JAKLDによる

Scheme入門

学術情報メディアセンター
スーパーコンピューティング研究分野
助教　　平石 拓
tasuku@media.kyoto-u.ac.jp
http://super.para.media.kyoto-u.ac.jp/~tasuku/

本講義の内容

- Lisp (Scheme)の基本的な機能を一通り解説
  - SICPの1.1〜1.1.6
  - 1.1.6までに（あるいはSICPそのものに）載ってないが，知っておいたほうがいいSchemeに関する知識
    - リスト (lists)
    - 局所変数 (local variables: let)
    - コンス・セル (cons cells)
    - ファイル入出力 (file I/O)
    - トレース (tracing)

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)

Hello! World in C

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

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

Hello! World in Scheme

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

Lispの方言

- Common Lisp
- Scheme
- Emacs Lisp
- AutoLisp
- Lisp1.5
- MACLISP
- ISLisp
- …

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

起動・終了

```
% java –jar jakld.jar ---------- 起動
JAKLD for SICP (October 10, 2008)
(c) Copyright Taiichi Yuasa, 2002.  All rights reserved.
> (+ 3 4)
7
> Ctrl+C
Bye.                ---------- 終了
%
```

処理系によっては(bye), (exit), (quit)

対話環境 (REPL: Read Eval Print Loop)

```
> (+ 3 4)
7 ---------- (+ 3 4)の評価値
> (* (+ 1 2) (- 10 7))
9 ---------- (* (+ 1 2) (- 10 7))の評価値
> 1234
1234 ---------- 1234の評価値
> (< (* 3 3) 10)
#t ---------- (< (* 3 3) 10)の評価値
```

変数定義(1)

```
> (define x 10) ---------- xという名前の変数を定義
x
> x
10
> (* x (+ x 3))
130
> (set! x 24) ---------- xの値を変更
> x
24
```

変数定義(2):局所変数

```
> (let ((x 20)
        (y 10))
    (* x (+ x y)))
300
> x
Error:  x is an unbound symbol.
```

関数定義

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

> (square 10)
100
> (square (* 3 4))
144
```

関数の本体
関数のパラメータ
関数の名前

階乗の計算

- n! を求める関数

```
> (define (fact n)
    (if (= n 0)
        1          ---------- n=0の時
        (* n (fact (- n 1)))))   ---------- n=0でない時

> (fact 10)
362800
```

Fibonacci数の計算

- 1, 1, 2, 3, 5, 8, 13, …
- fib(n) = fib(n-1) + fib (n-2)

```
> (define (fib n)
      (if (< n 2)        ; fib(0) = fib(1) = 1
          1
          (+ (fib (- n 1)) (fib (- n 2)))))
fib
> (fib 10)
89
```

引用符（1）

- (quote 〈式〉) ・・・スペシャル・フォーム

```
> (quote x)
x
> (quote (+ 3 4))
(+ 3 4)
> (define x (quote y))
> x
y
```

引用符（2）

- ’〈式〉 でも同じ意味

```
> 'x
x
> '(+ 3 4)
(+ 3 4)
> (define x 'y)
> x
y
```

リスト

- Lispにおける最も重要なデータ型の1つ
  - Lisp = List Processor
- データ（要素）の“並び”を表す
  - (1 2 3 4 5 6 7 8 9 10)
  - (we eat rice)
  - ((lions 0.543) (buffaloes 0.524) (fighters 0.514) (marines 0.510) (eagles 0.461) (hawks 0.454))

リストの生成（1）

```
> (list 1 2 3 4)
(1 2 3 4)
> (list 'w 'x (list 'y 'z))
(w x (y z))
> (define x 4)
> (list x (* x 5))
(4 20)
> (list)
()          ---------- 空リスト
```

リストの生成(2)

- リストの要素がわかっている場合は、
  (quote 〈リストを表す式〉) でもよい。

```
> '(x y)
(x y)
> '((x y) 1 2 (a b c))
((x y) 1 2 (a b c))
> '(define (square x) (* x x))
(define (square x) (* x x))
```

(quote (x y)) の略記

リストの要素の参照

```
> (car '(a b c d))
a
> (cdr '(a b c d))
(b c d)
> (car (cdr (cdr '(a b c d))))
c
> (cdr (cdr (cdr (cdr '(a b c d)))))
()
```

リストの先頭要素

先頭要素を除いたリスト

リストへの要素追加

- リストの先頭に要素を追加

```
> (cons 'we '(eat rice))
(we eat rice)
> (cons 'never (cdr '(we eat rice)))
(never eat rice)
> (cons '(a b c) '(x y z))
((a b c) x y z)
> (cons 'single '() )
(single)
```

リストの長さを求める関数

- 「リストの長さ」の定義は？
  - (length ()) = 0
  - (length '(*a b* ...)) = 1 + (length '(*b* ...))

