アルゴリズムとデータ構造入門
1.手続きによる抽象の構築
1.2 プログラムの構築

奥 乃 博
大学院情報学研究科知能情報学専攻
知能メディア講座 音声メディア分野
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/06/IntroAlgDs/
okuno@i.kyoto-u.ac.jp

TAの居室は10号館4階奥乃1研，2研
TAのページがオープン，質問箱もあります

10月17日・本日のメニュー

- 1-1-5 The Substitution Model for Procedure Application
- 1-1-6 Conditional Expressions and Predicates
- 1-1-7 Example: Square Roots by Newton's Method
- 1-1-8 Procedures as Black-Box Abstractions
- 1.2.1 Linear Recursion and Iteration
- 1.2.2 Tree Recursion
- 1.2.3 Orders of Growth
- 1.2.4 Exponentiation

1.1.4 Compound Procedures(合成手続き)

- "*To square something, multiply it by itself.*"

```
(define (square x) (* x x))
```

To square something, multiply it by itself

- "**square**"という名前の合成手続き.
- **(define (** *<name>* *<formal parameters>* **)**
  <body> **)**
  - *<formal parameter>* **仮パラメータ**
  - *<body>* **本体**

1-1-5 The Substitution Model for Procedure Application (置換モデル)

- **Vocabulary (語彙)** ⇒Primitives
- **Syntax (構文)** ⇒means of abstractions
- **Semantics (意味)** ⇒ Viewing the rules of evaluation from a computational perspective (計算という観点からの評価法)

- 手続き適用の評価法として「置換モデル」

4

置換モデルの例による説明

```
(define (square x) (* x x))
(define (sum-of-squares x y)
   (+ (square x) (square y)) )
(define (f a) (sum-of-squares (+ a 1) (* a 2)))

(f 5)
     (sum-of-squares (+ a 1) (* a 2)) に a = 5 を適用
(sum-of-squares (+ 5 1) (* 5 2))
     (+ (square x) (square y))  に                を適用
(+ (square 6) (square 10))
            に x = 6,        に x = 10 を適用
(+ (* 6 6) (* 10 10))
(+       )
136
```

6

Applicative order vs. normal order (適用順序と正規順序)

- 今見てみた置換モデルの評価順序:「適用順序(作用順序, Applicative order)」
- 別の順序:「正規順序(normal-order)」: 仮パラメータを展開してから, 簡約する.

```
1.  (f 5)
2.       ((sum-of-squares (+ a 1) (* a 2)) 5)
3.  (sum-of-squares (+ 5 1) (* 5 2))
4.       ((+ (square x) (square y)) (+ 5 1) (* 5 2))
5.  (+ (square (+ 5 1)) (square (* 5 2))
6.       (+ ((* x x) (+ 5 1)) ((* x x) (* 5 2))
7.  (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
8.  (+ (* 6 6) (* 10 10))
9.  (+ 36 100)
10. 136
```

2回同じものを計算

7

1.1.6 Condtional Expressions and Predicates (条件式と述語)

- 絶対値をcase analysis (場合分け)で定義

$$|x| = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -x & \text{if } x < 0 \end{cases}$$

```
1.(define (abs x)
    (cond ((> x 0) x)
          ((= x 0) 0)
          ((< x 0) (- x)) ))
```

糖衣

```
2.(define (abs x)
    (cond ((< x 0) (- x))
          ((>= x 0) x) ))
```

```
(or (> x 0) (= x 0))
```

```
3.(define (abs x)
    (cond ((< x 0) (- x))
          (else x) ))
```

```
4.(define (abs x)
    (if (< x 0)
        (- x)
        x ))
```

if : Syntax sugar

1.1.6 Condtional Expressions and Predicates (条件式と述語)

- 条件式の一般形; **cond** は特殊形式 (special form)
- **(cond** (<p_1> <e_{11}> ... <e_{1m}>)
  (<P_2> <e_{21}>...<e_{2k}>)
  ...
  (<p_n> <e_{n1}>...<e_{np}>) **)**
- 式の対 (<*p*> <*e*> ... <*e*>): 節(clause)
- <*p*>: 述語(predicate)
- 述語の値: true (**#t**) か false (**#f**).
- <*e*>: 帰結式(consequent expression)
- 特別の <*p*>: **else** (**#t** を返す)
- 節の評価は, **<*p*>が#tなら<*e*>**が順に評価される.
- 一旦述語が**#t**を返すと, それ以降の節は評価されない.

