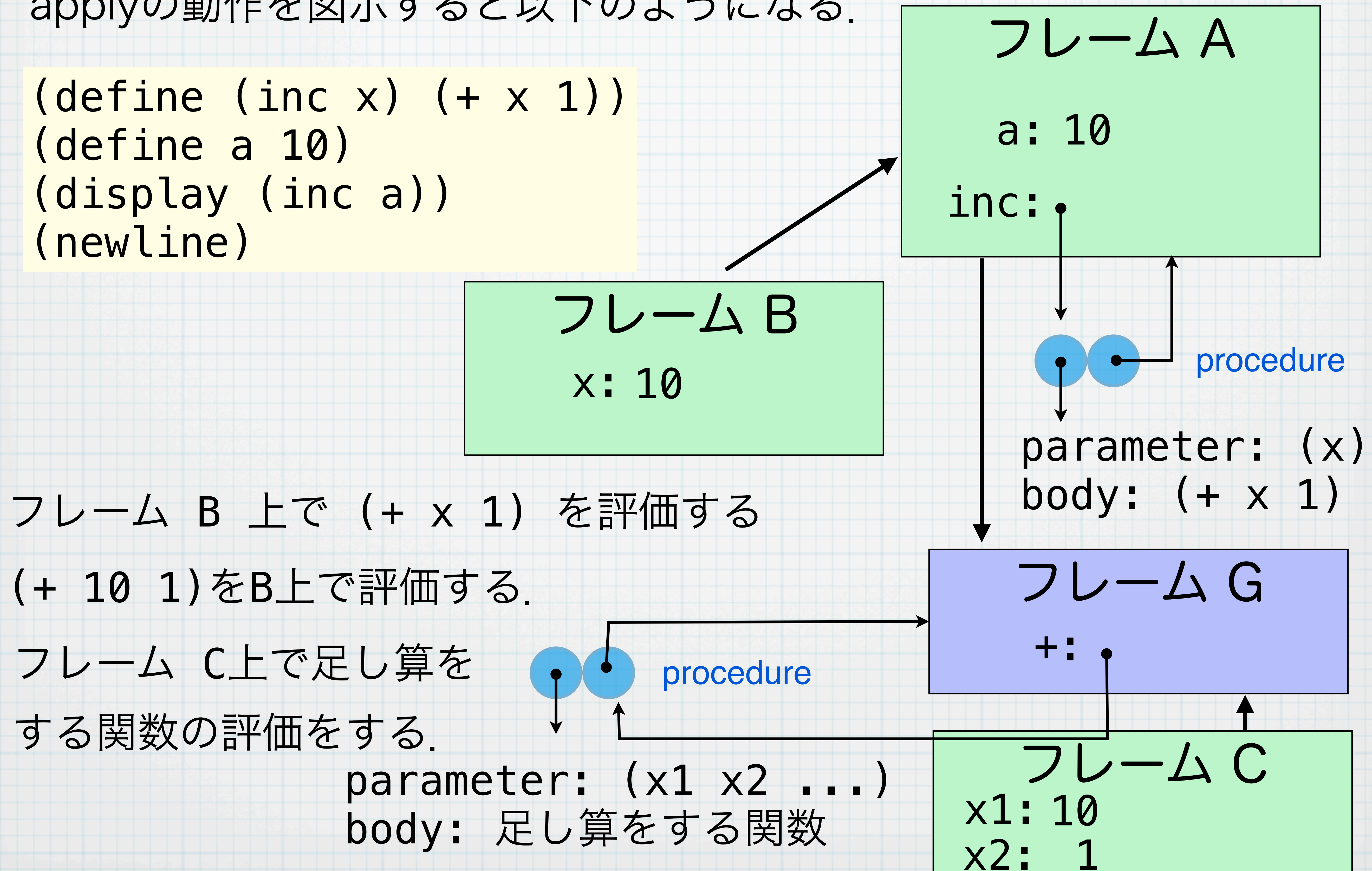
- ✦ **数** → 数を返す
- ✦ **文字列** → その文字列を返す
- ✦ **シンボル** → 環境Aからスタートしてその値を見つけて返す
- ✦ **リスト** → すべての要素を**この環境 A** で評価する．
  最初の要素はprocedureであるはず．
  そうでなければエラーを返す．

