# 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/07/IntroAlgDs/
okuno@i.kyoto-u.ac.jp

TAの居室は10号館4階奥乃1研，2研
池田 智志（mod(学籍番号, 2) ≡0） 奥乃研・音声対話G
神田 尚 （mod(学籍番号, 2) ≡1） 奥乃研・ロボットG



## 教科書 Structure and Interpretation of Computer Programs (SICP)

1. 世界中のComputer Scienceのトップレベルの教科書
2. 1回生後期で前半を
3. 2回生前期で後半を（湯淺先生）
4. MIT Press 提供オンライン版（無料）
5. Emacs Texinfo 形式（無料）
6. 日本語訳（邦訳・訳あり）約4.5K円

**教科書は持っているものとして進めます．**



## 参考書とScheme 処理系

1. ジョン・ベントリー（小林健一郎訳):『珠玉のプログラミング―本質を見抜いたアルゴリズムとデータ構造』（ピアソン）
2. **世界中にSICPのサイト・コースウェア等あり**
3. **宿題は自分でやること（答えを見ない）**
4. **TUT Scheme（湯淺研開発・メディアセンタ）**
   Windows, Cygwin, Linux, あり
5. 他の処理系は自己責任で使用のこと
   1. 過去のTAのページを参考に
   2. MIT-Scheme-6001
   3. Chez Scheme, ・・・

## 成績評価（上限は壱萬點）

1. 試験　80%
2. 必修課題　20%
   - ① 宿題で出した練習問題．レポート箱に提出（翌週同日12時締切，工学部10号館1階，今出川裏通り東端南）
   - ② 図形言語レポート（プログラムはメイルで提出）
3. 随意課題提出による“＋α”
   - ① 第2章までのすべての練習問題
   - ② Fixed-Point探索過程のSchemeによる可視化
   - ③ アルゴリズムのSchemeによる可視化
   - ④ これはすごいという抽象化を使ったSchemeプログラム（線形計画法，整数論，群論，組合せ論，古典力学，パズル解法，ゲーム，数独）
   - ⑤ Lego Mindstorm用Lisp　XS　を使った自律ロボット
   - ⑥ 図形言語で circle-limit（難しいが提出者あり）
   - ⑦ 他の学生の支援

5

## 成績評価（上限は壱百點）

過去3年間本講義
受講者の最終成績
の平均点は
75点を超えています。

6

## 10月2日・本日のメニュー


10月30日は
中間試験

1. SICP第1章　手続きによる抽象化(abstraction)
2. SICP 1.1　プログラムの要素
3. TUT Scheme の説明は
   10月9日　平石君（湯淺研D3，学振特別研究員）

OHPのマーク


教科書のまとめ


本講義独自

7

## 階乗のプログラムを書いてみよう

- 数学の記法で書くと
  n!=

8

## 階乗のプログラムを書いてみよう

1. **fact(n) = 1 * 2 * 3 * ··· * n**
2. **fact(n) = n *(n-1) *(n-2) * ···*1**
3. **factorial(n) = 1 if n ≦ 0**
   **n*factorial(n-1) otherwise**

```
4. (define (fact n)
     (if (<= n 0)
         1
         (* n (fact (- n 1))) ))
5. (fact 3)
6. (fact 10)
7. (fact 100)
```

12

## 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.
- This is a compound procedure, of which name is “square”.
- (define (<*name*> <*formal parameters*>)
  <*body*>)
  - <*name*> **名前**
  - <formal parameter> **仮パラメータ**
  - <body> **本体**

13

## 1.1.6 Conditional Expressions (条件式)

- 絶対値

```
abs(x) = x    if x >0
         -x   otherwise
```

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

- If : special form
- (if <*predicate*> <*consequent*> <*alternative*>)

述語　帰結部　代替部

14

## 階乗のプログラムを書いてみよう

1. fact(n) = 1 * 2 * 3 * ··· * n
2. fact(n) = n *(n-1) *(n-2) * ···*1
3. factorial(n) = 1　if n ≦ 0
   n*factorial(n-1)　otherwise

```
4. (define (fact n)
     (if (<= n 0)
         1
         (* n (fact (- n 1))) ))
5. (fact 3)
6. (fact 10)
7. (fact 100)
```