9

1.1.6 Condtional Expressions (条件式)

- **(define (abs x)**
  (cond ((< x 0) (- x))
  (else x)))
- **(define (abs x)**
  (if (< x 0)
  (- x)
  x))
- **If** : 特殊形式(special form)
- **(if** <*predicate*> <*consequent*> <*alternative*>**)**
- **If** は **cond** の特殊な場合に対する syntax sugar (構文シュガー)　糖衣錠ですね☺

10

1.1.6 Predicates (述語)

- (and $<e_1>$ … $<e_n>$) 論理積(左から評価)
- (or $<e_1>$ … $<e_n>$) 論理和(左から評価)
- (not $<e>$) 論理否定

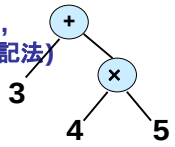
例:

- 5 < x < 10 ⇒
- (define (>= x y)
  (or (> x y) (= x y)))
- (define (>= x y)
  (not (< x y)))

11

Ex.1.2 前置記法(prefix notation)

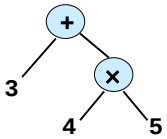
- 式 (演算子 被演算子 …)
  operator operands
- 式の記法
  - 前置記法 (prefix notation , Pollish notation, ポーランド記法)
    + 3 * 4 5
  - 中置記法 (infix notation)
    3 + (4 * 5)
  - 後置記法 (postfix notation, reverse Pollish notation、逆ポーランド記法)
    3 4 5 * +
- 木表現はどれも同じ



12

木の辿り方から3つの記法への変換

- 木の辿り方
  - 前順走査 (pre-order traversal)
    ノード⇒左部分木⇒右部分木
    + ⇒ 3 ⇒ * ⇒ 4 ⇒ 5
  - 間順走査 (in-order tr.)
    左部分木⇒ノード⇒右部分木
    3 ⇒ + ⇒ 4 ⇒ * ⇒ 5
  - 後順走査 (post-order tr.)
    左部分木⇒右部分木⇒ノード
    3 ⇒ 4 ⇒ 5 ⇒ * ⇒ +



Javaプログラムのデモをすること

13

Ex.1.3 Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

```
(define (sum-of-two-sq x y z)
   (cond ((> x y)
          (cond ((> y z)
                 (sum-of-squares x y) )
                (else (sum-of-squares x z)) ))
         ((> x z) (sum-of-squares y x))
         (else (sum-of-squares y z)) ))

(define (sum-of-two-sq x y z)
   (if (> x y)
       (if (> y z)
           (sum-of-squares x y)
           (sum-of-squares x z) )
       (if (> x z)
           (sum-of-squares y x)
           (sum-of-squares y z) )))
```

14

Ex.1.4

```
(define (a-plus-abs-b a b)
    ((if (> b 0) + -) a b) )

(a-plus-abs-b 5 8)
     ((if (> b 0) + -) a b) に
     a = 5, b = 8 を適用
(+ 5 8)
     13
(a-plus-abs-b 5 -8)
     ((if (> b 0) + -) a b) に
     a = 5, b = -8 を適用
(- 5 -8)
     13
```

15

Ex.1.5

```
(define (p) (p))
(define (test x y)
   (if (= x 0)
       0
       y ))
```

(test 0 (p))

Applicative order	
1	(if (= x 0) 0 (p))
2	(if (= x 0) 0 (p))
3	(if (= x 0) 0 (p))
∞	...