この動作を
applyと呼ぶ



このprocedureの持っている環境 B に
子環境 B’ を作って B’ に引数とその値の
ペアーを登録する．
環境 B’ 上でprocudureの本体を評価する．

環境の下で式を評価する (2)



関数の中で定義された関数について

関数内部で定義された関数が実行される様子.

```
(define (sample n)
  (define m 20)
  (define (lfunc x)
    (if (< x 0) (- x) x))
  (lfunc (- m (* m m))))

(display (sample 20))
(newline)
```

フレーム G
sample:

procedure

parameter: (n)
body: (define...)

フレーム B
x: -380

フレーム A
n: 20
m: 20
lfunc:

↑この環境で (if (< x 0) (- x) x)を評価する

procedure

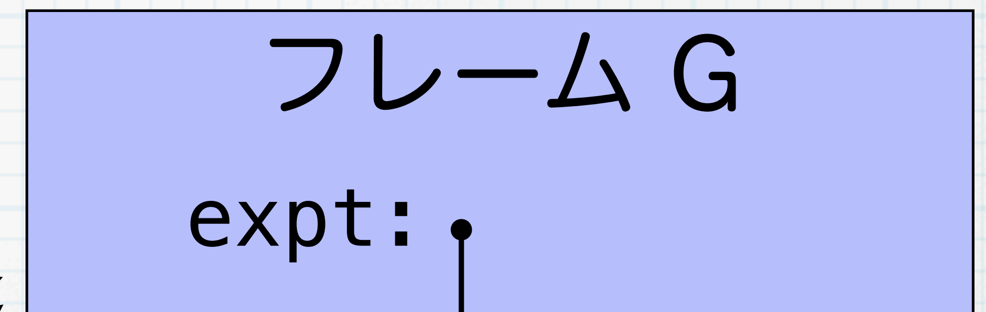
parameter: (x)
body: (if (< x 0) (- x) x)

再帰定義された関数が実行される様子

関数の再帰呼び出しが実行されるときは，いくつものフレームが生成される．

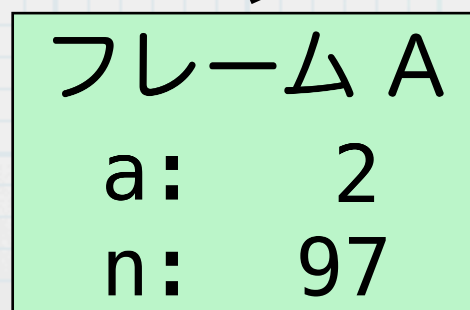
```
(define (expt a n)
  (if (zero? n) 1
      (* a (expt a (- n 1)))))
(display (expt 2 100))
(newline)
```

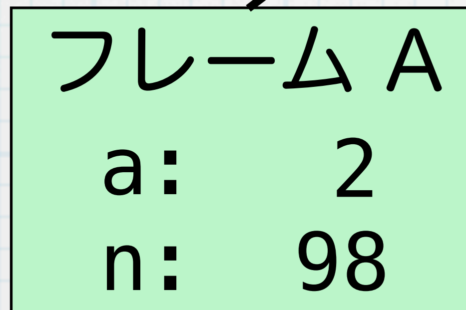
…

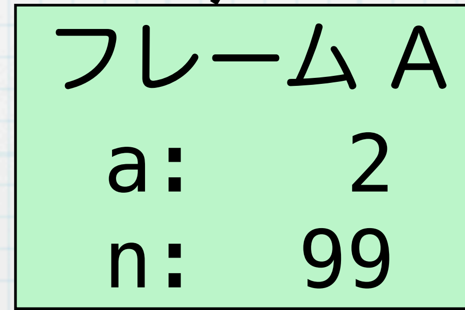


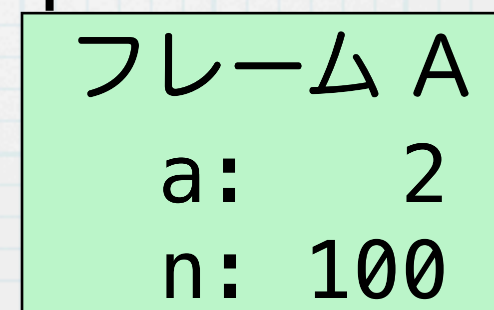
parameter: (a n)
body: (if ...)

