1.0 本日のトピック

- プロセス，プログラム，プログラミング言語
- プログラミング言語の類型
- 構文と意味
- Scheme の構文
- 名前と環境 (データの抽象化)
- 手続きの抽象化 (λ 式)
- 評価の規則，適用の規則

1.1 プログラム，データ，プロセス

(1) プログラム：コンピュータへの計算[1]の指示 cf. アルゴリズム：どのように計算をするか
Niklaus Wirth, *Algorithms + Data Structures = Program*, Prentice Hall 1978

(2) プログラミング言語：プログラムを記述するための言語 (形式言語 ⇔ 自然言語)
→【オートマトンと形式言語 (2Y2Q)】

(3) プロセス：計算をする実体

(4) データ：計算をするための材料 (計算の対象)

1.2 プログラミング言語の歴史 (抽象化の歴史)

(1) 機械語 : ハードウェア (CPU) の構造に依存，すべて数字 →【コンピュータ論理設計 (3Y1Q)】
e.g. PDP-11 (1970–1980) (DEC → Compaq → HP)

- 16 bit レジスタ 8 本 (R0–R7: R6=SP, R7=PC)
- 0110000001000011 (R1 の値と R3 の値を足算して R3 に格納せよ)

0110	000	001	000	011
op	addr.mode1	opr1	addr.mode2	opr2

(2) アセンブリ言語：機械語を読み・書きしやすく →【アセンブリ言語 (2Y3Q)】
e.g. PDP-11

- 0110 000 001 000 011 → ADD R1 R3
- ニューモニックによる可読性の向上
- 名前によるアドレスの参照
- ジャンプ命令による飛び先のアドレスの自動計算

```
1           .TITLE  HELLO WORLD
2           .MCALL  .TTYOUT,.EXIT
3   HELLO:: MOV     #MSG,R1  ;STARTING ADDRESS OF STRING
4   1$:     MOVB    (R1)+,R0 ;FETCH NEXT CHARACTER
5           BEQ     DONE     ;IF ZERO, EXIT LOOP
6           .TTYOUT          ;OTHERWISE PRINT IT
7           BR      1$       ;REPEAT LOOP
8   DONE:   .EXIT
9   MSG:    .ASCIZ /Hello, world!/
10          .END    HELLO
```

(3) 高級言語 : ハードウェアからの独立 →【手続型プログラミング基礎 (1Y1Q)，手続型プログラミング発展 (1Y2Q)，オブジェクト指向プログラミング (2Y4Q)】

[1] ここでいう「計算」は数値計算ではなく，より広い意味での計算 → 【計算基礎論 (3)】

(4) FORTRAN (The IBM mathematical FORmular TRANsrating system),

- 数値計算を算術式で書きたい.
- 算術式を機械語に翻訳 (コンパイラ →【コンパイラ構成 (3Y2Q)】)

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
C      FORTRAN
C    x SAMPLE PROGRAM
       DIMENSION A(1000)
10     READ(5,100) X
100    FORMAT(I3)
       IF (X = 0) GOTO 999
       WRITE(6,200) X
200    FORMAT('X=',X)
       GOTO 10
999    STOP 999
```

(5) COBOL (COmmon Business Oriented Language)

- 主に商用計算用 → 自然言語 (英語) に近い記述を目指す.
- 記述が冗長 → プログラムが長くなる.

```
000010* CALCULATING Z = (X**2 + Y**2) / 2
000020
000030 MULTIPLY X BY X GIVING X-SQ.
000040 MULTIPLY Y BY Y GIVING Y-SQ.
000050 ADD X-SQ TO Y-SQ GIVING SQ-SUM.
000060 DIVIDE SQ-SUM by 2 GIVING Z
```

(6) LISP (LISt Processor)

- 記号処理，人工知能研究向け.
- λ 計算が数学的基礎 →【計算基礎論 (2Y1Q)】

```
(define (factorial n) (if (= n 0) 1 (* n (factorial (- n 1)))))
(define (member x l)
  (cond ((null? l) #f)
        ((eq x (car l)) #t)
        (else (member x (cdr l)))))
```

(7) Prolog (PROgramming in LOGic)

- Lisp と並んで 1980 年代の人工知能研究で使われた.
- 一階述語論理が数学的基礎 → 【情報論理 (2Y3Q)】

```
member(X, [X|_]).
member(X, [_|Y]) :- member(X, Y).
father(taro, itiro).
mother(taro, hanako).
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

1.3 プログラミング言語の類型

(1) どういう問題を記述したいか?