15

## 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。
Substitution model (置換モデル)で実行

16

## 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。Substation 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）

18

## 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
```

19

## 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
```

20

## factorial の計算量（Complexity）

```
(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
```

**factorialの呼ばれる回数**
*n回 for n!* (time complexity)
**未実行の * の量**
*最大 n回 for n!* (space complexity)

21

## 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。Substation model (置換モデル) で実行

22

## 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）**

23

## 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
```

`fact-iter` の呼ばれる回数
- *n回 for n!* (time complexity)

未実行の演算の量はない

余分な仮パラメータ
- *2個* (space complexity)

## 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。

25

## Tail recursion の補足説明

```
(define (fact n)
  (if (<= n 0)
      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))) ))
```

26

## 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
```



## 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。



## スターリングの公式（Sterling's formula）

- n!は $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\lambda_n}$ ただし $\frac{1}{12n+1} < \lambda_n < \frac{1}{12n}$
- スターリング級数は

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{1}{51840n^3} - \frac{1}{2488320n^4} + \cdots\right)$$

- 対数をとると

$$\ln n! = n\ln n - n + \frac{1}{2}\ln(2\pi n) + \frac{1}{12n} - \frac{1}{360n^3} + \frac{1}{1260n^5} - \frac{1}{1680n^7} + \cdots$$

$$\ln n! \approx n\ln n - n \quad \text{for n>>0}$$

## Intermission

# Disneyの映画

# “TRON”

# をご存じですか？

31

## “TRON” Disney 映画 (1982)

**A masterpiece of breakthrough CGI ingenuity**, Disney celebrates the 20th anniversary of TRON, a dazzling film at the flashpoint of a continuing revolution in its genre. This special collector's edition showcases an epic adventure inside a brave new world where the action is measured in microseconds.

When Flynn (Jeff Bridges) hacks the mainframe of his ex-employer to prove his work was stolen by another executive, he finds himself on a much bigger adventure. Beamed inside by a power-hungry master control program, he joins computer gladiators on a deadly game grid, complete with high-velocity "light cycles" and Tron (Bruce Boxleitner), a specialized security program. Together, they fight the ultimate battle with the MCP to decide the fate of both the electronic world and the real world!

32

## “TRON” のベース ASAS on Lisp

Craig W. Reynolds (III): Computer animation with scripts and actors, *Computer Graphics*, Vo.16, No.3, pp.289-296.

```
(defop arch-fractalizer
   (param: arch-element top-color bot-color levels
           fractal-ratio height width leg-width)
   (local: (total-levels  levels)
           (offset-dist  (half (dif width leg-width)))
           (sub-tower-offset-1 (vector offset-dist 0 0))
           (sub-tower-offset-2 (mirror x-axis
                                       sub-tower-offset-1)))
   (arch-tower levels))

(defop arch-tower
   (param: levels)
   (if (zerop levels)
       (then nothing)
       (else (add-arch-level (arch-tower
                              (dif levels 1))))))
(defop add-arch-level
    (param: sub-tower)
    (grasp sub-tower
           (scale factal-ratio)
           (move (vector 0 height 0))
           (rotate 0.25 y-axis))
    (grasp arch-element
           (recolor (interp (quo levels total-levels)
                            bot-color top-color)))
    (subworld (group arch-element
              (move subtower-offset-1 sub-tower)
              (move subtower-offset-2 sub-tower)))))
```

34

## Intermission

**Pluto（冥王星） が**

**惑星から準惑星に変更**

**になったのを**

**ご存じですか？**

35

## Pluto（134349）

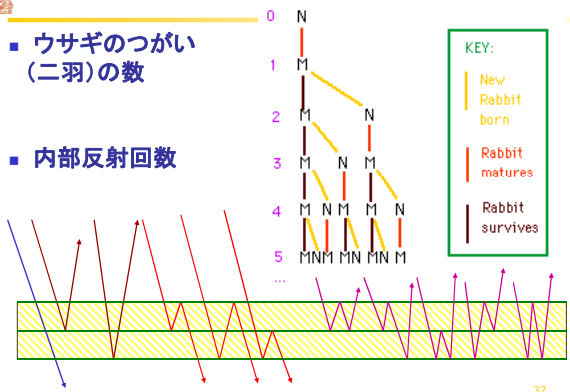
**"Chaotic Evolution of the Solar System"**,
Gerald Jay Sussman and Jack Sisdom,
*Science*, 257, July 1992.