```
(define (my-length x)
  (if (null? x)
      0
      (+ 1 (my-length (cdr x)))))
```

*x*が()か？

リストを結合する関数

- (append '(a b c) '(1 2 3)) → (a b c 1 2 3)
- 「リストを結合する」の定義は？
  - (append () *y*) = *y*
  - (append '(*a b* ...) *y*) = (cons *a* (append '(*b* ...) *y*))

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

リストを逆順に並べ換える関数

- (reverse '(a b c)) → (c b a)
- 「リストを逆順にする」の定義は？
  - (reverse ()) = ()
  - (reverse (*a b* ...)) = (append (reverse '(b ...)) '(*a*))

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

効率は悪い

仕様と実装

- Hello! Worldの様々な実装

```
(define (HelloWorld-1)
  (display "Hello World!")
  (newline) )

(define (HelloWorld-2)
  (display (string-append "H" "e" "l" "l" "o" " " "W" "o" "r" "l" "d" "!"))
  (newline) )

(define (HelloWorld-3)
  (display (string #\H #\e #\l #\l #\o #\  #\W #\o #\r #\l #\d #\!))
  (newline) )

(define (HelloWorld-4)
  (display "Hello")
  (display " ")
  (display "World!")
  (newline) )
```

仕様と実装

- Hello! Worldの様々な実装

```
(define (HelloWorld-5)
  (map display '("Hello" " " "World!"))
  (newline) )

(define (HelloWorld-6)
  (map display (list "Hello" " " "World!"))
  (newline) )

(define (HelloWorld-7)
  (map display '("H" "e" "l" "l" "o" " " "W" "o" "r" "l" "d" "!"))
  (newline) )

(define (HelloWorld-8)
  (map display '(#\H #\e #\l #\l #\o #\  #\W #\o #\r #\l #\d #\!))
  (newline) )

(define (HelloWorld-9)
  (((lambda (x) x) display) "Hello World!")
  (newline) )
```

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-10はどれも正解
- 現実には，有り得る実装から最適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))))
```

リストが要素を含むか判定

- (member 4 '(1 3 5 7 9)) → #f
- (member 5 '(1 3 5 7 9)) → (5 7 9)
- 定義は？

```
(define (my-member x lst)
  (if (null? lst)
      #f
      (if (equal? x (car lst))
          lst
          (my-member x (cdr lst)))))
```

式の評価

- (+ (* 3 2) 5) や (define x 30) なども、それ自身は単なるリスト

```
> (list '+ (list '* 3 2) 5)
(+ (* 3 2) 5)
```

- システムがこのリスト（データ）を「評価」すると、“関数呼び出し式”として処理を行い、値を返す

```
> (eval (list '+ (list '* 3 2) 5))
11
```

- フォーム：システムが評価できるデータ
- フォームでないデータを評価しようとするとエラーになる

フォームの分類（リスト・フォーム）

- リスト・フォーム
  - 関数適用 ・・・ (〈関数〉 〈式1〉 … 〈式n〉)
    1. 〈関数〉と〈式1〉～〈式n〉を評価
    2. 関数を〈式1〉～〈式n〉の評価結果に適用
  - スペシャル・フォーム
    - define, set!, quote, if など特定の記号で始まるリスト
    - それぞれのスペシャルフォームごとに決まった評価方法
  - マクロ・フォーム（関数適用とは違う）
    - マクロを表す記号で始まるリスト
    - プログラマが新しいマクロを定義できる
    - 詳細は省略（ただし，Lispを強力たらしめる最大の特徴）

フォームの分類（リスト以外）

- 記号
  - 変数の値が評価値になる

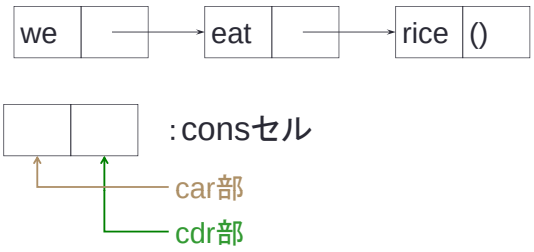
```
> (define x 30)
x
> x
30
> +
#<function +>
```