(2) 世界をどのように観るか? → プログラミング・パラダイム

(3) 手続き型プログラミング

- 計算 = 命令の実行によるコンピュータ内部の状態変化
  → Turing machine 【計算基礎論 (2Y1Q)】
- プログラム = 逐次的な手順を記述

(4) 関数型プログラミング

- 計算 = 関数適用 → λ 計算【計算基礎論 (2Y1Q)】
- プログラム = 関数の合成によってより複雑な関数の定義を記述

(5) 論理型プログラミング
- 計算 = 定理証明
  → 述語論理【情報論理 (2Y3Q)】
- プログラム = 物事の関係を公理として記述

(6) オブジェクト指向プログラミング (上 3 つとは直交した概念)
→ 【オブジェクト指向プログラミング (2Y4Q)，オブジェクト指向設計 (3Y3Q)】
- 計算 = オブジェクト間のメッセージ交換
- プログラム = オブジェクトの定義を記述

1.4 プログラミング言語の構文と意味

(1) 構文：プログラミング言語の文法的なきまり

```
C     FORTRAN
      DO 15 I = 1, 100
       ...
   15
```

```
/* C */
for (i=1; i<=100; i++)  ...
```

(2) 意味：記述されたプログラムによる具体的な指示 (コンピュータにやらせる手順)
変数 i の値を 1 から 100 まで 1 ずつ増加してブロック内を実行する

1.5 Scheme — 特徴 —

(1) 構文と意味が単純で明解 → 本質への近道
(2) プログラミングに関する主要概念を含む e.g. λ 抽象，閉包，遅延評価，継続，高階手続き ...
(3) 対話的利用が可能 (コンパイル・フリー)
(4) 関数型言語
(5) 記号処理が容易 ⇔ 数値計算
(6) 拡張が容易
(7) 括弧が多い → 慣れが必要
(8) 実用的でない?

1.6 プログラミング言語の道具立て

(1) **基本要素**：言語が扱う最も単純なモノ e.g. 0, 123，+, * ...
(2) **組合せ**：単純なモノから複雑なモノ (合成物) を作るしくみ
(3) **抽象化**：モノに命名し，名前で参照するしくみ

1.7 式 (組合せ)

式	:	(	+	13	4	)
構文：		(	<要素>	<要素>	<要素>	)
意味：			<手続き>	<引数>	<引数>	
			足し算	十進数 13	十進数 4	

(1) Scheme の解釈系は式を読み込み (Read)，評価し (Eval), 結果を表示 (Print)する．
→ 読込み-評価-印字ループ (REPL)
(2) () の中身を要素という．
(3) 要素は式でもよい．→ 式の入れ子

(4) 式の表記
- 前置形式：Scheme e.g. `(+ 137 349)`
- 中置形式：通常の数学記法 e.g. `137+349`
- 後置形式：HP の電卓 (逆ポーランド記法 (RPN)=日本語記法) e.g. `137 349 +`

1.8 名前と環境

(1) モノに名前を付けるには `define` を使う
構文：`(define <名前> <名前をつけるモノ>)`

```
(define pi 3.14159)
(define radius 10)
(* 2 pi radius)
```

(2) 名前に使える文字は空白と括弧とセミコロン以外のほとんどの文字[2]

(3) 第 2 引数には式も書ける．式の値に名前をつけることになる．

(4) 名前とモノの対応表 → 環境

(5) Scheme の処理系を起動すると大域環境 (トップレベル) が自動的に用意される．端末から `define` でつけた名前は大域環境に記録される．

1.9 式の値の計算：評価の規則

(1) 評価 (evaluation)：構文 (式) からその意味 (値) を計算する
  (a) 数字列 ⇒ 数字列が表わす数値 (デフォルト十進数)
  (b) 名前 ⇒ その環境でその名前を付けられたモノ
  (c) 式 ⇒ 以下の手順に従って計算する値
    1. 式の各要素の評価
    2. 最左要素の値 (手続き) を残りの要素の値 (引数) に適用
  (d) 特殊形式：その特殊形式に依存して決まる e.g. `define`

(2) Scheme の動作はこの規則によって規定される

(3) 評価の規則は Scheme の基本手続き `eval` として実装されている．
`(eval <評価する式> <環境>)`

1.10 特殊形式

(1) `define` による名前付けは普通の組合せではない

(2) Scheme の式には評価のやり方に一部例外がある → 特殊形式

1.11 Scheme の道具立て (ここまでのまとめ)

- 基本手続きや数は解釈系に最初から用意されている

	データ	手続き
基本要素	数	算術演算
組合せ	—	式
抽象化	命名	—

Q 1.1 式 `(+ 2 3)`, `(* (+ 2 3) 4)` をそれぞれ評価する過程と評価結果の値を答えよ．また，それぞれの式を中置形式 (通常の算術式) で書き直せ．

