

JAKLDの使い方

～ Scheme入門 ～

計算機科学コース
湯浅研究室
助教 馬谷 誠二

umatani@kuis.kyoto-u.ac.jp

<http://www.yuasa.kuis.kyoto-u.ac.jp/~umatani/>

(一部, メディアセンター 平石先生の資料より引用)

本日の内容

- Schemeプログラミング入門
 - JAKLD処理系を実際に使いながら
 - 教科書の1.1.6節まで + α
 - REPL
 - 評価と副作用(入出力など)
 - 関数呼出し, 関数定義, 再帰的定義
 - 基本的なデータ型(特にリストとシンボル)
 - クォート
 - リスト操作
 - データ構造(コンス・セル)
 - などなど...

Lisp言語

- John McCarthy によって発明(1958年)
- FORTRAN(1957年)に次いで2番目に古い
- 特徴
 - リスト処理が得意(List Processor)
 - 対話環境
 - 動的型付け, closureオブジェクト, ...
 - 「やりたいこと」だけに集中してプログラムが書ける
 - *rapid prototyping*
 - LispのプログラムをLisp自身で扱うことができる
 - 書きたいプログラムに合わせて言語自体をカスタマイズ可能
 - “You can write your own language *on* Lisp”
 - (Paul Graham, “On Lisp”, <http://www.paulgraham.com/onlisp.html>)

Scheme

- Lispの方言の一つ
- プログラミング言語として本当に必要な部分だけをできるだけコンパクトにまとめた仕様
 - 言語仕様(R5RS)は50ページ
 - (2007/09に成立した新仕様(R6RS)で3倍以上に増え, 一部反発)
 - C(C99)は538ページ
 - Common Lisp(第2版)は1029ページ
- 継続オブジェクト
 - 実行中の処理の「残りの計算」をプログラムで扱える
- 真の末尾再帰呼び出しをサポート
 - 繰り返し構文(Cでいうwhileなど)を持たない

Why Scheme?

- なぜ 1 回生から Scheme を習うのか(個人的主観)
 1. 関数型プログラミングは分かりやすい,かつ強力
 - 最近主流の言語の理解にもつながる
 2. 言語仕様がコンパクト
 3. 対話環境 } 習得が容易
- 4. (SICP が使ってるから ☺)
- 本講義では湯浅先生作成の Scheme 処理系 JAKLD (JAVA KUMIKOMI-you LISP DRIVER) を使用
 - Java (JVM)上で動作 → 使い始めるのが簡単
 - 教科書に出てくるコード(図形言語含む)を**完全サポート**

JAKLD導入

AndroScheme
なんて素敵なものも!

- 入手 & インストール
 - 前提条件: Java 処理系がインストールされている事
- 湯浅先生の下記のページから jar ファイルを入手
<http://www.yuasa.kuis.kyoto-u.ac.jp/~yuasa/jakld/index-j.html>
- 対話環境の起動

```
$ java -jar jakld.jar
JAKLD for SICP (October 10, 2008)
(c) Copyright Taiichi Yuasa, 2002. All rights reserved.
>
```

- 終了

```
> Ctrl-D (Controlキーと D キーを同時に入力)
Sayonara
$
```

REPL (Read-Eval-Print Loop)

- プロク `ラムを読み (read),評価 (evaluate) し, 結果を表示 (print) を繰り返す (loop)
- 例: 簡単な算術式

```
> (+ 1 2)
3
> (<< (* (+ 1 2) 3) 10)
#t
>
```

- Schemeではすべて**前置記法 (prefix notation)** で書く
 - (<operator> <operand1> <operand2> ...)