…









↑さらに，これら環境で (if ...) を評価する

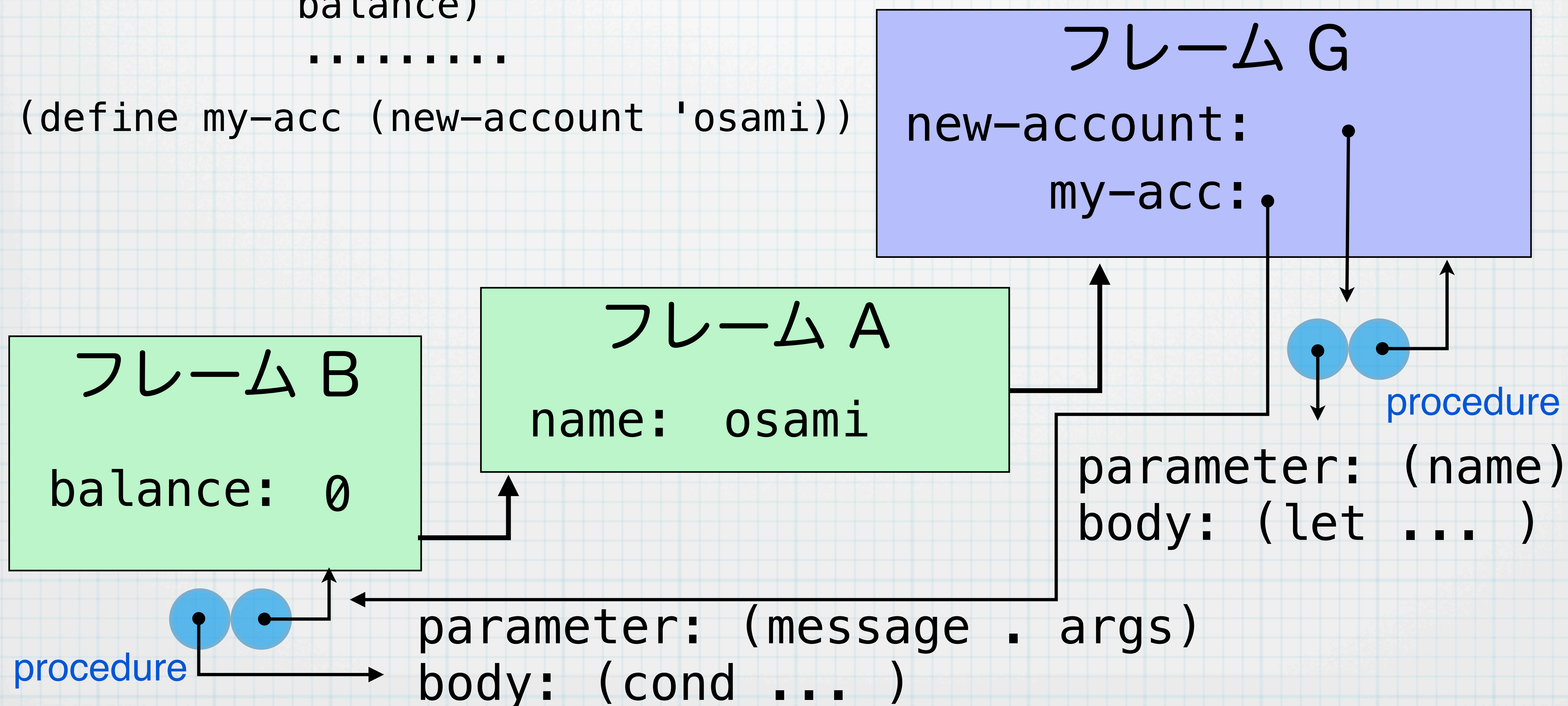
↑この環境で (if ...) を評価する

new-account の実行の様子

この講義の最初に示したオブジェクトの例について考えてみる.