Q 1.2 `define` が特殊形式でなければならない理由を説明せよ．

[2] 実際には，`[]{}|.#\'"‘,@` なども避けた方が無難．

1.12 λ 計算 (復習：【計算基礎論 (2Y1Q)】)

(1) λ 抽象

$f(x) = x + 1 \rightarrow \lambda x.x + 1$

(2) 束縛変数 vs 自由変数

$\lambda x.x + y$

(3) α 変換

$\lambda x.x + 1 \overset{[x:=y]}{\longrightarrow} \lambda y.y + 1$

(4) β 簡約

$\lambda x.x + 1 \overset{[x:=5]}{\longrightarrow} 5 + 1$

$((\lambda x.x + 1)\ 5) \Rightarrow 5 + 1$

1.13 手続きの抽象化

(1) 手続き：計算の方法を記述したもの

e.g. `(* 5 5)`, `(* 3 3)`, `(* 25 25)`, ...

(2) ラムダ抽象 (λ 記法)

`(lambda (x) (* x x))` → 「引数を 2 乗する」手続き cf. $(\lambda x.x^2)$

一般形：`(lambda (<仮引数の並び>) <本体>)`

<本体>はひとつ以上の式の並びで，計算結果は最後 (最右) の式の値となる．

1.14 lambda 式 (特殊形式)

(1) lambda 式を評価した値は手続き (lambda 式=手続きではない)

`(lambda (x) (* x x))` ⇒ `#<procedure>` (引数を 2 乗する手続き)

`((lambda (x) (* x x)) 5)` ⇒ `25` (手続きの適用；β 簡約)

(2) 手続きにも名前をつけることができる → 合成手続き

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

1.15 糖衣構文 (syntax sugar)

```
(define (<名前> <仮引数の並び>) <本体>)
```

≡ `(define <名前> (lambda (<仮引数の並び>) <本体>))`

e.g. `(define (square x) (* x x))`

≡ `(define square (lambda (x) (* x x)))`

- 手続きに名前を付けたら，基本手続きと同じように使える．

1.16 適用の規則 (置換モデル)

(1) 基本手続きは，そのまま適用 (解釈系の定義にしたがって計算)

e.g. `+`は足し算をする

(2) 合成手続きは，仮引数を実引数で置き換えて手続きの本体を評価[3]

(3) 適用の規則は Scheme の基本手続き `apply` として実装されている．

e.g. `(apply <手続き> <引数のリスト>)`

1.17 条件式

(1) Boolean 定数：`#t` (真) と`#f` (偽)[4]

(2) 述語：真偽値を返す手続き e.g. `>`, `>=`, `=`, `<=`, `<`, `zero?`, `even?`, `odd?`, ...

[3] 評価の規則 (1.9) と相互呼び出しになっている点に注意しよう．

[4] 多くの処理系では`#f` 以外はすべて`#t` とみなされる．

(3) 論理演算子：and, or, not (and, or は特殊形式)

(4) if 式：(if p e_t e_f)[5]
p を評価し，真なら e_t を評価した結果を，偽なら e_f を評価した結果を if 式の値とする．if 式は特殊形式である．

(5) cond 式：
(cond (p_1 $e_{11}\ldots e_{1N_1}$)
(p_2 $e_{21}\ldots e_{2N_2}$)
$\vdots$
(p_n $e_{n1}\ldots e_{nN_n}$))
述語 $p_1\ldots p_n$ を上から順に評価し，最初に真になった p_i に対応する $e_{i1}\ldots e_{iN_i}$ を左から順に評価し，最後の式 e_{iN_i} の値を cond 式の値とする．cond 式は特殊形式である．

Q 1.3 評価の規則と適用の規則を使って (+ 2 (square 5)) を評価せよ．square は 1.15 で定義したものである．

Q 1.4 以下の論理式を Scheme の式に書き直せ．
$(x > 0 \wedge y < 100) \vee z \neq 0$

[5] e_f はなくてもよい．

付録：λ 計算 (【計算基礎論】(2Y1Q) の復習)

(1) 動機：関数 $f(x) = x + 1$ を考える．$f(x)$ は何か? $f(a)$ は何か?
→ どういう計算をするかという関数と関数の値の表記の区別をしたい

(2) λ 記法で書くと $(\lambda x.x + 1)$ となる[6]