Hello! World in C

新しいプログラミング
言語を習う時, 最初
は始める
すること

```
#include<stdio.h>
int main (void)
{
    printf ("Hello! World¥n");
    return 0;
}

% gcc hello.c
% ./a.out
Hello! World
```

Hello! World in Scheme

- Schemeならこんなに簡単

```
% java -jar jakld.jar
> (begin
  (display "Hello World!")
  (newline))
Hello! World
#t
```

Hello world: つづき

- (display <式>)
 - <式>の評価結果を表示
 - 任意の型の値を表示可能 (人間にとってなるべく見やすい表現)
- (newline)
 - 改行を表示
- (begin <式1> <式2> ... <式n>)
 - 左から右へ順に評価し、最後の式の評価結果をbegin式の評価結果として返す

Hello world: つづき

- さきほどのJAKLDの応答を見直すと:

```
% java -jar jakld.jar
> (begin
  (display "Hello World!")
  (newline))
Hello! World           :画面への表示
#t                     : (newline) の評価結果
```

変数の定義と代入

- (define <変数名> <式>)
- (set! <変数名> <式>)

```
> (define greeting "Hello World!")
ok
> (begin (display greeting) (newline))
Hello World!
#t
> (define x 1)
ok
> (+ x 1)
2
> (set! x 10)
10
> (+ x 1)
11
```

評価と副作用

- 評価: 式の値を求める計算プロセス
- 副作用: 評価に伴う副次的効果(入出力など)
- $(* (+ 1 2) 3) \Rightarrow (* 3 3) \Rightarrow 9$
- `(display "Hello World!")`
[Hello World を画面に表示]
⇒ "Hello World!"
- `(define greeting "Hello World!")`
[変数 `greeting` を文字列に束縛]
⇒ **未定義** (何か仮定してはいけない)
- `set!` も同様

手続き(関数)定義

- 何度も同じコードを入力するのは手間
- 部分的に異なるだけのコードをまとめたい
→ **手続きによる抽象化**
- 手続き定義については, 奥乃先生の第 1 回講義 のとおり
 - 復習: `abs`

```
> (define (abs x)
      (if (>= x 0) x (- x)))
ok
> (abs -3) ; or (abs (- 3))
3
```

局所変数定義

- `(let ((<変数1> <式1>)
 ...
 (<変数n> <式n>))
 <本体>)`
 - <本体>中でのみ有効な変数を定義

```
> (define x 1)
ok
> (let ((x 10))
      (+ x 1))
11
> (+ x 1)
2
```

Hello world その2

```
> (define (helloworld-1)
      (display "Hello World!")
      (newline))
ok
> (define (helloworld-2)
      (display "Hello")
      (display " ")
      (display "World!")
      (newline))
ok
> (define (helloworld-3)
      (define (hw greeting)
        (display greting) (newline))
      (hw "Hello World!"))
ok
> hw
RuntimeException: undefined variable hw
at top-level
```

組み込みデータ型

- 数: 1, 10, -3, 3.14, 3.0e8
- 文字列: " Hello world!"
- 文字
- リスト
- シンボル

まずは文字列操作について少しだけ

文字列操作関数 (1)

- (string-length <文字列>): 文字列の長さを返す

```
> (string-length "Hello World!")
12

> (string-length (+ 1 2))
RuntimeException: 1st argument 3 to string-length
not String object

>
```

文字列操作関数 (2)

- (string-append
 <文字列1> <文字列2> ... <文字列n>)
- 文字列の連結

```
> (define (helloworld-4)
  (string-append
    "H" "e" "l" "l" "o" " " "W" "orld!"))
```

- ちなみに:

```
> (string-append "Hello World!")
"Hello World!"
> (string-append)
""
```

文字列操作関数 (3)

- (substring <文字列> <i> <j>)
- i 番目から j 番目までの文字を含んだ部分文字列を返す
(j は省略可能)

```
> (define (helloworld-5)
  (let ((sentence "Hello again. Small world!"))
    (display (substring sentence 0 6))
    (display (substring sentence 19))
    (newline)))
```

組込みデータ型

- 数: 1, 10, -3, 3.14, 3.0e8
- 文字列: " Hello world!"
- **文字**
- リスト
- シンボル

文字

- **#%c**: 文字の**ならび**ではなく**1文字だけ**からなるデータ
- (string <文字1> <文字2> ... <文字n>)

```
> (define (helloworld-6)
  (display (string #%H #%e #%l #%l #%o
                 #% #%W #%o #%r #%l #%d #%!)))
```

- (string-ref <文字列> <i>)

```
> (string-ref "Hello World" 6)
#%W
```

- **なぜ素直に** (string H e l l o ...) と書けない?
 - 試せばすぐ分かる
 - シンボル (後述)

組込みデータ型

- 数: 1, 10, -3, 3.14, 3.0e8
- 文字列: " Hello world!"
- 文字
- **リスト**
- **シンボル**

Scheme (Lisp) プログラミングで最も重要なデータ型はこの2つです!