環境モデルではbalanceは場所として実在するので，set!で内容を書き換えることができる.

```
(define (new-account name)
  (let ((balance 0))
    (lambda (message . args)
      (cond ((eq? message 'withdraw)
             (set! balance (- balance (car args)))
             balance)
             .........

 (define my-acc (new-account 'osami))
```



静的スコープルールと動的スコープルール

多くの言語はSchemeと同様にプログラムの入れ子構造が変数の見え方に反映される**静的スコープルール (lexical (static) scoping rule)**を採用している．これに対して，いくつかの言語では，**動的スコープルール (dynamic scoping rule)** を採用しているものがある．静的スコープを利用するプログラミング言語には，C/C++, Python, Java, Scheme, Rubyなどがある．動的スコープが使える言語としては，原始LISP, LOGO, Perl, Common LISPなどがある．

また，C/C++は関数の中に関数を定義することができず，ここで問題にするスコープの議論自体あまり意味がない．C/C++では大域環境と直接にそれを親環境とする局所環境があるだけである．このようなしくみを**平坦スコーピング（flat scoping)**と呼ぶことがある．

静的スコープ

静的スコープは，通常のSchemeのプログラミングを思い浮かべれば良い．このスコープはプログラムに書かれた入れ子の構造のとおり変数が見えるしくみであり，本日で説明したフレームによって実現される．

```
(define (vec-length p)
  (define EPS 1.0e-6)
  (define (sqr x) (* x x))
  (define (sqrt y)
    (define (sqrt-iter x)
      (if (<= (abs (- (sqr x) y)) EPS) x
          (sqrt-iter (- x (/ (- (sqr x) y) (* 2 x))))))
    (sqrt-iter y))
  (let ((p0 (car p))
        (p1 (cadr p))
        (p2 (caddr p)))
    (sqrt (+ (sqr p0) (sqr p1) (sqr p2)))))

(display (vec-length '(1.0 1.0 2.0)))
(newline)
```

special form

in global frame (level 0)

in frame level 1

in frames level 2, 3, ...

静的スコープにおける変数の遮蔽

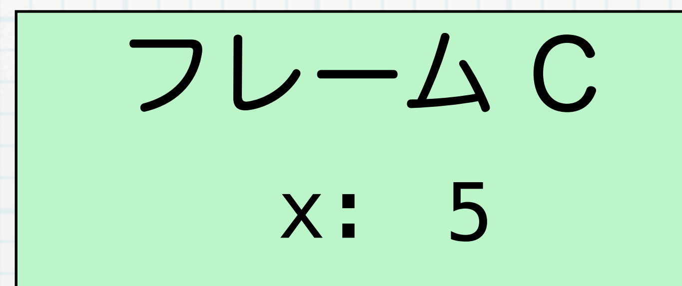
環境モデルでは現在の環境として指しているフレームからスタートして親フレームを順にグローバルフレームまで見て行き，**最初に見つかったバインディングを利用する．**したがって，その後に出て来るバインディングを利用することができなくなる．

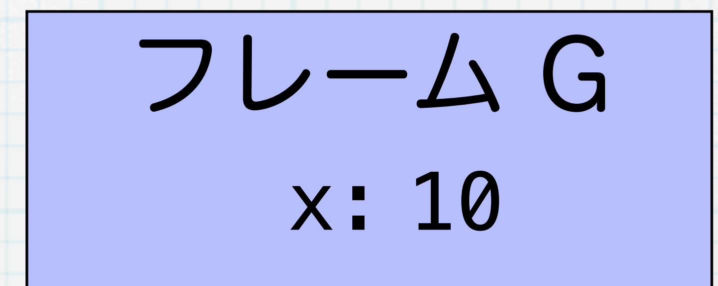
```
(define x 10)

(let ((x 2))
  (let ((y 3))
    (let ((x 5))
      (display x)
      (newline))))
```