(3) 多変数の場合：$f(x, y, z) = x + y + z$ は $(\lambda xyz.\ x + y + z)$ と書く
これはまた，$(\lambda x.(\lambda y.(\lambda z.\ x + y + z)))$ と書ける．これをカリー化 (currying) という．

(4) ラムダ項 (λ term) の定義 (構文)：M, N はラムダ項
  (a) 変数 e.g. x
  (b) ラムダ抽象 e.g. $(\lambda x.M)$
  (c) 関数適用 e.g. $(M\ N)$：N (実引数) に M (手続き) を適用する

(5) 略記法
  (a) $(M_1 M_2 \ldots M_n) \equiv (((M_1 M_2) \ldots) M_n)$ → 関数適用は左結合
  (b) $(\lambda x_1 x_2 \ldots x_n.M) \equiv (\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.M))))$ → ラムダ抽象は右結合
  (c) $\lambda x.M\ N \equiv \lambda x.(M\ N)$ → 関数適用はラムダ抽象より結合の優先度が高い

(6) 束縛変数と自由変数
  (a) ラムダ項 $(\lambda x.M)$ において変数 x は束縛されているといい，x を束縛変数という．
  (b) 束縛されていない変数を自由変数という．ある変数の出現が自由変数であるかどうかは，上述のラムダ項の定義に対応して以下のように定義できる．
    i. ラムダ項が変数なら自由変数
    ii. ラムダ項が $(\lambda x.\ M)$ なら M 中の x 以外の変数が自由変数
    iii. ラムダ項が $(M\ N)$ なら M 中の自由変数と N 中の自由変数の和集合が自由変数

(7) α 変換：
ラムダ項中の束縛変数を別の名前の変数に置き換えても同じ項を表わす．ここで，M 中の自由変数 v をすべて変数 w に置き換えた式を $M[v := w]$ と書くことにすると α 変換は以下のように書ける．

$$\lambda v.M \xrightarrow{\alpha} \lambda w.M[v := w]$$

ただし，以下の 2 つの条件を同時に満たす必要がある．
  (a) M 中に w が自由変数として出現しない
    e.g. $(\lambda x.x + y)[x := y] \xrightarrow{\alpha} (\lambda y.y + y)$ → 自由なはずの y が束縛されてしまうのでダメ．
  (b) $v \to w$ の置換により M で新たに w が束縛されない．
    e.g. $(\lambda x.(\lambda y.x))[x := y] \xrightarrow{\alpha} (\lambda y.(\lambda y.y))$ → y が内側のラムダで束縛されてしまうのでダメ．

(8) β 簡約：$((\lambda x.M)\ N) \xrightarrow{\beta} M[x := N]$
ただし，$[x := N]$ の置換によって N 中の自由変数が新たに束縛されてはならない[7]．また，置換は M 中の束縛されている x についてのみおこなう．
  e.g. $((\lambda x.x + 1)\ 5) \Rightarrow 5 + 1$
    $((\lambda xy.x - y)\ 5) = ((\lambda x.(\lambda y.x - y))\ 5) \Rightarrow (\lambda y.5 - y)$ ($(\lambda x.x - 5)$ ではないので注意)
    $((\lambda x.(x\ (\lambda y.y)))\ (\lambda z.z)) \Rightarrow ((\lambda z.z)\ (\lambda y.y)) \Rightarrow (\lambda y.y)$

(9) 高階関数：関数を引数とする，あるいは値として返す関数

[6] 厳密には $x + 1$ は以下で定義する λ 項ではないが，細かい点は目をつぶって以下の λ 記法の例の中ではこれも λ 項のように扱う．
[7] x を仮引数 (formal parameter)，N を実引数 (actual parameter) という

e.g. $(\lambda x.(\lambda y.x - y))$
$(\lambda f.(\lambda x.(f\ (f\ x))))$

(10) これ以上 β 簡約によって書き換えできない式を正規形という.

$\mathcal{Q}$ 1.5 以下の α 変換をおこなえ.

(a) $(\lambda x.x + y)[x := y]$
(b) $(\lambda x.(\lambda y.x))[x := y]$

$\mathcal{Q}$ 1.6 以下の式を β 簡約して正規形にせよ.

(a) $((\lambda x.(x\ (\lambda x.x)))\ (\lambda z.z))$
(b) $((\lambda f.(\lambda x.(f\ x)))\ (\lambda y.\ x + y))$
(c) $((\lambda x.(\lambda y.x - y))\ 5)$
(d) $((\lambda f.(\lambda x.(f\ (f\ x))))\ (\lambda z.z^2))$