The evolution of the entire planetary system has been numerically integrated for a time span of nearly 100 million years. This calculation confirms that the evolution of the solar system as a whole is chaotic, with a time scale of exponential divergence of about 4 million years. Additional numerical experiments indicate that the Jovian planet subsystem is chaotic, although some small variations in the model can yield quasiperiodic motion. The motion of Pluto is independently and robustly chaotic.

**証明はLisp（Scheme）で**

## フィボナッチ関数（Fibonacci Function）

- ウサギのつがい（二羽）の数
- 内部反射回数




37

## フィボナッチ数のプログラムを書こう

- 数学の記法で書くと
  fib(n)=

38

## 1.1.6 Conditional Expressions and Predicates (条件式と述語)

- General form of a conditional expression
- `(cond (`*<p1> <e1>*`)`
  `(`*<P2> <e2>*`)`
  …
  `(`*<pn> <en>*`) )`
- A pair of expressions (*<p> <e>*) called clauses **節**.
- *<p>* **predicate 述語**. Its value is interpreted as either `true` or `false`.
- *<e>* **consequent expression 帰結式**
- Special <p>: `else` **otherwise を意味する。**

39

## 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 ))
```

40

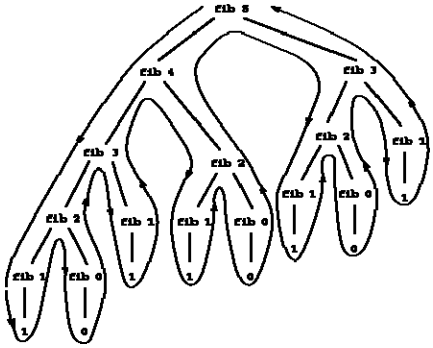
## フィボナッチ数のプログラムを書こう

fib(n) = 0 if n ≦ 0
1 if n = 1
fib(n-1)+fib(n-2) otherwise

```
(define (fib n)
  (cond ((<= n 0) 1)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2)) ))))
(fib 3)
(fib 10)
(fib 30)
```

41

## 1.2.2 Fibonacci – Tree Recursion




42

## 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

43

## フィボナッチ数の一般項を求める。

- 漸化式: $a_0 = 0 \quad a_1 = 1 \quad a_{n+2} = a_{n+1} + a_n$
- 一般的な解法 $a_{n+2} - a_{n+1} - a_n = 0$
  1. 方程式 $t^2 - t - 1 = 0$ の根を$\alpha$, $\beta$とすると
     一般項は $a_n = \mathrm{A}\alpha^n + \mathrm{B}\beta^n$ と表せる。
     ただし、A, B は定数である。
  2. 初期値等から、定数項の値を求める。
- 一般項を5分間で求めて下さい。

44

## フィボナッチ数は

- Fib(n) は $a_n = \frac{1}{\sqrt{5}}\left(\phi^n - \psi^n\right)$
- $\psi = \frac{1-\sqrt{5}}{2}$ より $-0.62 < \psi < 0$
- 従って、fib(n) は $a_n = \frac{1}{\sqrt{5}}\phi^n$ に最も近い整数

46

## アッカーマン関数を知っていますか

1. ack(m, n) =
   n+1 if m = 0
   ack(m-1, 1) if n = 0
   ack(m-1, ack(m, n-1)) otherwise