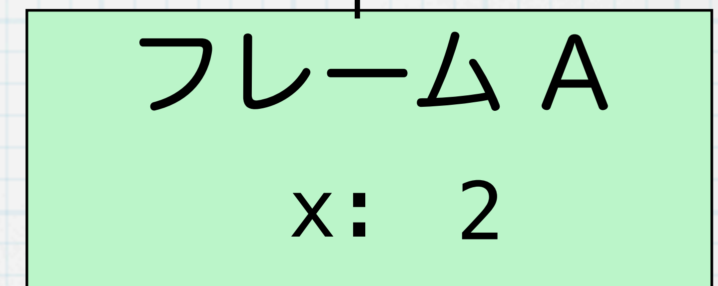
←5を出力する

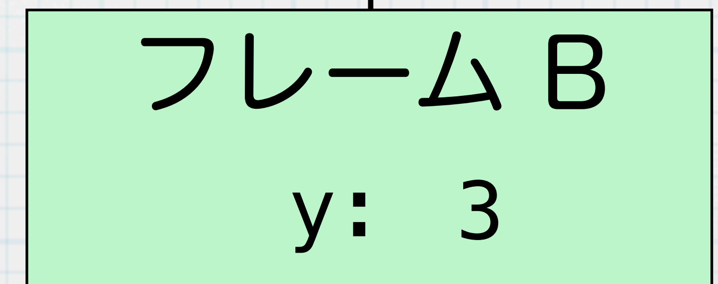
```
(let ((x 2))
  (let ((y 3))
    (let ((x1 5))
      (display x)
      (newline))))
```

←2を出力する









このような現象はほとんどのプログラミング言語で起こる

JavaScriptにおける環境と変数のスコープ (1)

JavaScriptは `function (...) {...}` という文法を用いて無名関数を作ることができ，これはSchemeにおける`lambda`と同様のものである．環境もしくみもここで説明したしくみと同様であると考えて良い．ただし，以下の点で異なる．

以下の2つの定義は全く同じである

```
function foo(x){
    return x * x
}

var foo  = function (x){
    return x * x
}
```

varで定義しているが値が代入されない変数の値はundefinedである

```
var mx
puts(mx) ←undefinedを出力する
```

varは現在のフレーム中に束縛を作る命令である．現在のフレームに束縛がなければ親フレームを見にいく．フレームを作られるのは関数呼び出しのときのみである．

```
var abc = 20;

function mm(){
    var abc = 30;
    function nn(){
        puts(abc); ←30を出力する
    }
    nn()
}

mm()
```

JavaScriptにおける環境と変数のスコープ (2)