Normal order	
1	(if (= x 0) 0 (p))
2	0

一般にapplicative order で値が求まれば、normal order でも同じ値が求まる

1.1.7 Square Root by Newton's Method

$\sqrt{x}$ *is the* y *such that* $y^2 = x$ *and* $y \geq 0$

Recursive（再帰的）

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x) ))
(define (improve guess x)
  (average guess (/ x guess)) )
(define (average x y)
  (/ (+ x y) 2) )
(define (good-enough? guess x)
   (< (abs (- (square guess) x)) 0.001))
```

17

1.1.7 Square Root by Newton's Method

```
(define (sqrt x)
  (sqrt-iter 1.0 x) )
```

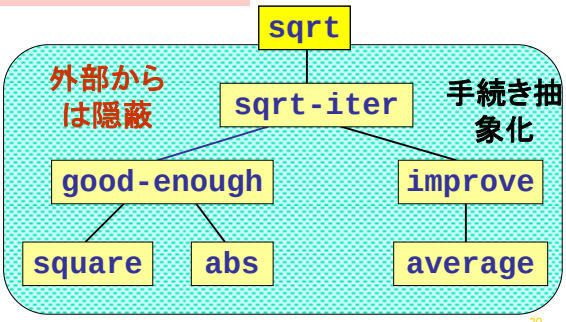
と定義すれば,

```
(sqrt 9)

(sqrt (+ 100 37))

(sqrt (+ (sqrt 2) (sqrt 3)))

(sqrt (sqrt 1000))
```

19

1.1.8 Procedures as Black-Box Abstraction (手続き:ブラックボックス抽象化)

Sqrtの手続き分解



20

手続き抽象化の効用 Square の定義

1.内部実装(implementation)の隠蔽

```
• (define (square x) (* x x))
• (define (square x)
     (exp (double (log x))) )
  (define (double x) (+ x x))
```

2.局所名の隠蔽

```
• (define (square x) (* x x))
• (define (square y) (* y y))
```

21

束縛変数（bound variables）と 自由変数（free variables）

bound（束縛）

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x) ))
(define (good-enough? guess x)
   (< (abs (- (square guess) x)) 0.001)
(define (good-enough? v target)
   (< (abs (- (square v) target)) 0.001)
```

- 束縛変数：仮パラメータは手続きで束縛
- 自由変数：束縛・captureされていない
- 有効範囲（scope）変数の束縛されている式の範囲

22

Block Structure（ブロック構造）

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001) )
  (define (improve guess)
    (average guess (/ x guess)) )
  (define (sqrt-iter guess)
     (if (good-enough? guess)
         guess
         (sqrt-iter (improve guess)) ))
  (sqrt-iter 1.0) )
```

23

Block Structure(ブロック構造): x の scope(有効範囲)は

```
(define (sqrt x)
  (define (improve guess x)
    (average guess (/ x guess)) )
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001) )
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x) ))
  (sqrt-iter 1.0 x) )
```

静的有効範囲(lexical scoping)

10月17日・本日のメニュー

- 1-1-5 The Substitution Model for Proce Application
- 1-1-6 Conditional Expressions and Prec
- 1-1-7 Example: Square Roots by Newt Method
- 1-1-8 Procedures as Black-Box Abstractions
- Intermission
- 1.2.1 Linear Recursion and Iteration
- 1.2.2 Tree Recursion
- 1.2.3 Orders of Growth
- 1.2.4 Exponentiation

25

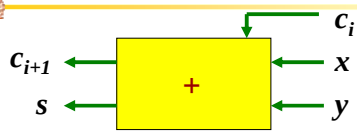
What is this instrument?

- A traditional roller-blader?
- A traditional inliner skate?
- Abacus
- 算盤 （そろばん）

- そろわん

26

Abacus & Binary Adder (2進加算器)