リスト

- **任意のデータ**の**ならび**(⇔ 文字列は文字の**ならび**)
 - (list <式1> <式2> ... <式n>)

```
> (list "Hello" " " "World!")
("Hello" " " "World!")
```

```
> (list 1 3.14 "Hello" #%W)
(1 3.14 "Hello" #%W)
```

```
> (list)
() ; 空リスト
```

```
> nil
() ; 空リスト
```

リスト操作関数 (1)

• 基本操作関数

- (cons <式> <リスト>): <式>の値を<リスト>の先頭に追加
- (car <リスト>): <リスト>の先頭の要素を返す
- (cdr <リスト>): 先頭の要素を除いた<リスト>を返す

```
> (define l (list "Hello" " " "World!"))
ok
> (cons "Again, " l)
("Again, " "Hello" " " "World!")
> (car l)
"Hello"
> (cdr l)
(" " "World!")
```

- cadr, caddr, caddr, ...: 便利な記法

例

```
> nil
()
> (cons 1 nil)
(1)
> (cons 2 (cons 1 nil))
(2 1)
> (cons 3 (cons 2 (cons 1 nil)))
(3 2 1)
> (define l (cons 3 (cons 2 (cons 1 nil))))
ok
> (car l)
3
> (cadr l) ; (car (cdr l))
2
> (caddr l) ;; (car (cdr (cdr l)))
1
```

リスト操作関数 (2)

```
> (define l (list "Hello" " " "World!"))
ok
> (length l)
3
> (append (list "You" " " "said," " ") l)
("You" " " "said," " " "Hello" " " "World!")
> (reverse l)
("World!" " " "Hello")
> (map abs (list 1 -2 3 4 -5))
(1 2 3 4 5)
> (begin (foreach display l) (newline))
Hello World!
#t
```

上の関数 (と先程のlist関数も) はすべて自分で定義することも可能 (この講義ですぐに学びます)

リストを結合する関数

- (append (list 1 2 3) (list 4 5 6))
⇒ (1 2 3 4 5 6)
- 「リストを結合する」の定義は?
 - (append () y) = y
 - (append ' (a b ...) y) = (cons a (append ' (b ...) y))
- 答:

```
> (define (my-append x y)
  (if (null? x)
      y
      (cons (car x) (my-append (cdr x) y))))
```

閑話休題 その1

- ファイルからのプログラムの読み込み
- これまでの手続き定義が全て jakldtut.scm という名前のファイルに入っているとする

```
> (load "jakldtut.scm")
"jakldtut.scm"

> (helloworld-2)
Hello World

>
```

(注) ファイル中に式を書くだけではJAKLDは評価結果を表示しません

クォート (引用, quote)

- プログラムとデータの区別
 - (+ 1 2) ... 「1と2を足す」プログラム
 - (1 2 3 4) ... 4つの整数からなるデータ
- ...というのは人間の勝手な解釈!
- REPL は何でもプログラムと解釈する (デモ)
- (quote <式>)
 - <式>をデータとして扱うためのスペシャルフォーム
 - 通常関数呼出しと異なり, <式>を評価せずそのまま返す
 - 省略記法: ' <式> = (quote <式>)
- その他のスペシャルフォーム: define, ...
- 数, 文字列, 文字なんかは?

Hello world その3

```
> (define (helloworld-7)
  (foreach display (list "Hello" " " "World!") ))
  (newline))
> (define (helloworld-8)
  (foreach display '("Hello" " " "World!") ))
  (newline))
> (define (helloworld-9)
  (foreach display
    '("H" "e" "l" "l" "o" " " "W" "o" "rld!") ))
  (newline))
> (define (helloworld-10)
  (foreach display
    '#(H e l l o " " W o rld!)))
  (newline))
```