JavaScriptのスコープで注意すべきことは，いかなる場所で書くことができて，それが影響する可能性があるということである．

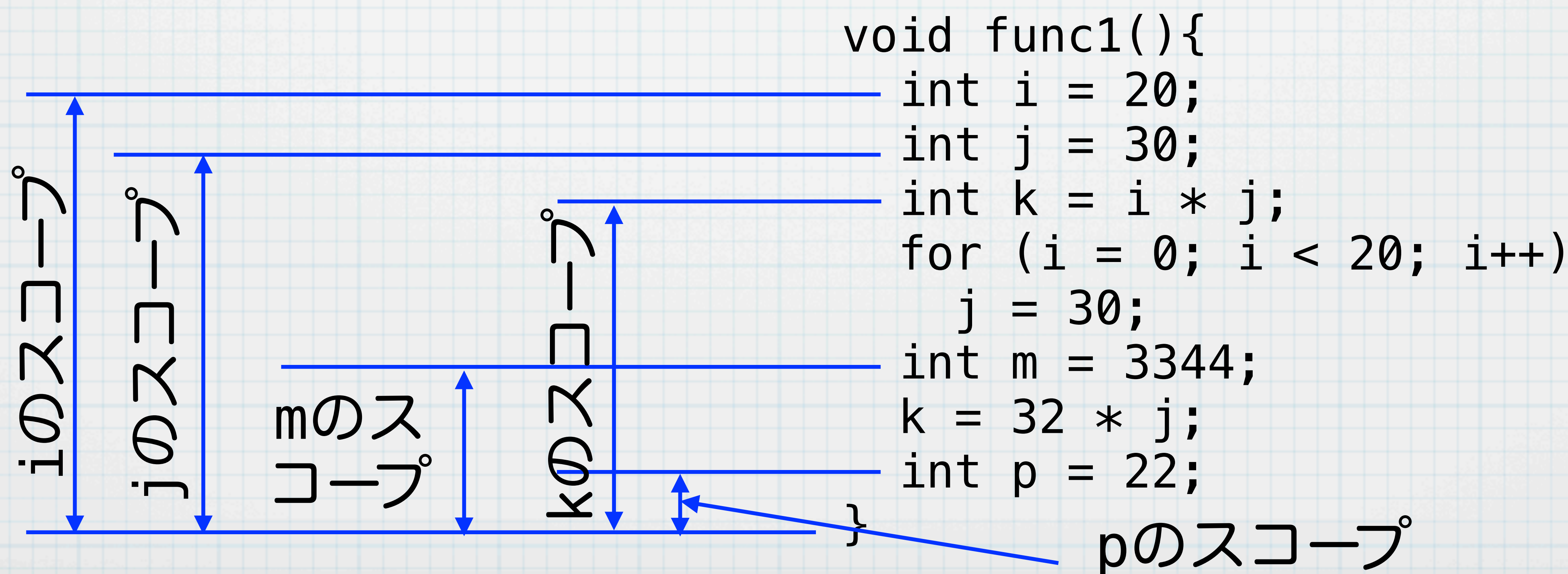
```
var abc = 30;

function func_a(){
    puts(abc)     ←undefinedを出力する
    var s = 0;
    for (var i = 0; i < 20; i++){
        s += i
        while (false){
            var abc = 40; ←この行がなければ上記puts
                              は30を出力する
        }
    }
    puts(s)
}
func_a()
```

JavaScriptの場合，上記var abc; はfunc_aの関数定義直後に書かれたのと同じことになる．このしくみを**hoisting (つり上げ)** と呼ぶ．**ただし，値の代入が実行されるのは書かれた場所で，実行されるまでは値はundefinedとなる**

C/C++におけるスコープ (1)

C/C++は関数の中に関数を書くことができず，さらにlambdaに対応するものも存在しないので，環境は必然的に2層構造となる．環境の作り方は静的スコーピングであるが，前述のようにこのようなスコープを**平坦スコープ**と呼ぶ．基本的にスコープはブロック単位（{}で囲まれた単位）で管理されているが，それぞれの変数は `int i;` など変数定義をした場所から有効となり，それよりも前では同じブロックに入っていても見えない．JavaScriptのhoistingのようなしくみはない．

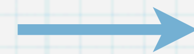
※ ただし，ANSI Cはブロックの先頭でしか変数定義できない．



C/C++におけるスコープ (2)

C/C++ではブロック単位でスコープを管理するので，適当にブロックをつくることで，そこだけの局所変数を利用することが可能となる．

```
#include <stdio.h>

void foo(){
  int i = 20;
  int j = 30;
  {
    int j = 40;
    printf("%d\n", i + j);
  }
  {
    int i = 50;
    printf("%d\n", i + j);
  }
  printf("%d\n", i + j);
}

int main(){
  foo();
  return 0;
}
```



```
osami-2:yama527> ./scope4-1
60
80
50
```

それぞれのブロックでフレームが生成されている．

Pythonにおけるスコープ (1)

PythonやRubyといったプログラミング言語は静的スコープを利用するが，変数を利用すると暗黙にその変数が定義される．したがって，変数を定義するための文法（JavaScriptにおける var のような文法）は存在しない．関数は何重にも深く定義することができるので，このことに起因する問題が発生する．