```
(define (adder x y c)
  (define (carry x y c)
    (if (or (and (= x 1) (= y 1))
            (and (= y 1) (= c 1))
            (and (= c 1) (= x 1)) )
        1 0 ))
  (define (sum x y c)
    (xor x y c) )
  (cons (sum x y c) (carry x y c)) )

(define (xor x y z)
  (if (= x 0)
      (if (= y 0) z (if (= z 0) 1 0))
      (if (= y 0) (if (= z 0) 1 0) z) ))
```

28

10月17日・本日のメニュー

- 1-1-5 The Substitution Model for Proce Application
- 1-1-6 Conditional Expressions and Prec
- 1-1-7 Example: Square Roots by Newt Method
- 1-1-8 Procedures as Black-Box Abstractions
- Intermission
- 1.2.1 Linear Recursion and Iteration
- 1.2.2 Tree Recursion
- 1.2.3 Orders of Growth
- 1.2.4 Exponentiation

32

1-2-1 Linear Recursion and Iteration

- 階乗の定義

```
(define (factorial n)
  (if (<= n 0)
      1
      (* n (factorial (- n 1))) ))
```

To define n!, if it is non-positive, return 1
otherwise, multiply it by (n-1)!

n! = n * (n-1)!

どう実行されるか。Substition model で実行

33

factorial の置換モデルによる実行

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 (* 1 (factorial 0)))))))
(* 6 (* 5 (* 4 (* 3 (* 2 (* 1 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

34

1-2-1 Linear Recursion and Iteration

- 階乗の定義（その1）

```
(define (factorial n)
   (if (<= n 0)
       1
       (* n (factorial (- n 1))) ))
```

To define N!, *if it is non-positive, return* 1
otherwise, multiply it by (N-1)!

- どう実行されるか。Substition model で実行
- Linear recursive process （線形再帰的プロセス）
  （Nに比例して再帰プロセスが生じる）
- 積は deferred operations（遅延演算）

35

factorial の置換モデルによる実行

Deferred operation

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 (* 1 (factorial 0)))))))
(* 6 (* 5 (* 4 (* 3 (* 2 (* 1 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

36

1-2-1 Linear Recursion and Iteration

- 階乗の定義（その2）

```
(define (factorial n)
  (fact-iter 1 1 n) )
(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count) ))
```

To define n!, `n! = 1 * 2 * ··· * n`

*product = counter * product*
counter = counter + 1

どう実行されるか。Substition model で実行

37

factorial の置換モデルによる実行

```
(factorial 6)
(fact-iter   1 1 6)
(fact-iter   1 2 6)
(fact-iter   2 3 6)
(fact-iter   6 4 6)
(fact-iter  24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

- **Linear iterative process**
  (線形反復プロセス)

38

factorial – Block Structure

```
(define (factorial n)
   (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
                    (+ counter 1) )))
   (iter 1 1) )
```

- 手続き iter は、factorial の中で有効。
- 外部からは隠蔽。

39

Tail recursion の補足説明

```
(define (fact n)
  (if (= n 1)
      1
      (* (fact (- n 1)) n) ))
```

- このプログラムは次の翻訳
  n! = (n-1)! * n
- 先ほどのfactorialとの違いは

```
(define (factorial n)
    (if (<= n 0)
        1
        (* n (factorial (- n 1))) ))
```

40

factの置換モデルによる実行

```
(fact 6)
(* (fact 5) 6)
(* (* (fact 4) 5) 6)
(* (* (* (fact 3) 4) 5) 6)
(* (* (* (* (fact 2) 3) 4) 5) 6)
(* (* (* (* (* (fact 1) 2) 3) 4) 5) 6)
(* (* (* (* (* (* (fact 0) 1) 2) 3) 4) 5)
  6)
(* (* (* (* (* 1 1) 2) 3) 4) 5) 6)
(* (* (* (* 1 2) 3) 4) 5) 6)
(* (* (* 2 3) 4) 5) 6)
(* (* 6 4) 5) 6)
(* 24 5))
(* 120)
720
```