シンボル (記号)

```
> 'hello
hello
```

- 文字と何がちがう? → 1文字以上書ける
- 文字列と何がちがう? → 分割や結合ができない
- といった些細な事ではない本質的な違い:
- シンボルをプログラムとして見ると変数になる

```
> (define hello "Hello World!")
> (display hello)
Hello World!
```

メタプログラミング
(プログラムを操作するプログラム)

- 逆に, プログラムをクォートするとデータになる

```
> (define prog '(list "Hello" " " "World!"))
> (car prog)
list ; これはシンボル
```

シンボル (記号): つづき

• 備考:

```
> (for-each display '("Hello" " " "World!"))
```

- 実は for-each, displayも通常の変数
- 評価した結果の値が**手続きオブジェクト**

```
(define for-each <手続きオブジェクト>)
```

```
(define display <手続オブジェクト>)
```

→ <手続オブジェクト>の詳細は次回以降の講義にて (lambda式)

Hello world その4

- クォートによるシンボルデータと変数の両方を使った (作為的な) 例:

```
> (define (helloworld-11)
  (let ((space " ")
        (wor "Wor"))
    (for-each display
              (append '(Hello)
                      (list space wor)
                      '(!))))
  (newline))
ok
> (helloworld-11)
Hello World!
#t
```

再帰的な手続きの定義

- 「中身を知らないfor-eachに頼りたくない！」という方へ
- もちろん,自分でもリストに対する Hello World プログラムを定義できます
- 復習: factorial

```
> (define (fact n)
  (if (<= n 0)
      1
      (* n (fact (- n 1)))))
ok
```

```
> (fact 16)
20922789888000
```

Hello world その5

```
> (define (helloworld-12)
  (define (display-list xs)
    (if (null? xs)
        (newline)
        (begin (display (car xs))
                 (display-list (cdr xs)))))
  (display-list '("Hello" " " "World!")))
ok
```

```
> (helloworld-12)
Hello World!
#t
```

仕様と実装

- 1つの要求仕様に対して様々な実装が有り得る
- helloworld-1~helloworld-12はどれも正解
- 現実には、有り得る実装から最良なものを選択すべき
 - 評価指標
 - 実行速度, 実装コスト(手間), 可読性, メンテナンス性, 消費電力, など
- 実行効率と実装コストは一般にはトレードオフ

```
(define (slow-reverse x)
  (if (null? x)
      ()
      (append (slow-reverse (cdr x))
                (list (car x))))))

(define (fast-reverse x)
  (fast-reverse-aux x ()))

(define (fast-reverse-aux x acc)
  (if (null? x)
      acc
      (fast-reverse-aux (cdr x) (cons (car x) acc))))
```

閑話休題 その2

- (trace <関数名>)
- 末尾再帰最適化を確認してみましょう
- 非末尾再帰版fact

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

ok

閑話休題 その2

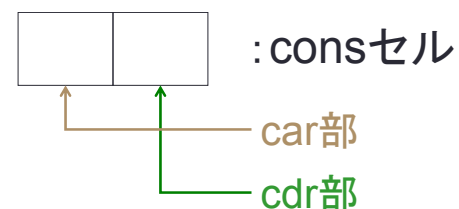
- 末尾再帰版fact

```
> (define (fact-iter prod counter max-count)
  (if (> counter max-count)
      prod
      (fact-iter (* counter prod)
                  (+ counter 1)
                  max-count)))
```

ok

consセル(1)

- 例1: (we eat rice) の内部表現

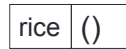


consセル(2)

- (cons <データ1> <データ2>):
car部とcdr部がそれぞれ<データ1>,<データ2>
であるconsセルを作る

> (cons 'rice '())

(rice)



> (cons 'eat (cons 'rice '()))

(eat rice)



> (cons 'we (cons 'eat (cons 'rice '())))

(we eat rice) = (list 'we 'eat 'rice)



consセル(3)

- 例2:
(define x ' (we eat rice))
(define y (cons ' they (cdr x)))

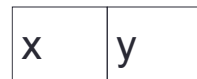


consセル(4)

- ドット・ペア

> (cons 'x 'y)

(x . y)



- ドット・リスト

> (cons 'x (cons 'y 'z))

(x y . z)

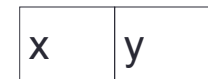


consセル(5)