Pythonは **L** (ローカルフレーム), **E** (その周りのフレーム），**G**（グローバルフレーム），**B**（ビルトイン環境）の順に変数を探す．ただし，右のプログラムはエラーとなる．

```
def make_account(name):
    balance = 0
    def foo(mesg):
        if mesg == 'deposit':
            balance += 20
        elif mesg == 'withdraw':
            balance -= 20
        elif mesg == 'balance':
            return balance
    return foo

acc = make_account('yama')
acc('deposit')
acc('deposit')
print acc('balance')
```

```
osami-2:yama519> python scope5.py
Traceback (most recent call last):
  File "scope5.py", line 14, in <module>
    acc('deposit')
  File "scope5.py", line 6, in foo
    balance += 20
UnboundLocalError: local variable 'balance'
referenced before assignment
```

balanceは関数fooの局所変数となってしまう

Pythonにおけるスコープ (2)

前のスライドで示したPythonプログラムを動くように修正した物が下記のプログラムである．**Pythonでは普通の変数への代入が存在すると，局所変数を自動的に生成してしまう．**ところが，配列への代入は配列オブジェクトの操作であって，変数への代入とは解釈しない．よって，局所変数が生成されずに，一段上のフレームの変数を参照できる．

これ自体はかなり姑息なテクニックである．Pythonの新しいバージョンであるPython3では，nolocalという宣言文が導入されており，このような方法を使わなくても同様のプログラムが書けるようになっている．

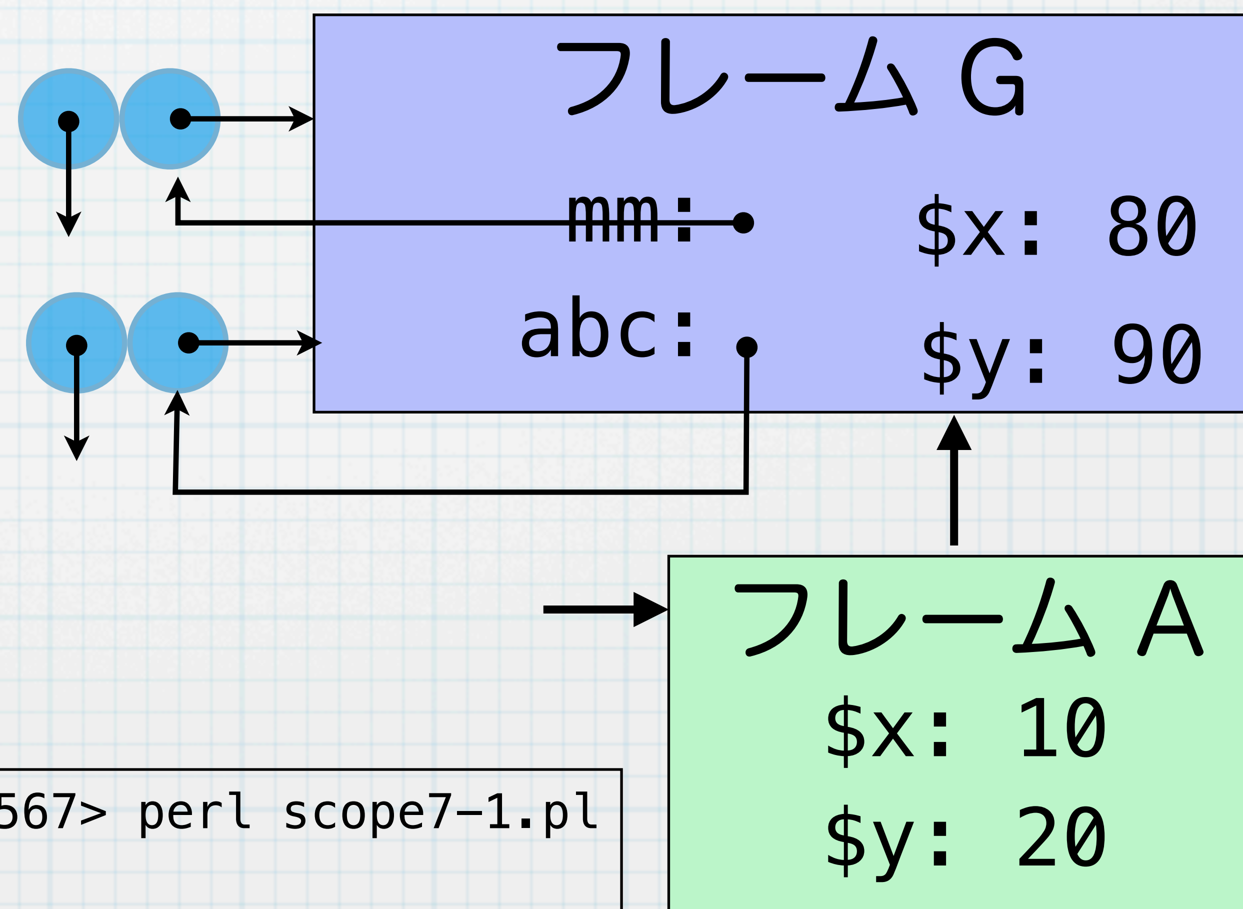
```
def make_account(name):
    balance = [0]
    def foo(mesg):
        if mesg == 'deposit':
            balance[0] += 20
        elif mesg == 'withdraw':
            balance[0] -= 20
        elif mesg == 'balance':
            return balance[0]
    return foo

acc = make_account('yama')
acc('deposit')
acc('deposit')
print acc('balance')
```