41

Tail recursion による高速化

```
SC> (time (null? (factorial 5000)))
total time: 0.729999999999563 secs
user time: 0.690993 secs
system time: 0 secs
#f
SC> (time (null? (fact 5000)))
total time: 1.34000000000015 secs
user time: 1.321901 secs
system time: 0 secs
#f
SC> (time (null? (fact-iter 5000)))
total time: 0.720000000001164 secs
user time: 0.701008 secs
system time: 0 secs
#f
```

コンパイルすると factorial とfact-iterは同じコードに変換される。

42

Procedure（手続き）vs. Process(プロセス)

- 手続きが再帰的とは、構文上から定義。
  自分の中で自分を直接・間接に呼び出す。
- 再帰的手続きの実行
  - 再帰プロセスで実行
  - 反復プロセスで実行
- 線形再帰プロセスは線形反復プロセスに変換可能
  「tail recursion（末尾再帰的）」
- 再帰プロセスでは、deferred operation用にプロセスを保持しておく必要がある
  ⇒ スペース量が余分にいる。
- Scheme のループ構造はsyntactic sugar
  - do, repeat, until, for, while

44

Ex.1.10 Ackermann Function

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1)
                 (A x (- y 1)) ))))
```

- Ackermann関数は線形再帰ではない！

```
(define (Ack m n)
  (cond ((= m 0) (+ n 1))
        ((= n 0) (Ack (- m 1) 1))
        (else (Ack (- m 1)
                   (Ack m (- n 1)) ))))
```

45

10月17日・本日のメニュー

- 1-1-5 The Substitution Model for Proce Application
- 1-1-6 Conditional Expressions and Pred
- 1-1-7 Example: Square Roots by Newt Method
- 1-1-8 Procedures as Black-Box Abstractions
- Intermission
- 1.2.1 Linear Recursion and Iteration
- 1.2.2 Tree Recursion
- 1.2.3 Orders of Growth
- 1.2.4 Exponentiation

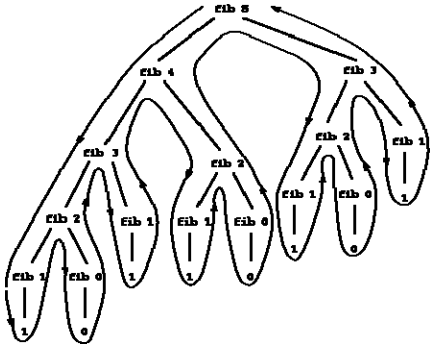
46

1.2.2 Fibonacci – Tree Recursion（木構造再帰）

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2)) ))))
(define (fib n)
  (fib-iter 1 0 n))
(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1)) ))
```

48

1.2.2 Fibonacci – Tree Recursion



49

1.2.2 Fibonacci – Iteration

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2)) ))))
```

- トップダウン(top-down)式に計算

```
(define (fib n)
  (fib-iter 1 0 n))
(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1)) ))
```

- ボトムアップ(bottom-up)式に計算

50

n種類の硬貨，金額aの両替の場合の数は

- 使える硬貨をある順番で並べておくと
- n種類の硬貨で金額aの両替の場合の数は
  1. 先頭の種類を除いたすべての硬貨を使って金額aを両替する場合の数
     ＋
  2. 先頭の種類の硬貨（額面d）とすると，a−dの額を全n種の硬貨を使って両替する場合の数
  3. 初期値：a=0の時1，a<0の時かn=0の時0

52

Ex. Counting Change

```
(define (count-change amount)
  (cc amount 5) )
(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount (- kinds-of-coins 1))
                 (cc (- amount (first-denomination
                                kinds-of-coins))
                     kinds-of-coins )))))
(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50) ))
```

54

宿題：10月23日午後5時締切

1. Ex.1.6
2. Ex.1.8
3. Ex.1.10
4. 両替のプログラムを動かす
5. ＄1、＄3の両替は何通り？

55