

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

1.1 プログラムの要素

奥 乃 博



1. TUT Schemeが公開されました.
 - Windowsは動きます.
 - Linux, Cygwin はうまく行かず. 調査中.
2. 随意課題7の追加
 - 友人の勉強を助け, TAの手伝いをする.
 - 支援した内容を毎回のレポート等で詳細に報告.
3. TAのページ
<http://winnie.kuis.kyoto-u.ac.jp/~fujihara/algorithm/>

10月4日・本日のメニュー

- 1-1-1 Expressions
- 1-1-2 Naming and the Environment
- 1-1-3 Evaluating Combinations
- 1-1-4 Compound Procedures
- 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
- 2-1-1, 2-2-2 Pairs and sequences



2

左上教科書表紙 : <http://mitpress.mit.edu/images/products/books/0262011530-f30.jpg>




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

3




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」



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

(+ (square 6) (square 10))

に x = 6, に x = 10 を適用

(+ (* 6 6) (* 10 10))

(+)

136



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

1.1.6 Conditional 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)))
```

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

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

糖衣

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

if : Syntax sugar

1.1.6 Conditional 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 (<p₁> <e₁₁> ... <e_{1m}>)
(<p₂> <e₂₁> ... <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。

1.1.6 Conditional 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☺



1.1.6 Predicates (述語)

- (and $\langle e_1 \rangle \dots \langle e_n \rangle$) 論理積(左から評価)
- (or $\langle e_1 \rangle \dots \langle e_n \rangle$) 論理和(左から評価)
- (not $\langle e \rangle$) 論理否定

例:

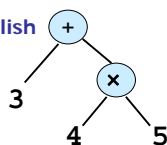
- $5 < x < 10 \Rightarrow$
- (define ($\geq x y$)
 (or ($> x y$) ($= x y$)))
- (define ($\geq x y$)
 (not ($< x y$)))

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, Polish 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 Polish notation, 逆ポーランド記法)
 $3 4 5 * +$
- 木表現はどれも同じ

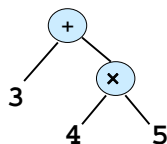


12




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
 $+ \Rightarrow 3 \Rightarrow * \Rightarrow 4 \Rightarrow 5$
 - 間順走査 (in-order tr.)
左部分木⇒ノード⇒右部分木
 $3 \Rightarrow + \Rightarrow 4 \Rightarrow * \Rightarrow 5$
 - 後順走査 (post-order tr.)
左部分木⇒右部分木⇒ノード
 $3 \Rightarrow 4 \Rightarrow 5 \Rightarrow * \Rightarrow +$



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normal order で
値が求まれば、
applicative 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))

```



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))

```



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



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)

```
(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))
```

bound
(束縛)

- 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)

2.2 対(pairs)とシーケンス(sequences)

- 対 (pair)
- シーケンス(並び, sequence)
- (cons 1 nil)
- (list 1)と同じ
- nilは空リスト
- (list 1 2 3 4)
- (cons 1 (cons 2 (cons 3 (cons 4 nil))))

	(cons 1 2)	(cons 1 nil)
ドット記法 (dotted notation)	(1 . 2)	(1 . nil) または (1)
箱ポインタ記法 (box-and-pointer notation)		

(1 2 3 4) あるいは (1 . (2 . (3 . (4 . nil))))

リスト処理演算 (list processing)

- (null <expression>) <expression> が nil か?
- (eq? <e₁> <e₂>) <e₁> と <e₂> が同じオブジェクトか?
- (cons <e₁> <e₂>) <e₁> と <e₂> から pair を作成
- (car <list>) <list> の car を抽出
- (cdr <list>) <list> の cdr を抽出

	⇒
(car (cons 1 2))	1
(car (list 1 2 3 4))	1
(cdr (cons 1 2))	2
(cdr (cons 1 nil))	nil
(cdr (list 1))	nil
(cdr (list 1 2 3 4))	(2 3 4)
(car (cdr (list 1 2 3 4)))	2
(car nil)	error
(cdr nil)	error



リスト処理の練習問題

- `(quote <expression>)` `<expression>` を返す.
- `' <expression>` `<expression>` を返す.
- `(cons '1 '(2 3)) ⇒ (1 2 3)`
- `(cons '(1) '(2 3)) ⇒ ((1) 2 3)`
- **equal? を定義せよ.**
- `(equal? 1 1) ⇒ #t` `(equal? 1 2) ⇒ #f`
- `(equal? 3 '(1 2)) ⇒ #f` `(equal? '(1 2) '(1 2)) ⇒ #t`
- `(equal? '(1 2) '(1 3)) ⇒ #f` `(equal? '(1 2) '(1 3)) ⇒ #f`
- **member? を定義せよ.**
- `(member 1 '(1 2)) ⇒ (1 2)` `(member 3 '(1 2)) ⇒ #f`
- `(member '(1 2) nil) ⇒ #f`
- `(member '(1 2) '(1 2 (1 2) 3)) ⇒ ((1 2) 3)`
- **append を定義せよ.**
- `(append '(1 2) '(3 4)) ⇒ (1 2 3 4)`
- `(append '(1 2) nil) ⇒ (1 2)` `(append 1 '(1 2)) ⇒ error`
- **reverse を定義せよ.**
- `(reverse (list 1 2 3 4)) ⇒ (4 3 2 1)`
- `(reverse '(1 2 3)) ⇒ (3 2 1)` `(reverse nil) ⇒ nil`

28



リスト処理の練習問題(自習用)

- **but-last を定義せよ.**
- `(but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)`
- **assoc を定義せよ.**
- `(assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))`
 `⇒ (hgo IntroAlgDS)`
- `(assoc hgo '((ishi Gairon) (iso math) (tom HW)) ⇒ #f`
- **length を定義せよ.**
- `(length '(1 2)) ⇒ 2` `(length '(1 2 3 5)) ⇒ 4`
- `(length nil) ⇒ 0`
- **copy を定義せよ.**
- `(copy '(1 2)) ⇒ (1 2)`
- `(copy '((1 . 2) . (3 . 4))) ⇒ ((1 . 2) 3 . 4)`
- **flatten を定義せよ.**
- `(flatten '((1)(2 (3) 4) 5)) ⇒ (1 2 3 4 5)`
- `(flatten '((1 2 3))) ⇒ (1 2 3)`
- `(flatten '(1 2 3)) ⇒ (1 2 3)` `(flatten nil) ⇒ nil`

29



宿題: 10月17日午後5時締切

- Ex.1.6, 1.8
- `(member? e lst)` を定義せよ.
- `(append e1 e2)` を定義せよ.
- Ex.2.18 `(reverse lst)` を定義せよ.

DON' T PANIC!



30