Perlにおけるスコープ

PerlはLarry Wallによって定義されたスクリプト言語であり，現在では色々なツールを記述したり，ネットワークの処理を記述したりするのに色々なところで利用されている．この言語では，動的スコープと静的スコープを併用することができる．動的スコープのしくみを利用する変数は`local`というキーワードで定義される．下の例の場合，関数mmは呼び出されたフレームabcを親フレームとするフレーム上で評価される．

```
sub mm {
    print $x, "-", $y, "\n"
}

sub abc {
    mm;
    local ($x, $y);
    ($x, $y) = (10, 20); mm;
    ($x, $y) = (30, 20); mm;
}

($x, $y) = (80, 90);
abc;
```

`local`を`my`に変えると静的スコープとなり，すべて80-90と表示される．




```
osami-2:yama567> perl scope7-1.pl
80-90
10-20
30-20
```

Emacs Lisp における動的スコープ

Emacs LispやCommon LispはSchemeの影響を受け，現在では静的スコープがデフォルトとなっているが，同時に動的スコープで見える変数を作ることができる．このような変数はdefvarによって生成することができ，通常 *abc* のように両側に*を伴った変数名をつける慣例がある．

```
(defvar *a*)
==> *a*
(defun foo () *a*)
==> foo
(setf *a* 20)
==> 20
(let ((*a* 30)) (foo))
==> 30
(foo)
==> 20
```

Emacs Lisp の実行例

変数 *a* は動的スコープになる

```
#|kawa:1|# (define *a* 20)
#|kawa:2|# (define (foo) *a*)
#|kawa:3|# (foo)
20
#|kawa:4|# (let ((*a* 30)) (foo))
20
#|kawa:5|#
```

Kawaでの同様の関数の実行例

変数 *a* は静的スコープ

C++の名前空間について

```
namespace meijo{
  int i = 20;
  namespace riko{
    int i = 30;
    int j = 40;
    int foo(int x, int y){
      return x + y;
    }
    namespace joho{
      int bar(){
        return foo(i, j);
      }
    }
    namespace denshi{
      int bar(){
        return 2 * i + j;
      }
    }
  }
}
```

C++は平坦スコープを利用しているので，大量の変数を管理することが難しい．そのため，**名前空間 (namespace)** というしくみが用意されている．これにより適当に変数を隠蔽して必要な関数だけを見せることが可能となる．

名前空間の階層構造を作ることができるので効率的に変数を管理できる．

```
#include <stdio.h>

int main(){
  printf("%d\n", meijo::riko::joho::bar());
  printf("%d\n", meijo::riko::denshi::bar()
  return 0;
}
```