```
2.  (define (ackermann m n)
      (if (= m 0)
          (+ n 1)
          (if (= n 0)
              (ackermann (- m 1) 1)
              (ackermann (- m 1)
                         (ackermann m (- n 1))
              ))))
3.  (ackermann 0 2)
4.  (ackermann 1 2)
5.  (ackermann 2 2)
6.  (ackermann 3 2)
```

49

## 宿題1：アッカーマン関数の値

提出先は10号館レポート箱，締切9日正午

- (ackermann 0 2)
- (ackermann 1 2)
- (ackermann 2 2)
- (ackermann 3 2)

計算過程を書くこと．以下随意課題

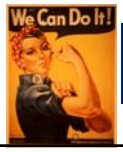
1. (ackermann 0 n) ≡ n+1 （理由も）
2. (ackermann 1 n) ≡ ？ （理由も）
3. (ackermann 2 n) ≡ ？ （理由も）
4. (ackermann 3 n) ≡ ？ （理由も）

50

## 宿題：10月9日正午締切

- **[必修] 問題1.13**
- **[随意] Stirling's Formula を導け.**
- レポート用紙に読める字で書くこと.
- ワープロ可．その場合名前等もワープロで.
- 友達に教えてもらったら，その人の名前を明記すること．Webは出展を明記.

（otherwise『同じ』回答は減点します）


**10月30日は中間試験**

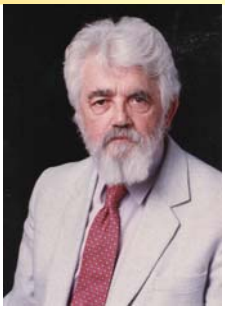
52

## Lisp言語

1. John McCarthyが1959年に設計・開発
   http://www-formal.stanford.edu/jmc/recursive.html
2. Fortran 言語についで2番目に古い言語
3. 種々の方言・実装あり、Schemeもその一つ
   MacLisp, Interlisp, TAO, Kyoto Common Lisp, …
4. 今日のオブジェクト指向などさまざまなアイデアを創出してきた「原言語」
5. 統合的プログラミング環境が提供
6. TRON（Disney）最初のCGによる映画
7. Pluto（134349）の軌道がChaoticの計算による証明
   - Galileo 以来の open problemの解決

54

## John McCarthy, Prof. Emeritus, SU

弟子さん：佐藤雅彦教授（情報），林晋教授（文学部），居候：奥乃

55

## Lispによるプログラミング

1. 計算モデル：再帰方程式（recursion equation) という論理表現とその推論方式
2. Lispの方言－scheme, CtCL, …
3. Lispの処理系(Implementation)
   - 解釈系（Interpreter): プログラムをそのまま解釈し、実行
   - コンパイラ（Compiler)：プログラムを機械語へ変換し、機械語をランタイムシステムの下で実行
4. 実装（implement)
5. 手続き（procedure)であるプログラムとデータが同じ形

56

## 数学とコンピュータサイエンスの違い

1. 宣言的知識（Declarative Knowledge)
   **“What is true”という知識**

   $\sqrt{x}$ *is the* $y$ *such that* $y^2 = x$ *and* $y \geq 0$
2. 規範的知識（命令的，Imperative Knowledge)
   **“How to”という知識**

   $x = 2$ *に対する* $\sqrt{x}$ *の値を求めるには*

| 予測 (guess) | 相棒は商で求まる | 予測とその相棒の平均値で改善 |
|---|---|---|
| 1 | 2/1 = 2 | (1 + 2)/2 = 1.5 |
| 1.5 | 2/1.5 = 1.333 | (1.5 + 1.333)/2 = 1.4167 |
| 1.4167 | 2/1.4167 = 1.4118 | (1.4167 + 1.4118) = 1.4142 |
| 1.4142 | 2/1.4142 = | |

57

## “How to” 知識を概念化する

1. **手続き(procedure)**
   所望の値を求めるステップ系列の概念 ーrecipeのようなもの
2. **計算プロセス(computational process)**
   具体的に計算機の中で実行されるステップの展開ー実際の調理
3. **データ(data)** ー 材料
4. **プログラム(program)= 手続き ＋ データ**
   計算プロセスはプログラム指示によりデータを操作
5. 指示誤り:バグ(虫, bug)、スリップ(glitches)
6. 間違い修正:虫とり(debug)
7. 言語(language) あるいはプログラミング言語
   計算プロセスを記述するために使用
   - **Vocabulary (語彙)**
   - **Syntax (構文)**ー 複合式を構築するためのルール
   - **Semantics (意味)**ー構成子に意味を付与するためのルール

58

## “How to” 知識・概念化のポイント

複雑さとの戦いー単純なデータと手続き

- Vocabulary (語彙)
- Syntax (構文)
- Semantics (意味)

1. 手続き抽象化(procedure abstraction)
2. データ抽象化(data abstraction)

59