- ドット・ペア

> (cons 'x 'y)

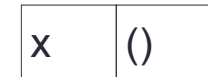
(x . y)



> (cons 'x '())

(x)

(x . ())の略記



> (cons 'x (cons 'y '()))

(x y)

(x . (y . ()))の略記



consセル(6)

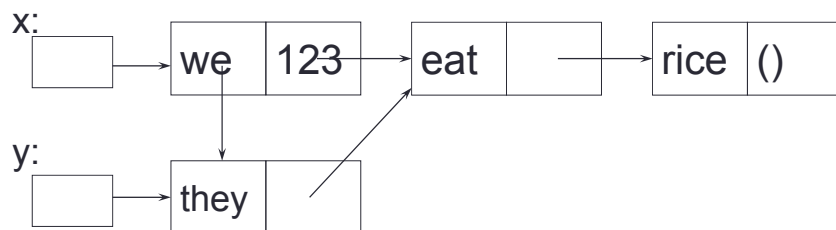
- 破壊的操作

```
> (set-cdr! x 123)
```

```
x: (we . 123)
```

```
> (set-car! x y)
```

```
x: ((they eat rice) . 123)
```



キーボードからの入力

- 入力関数

```
> (read)
```

```
abcde ← キーボードから入力
```

```
abcde システムの出力(read関数の返り値)
```

```
> (define (prompt-fact)
```

```
  (display "input: ")
```

```
  (fact (read)))
```

```
prompt-fact
```

```
> (prompt-fact)
```

```
input: 11 ← キーボードから入力
```

```
39916800 ← プロンプト(display関数が出力)
```

eval

- (eval <式> <環境>)

- <式>: 評価したいプログラムのデータ表現

- <環境>: 変数束縛の集合(+, list, map, ...)

```
> (define prog1 '(+ (* 7 3) 4))
```

```
ok
```

```
> (eval prog1)
```

```
25
```

```
> (define prog2 (list '- (cadr prog1)  
                      (caddr prog1)))
```

```
ok
```

```
> prog2
```

```
(- (* 7 3) 4)
```

```
> (eval prog2)
```

```
17
```

REPLの定義 (簡易版)

```
> (define (repl)  
  (display ">> ")  
  (let ((input (read)))  
    (let ((result (eval input)))  
      (display result)  
      (newline)  
      (repl))))
```

練習問題(小テスト)

- 以下の中から1~2個選んで、関数定義を書け
 - `but-last`

```
(but-last '(1 2 3 4 5)) ⇨ (1 2 3 4)
(but-last '()) ⇨ ()
```
 - `assoc`

```
(assoc 'hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))
        ⇨ (hgo IntroAlgDS))
(assoc hgo '((ishi Gairon) (iso math) (tom HW)) ⇨ #f
```
 - `length`

```
(length '(1 2)) ⇨ 2           (length '(1 2 3 5)) ⇨ 4
(length nil) ⇨ 0
```
 - `copy`

```
(copy '(1 2)) ⇨ (1 2)
(copy '((1 . 2) . (3 . 4))) ⇨ ((1 . 2) 3 . 4)
```
 - `flatten`

```
(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
```
- 講義の感想を書いてください
1&2 を出席票に書いて提出

以上

- 質問, コメントは:
umatani@kuis.kyoto-u.ac.jp
まで.

宿題: 10月25日正午締切

- リストの長さを求める2種類の手続きの再帰型(my-length)と反復型(my-length-i)をfileに作成せよ.
my-length.scm
- 2種類のmy-length の手続きを実行し, その出力結果を求めよ.
 - (my-length ()), (my-length-i ())
 - (my-length '(1 2 3 4)), (my-length-I '(a b c d e f g))等.
- 2種類のmy-length の手続きの計算量を求めよ.
- 説明と出力結果, 及び3について, レポートをlatexで作成し, pdf で提出すること. (1回生は紙でも可)
- プログラムファイルとレポートのpdf を
[SICP-3@zeus.kyoto-u.ac.jp](mailto:SICP-3@zeus.kuis.kyoto-u.ac.jp) に送付
 - 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)
- 質問, コメントは: umatani@kuis.kyoto-u.ac.jp まで. 51