- その他のデータ（数値，文字列など）
  - それ自身が評価値となる

```
> 123
123
> "abcde"
"abcde"
```

consセル（1）

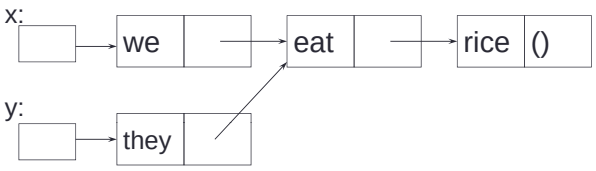
- 例1：(we eat rice) の内部表現



consセル（2）

- 例2：

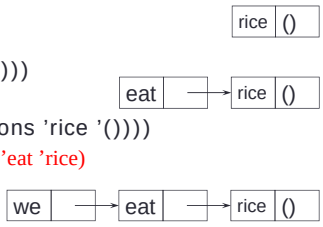
```
(define x '(we eat rice))
(define y (cons 'they (cdr x))
```



consセル（3）

- (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

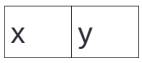
```
> (cons 'rice '())
(rice)
> (cons 'eat (cons 'rice '()))
(eat rice)
> (cons 'we (cons 'eat (cons 'rice '())))
(we eat rice)        = (list 'we 'eat 'rice)
```



consセル（4）

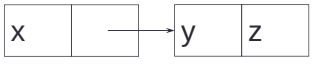
- ドット・ペア

```
> (cons 'x  'y)
(x . y)
```



- ドット・リスト

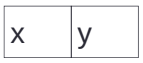
```
> (cons 'x (cons 'y 'z))
(x y . z)
```

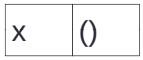


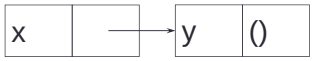
consセル（5）

- 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)

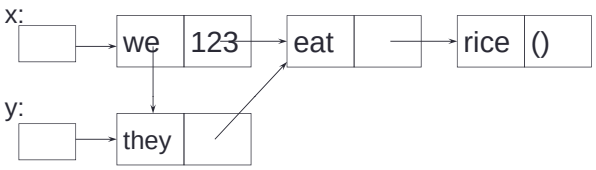
- 破壊的操作

> (set-cdr! x 123)

→ x: (we . 123)

> (set-car! x y)

→ x: ((they eat rice) . 123)

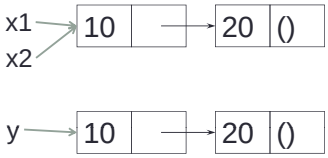


データの比較

- 数値どうしの比較は =, <, >, <=, >= を使うべき
  - (< 3 10) → #t
  - (= 4 5) → #f
- リストどうしの比較
  - (eq? *<e1>* *<e2>*)
    - *<e1>*と *<e2>*が同じオブジェクトなら#t
  - (eqv? *<e1>* *<e2>*)
    - *<e1>* と*<e2>*がeq?の関係か，同じ数値，文字，etc. なら#t
  - (equal? *<e1>* *<e2>*)
    - *<e1>* と*<e2>*が同じ内容（のリスト）なら#t

eq?とequal?の違い

```
> (define x1 (list 10 20))
(10 20)
> (define x2 x1)
> (define y (list 10 20))
(10 20)
> (eq? x1 x2)
#t
> (eq? x1 y)
#f
> (equal? x1 y)
#t
```



equal?の定義

```
(define (my-equal? e1 e2)
  (if (pair? e1)
      (and (pair? e2)
           (my-equal? (car e1) (car e2))
           (my-equal? (cdr e1) (cdr e2)))
      (eqv? e1 e2)))
```

※ 本物のequal?は，文字列や配列にも対応

画面への出力(1)

- 出力関数

> (write '(a b c))

(a b c)(a b c)

システムの出力（評価値）

write関数の出力

> (begin (write '(a b c)) (newline))

(a b c) write関数の出力

#t システムの出力（評価値）

画面への出力(2)

```
> (define (square x)
    (write (list 'x x))
    (newline)
    (* x x))
square
> (square (* 3 4))
(x 12)
144
```

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）

ファイルへ出力

```
> (define out (open-output-file "outfile"))
out
> out
#<port to outfile>
> (write (fact 7) out)        ------- 結果を"outfile"に書き込む
5040
> (newline out)
#t
> (close-output-port out)
#t
```

ファイルから入力

```
> (define in (open-input-file "infile"))
in
> (read in)        ------- "infile"からデータを1つ読み込む
data
> (read in)
#<end-of-file>
> (close-input-port in)        ----- ファイルの終端かチェック
#t
```

ファイル入出力のための関数

- (call-with-output-file 〈ファイル名〉〈関数〉):
  指定されたファイルへの出力ポートを 引数として
  〈関数〉を呼び出し，その返り値を返す．
- (call-with-input-file 〈ファイル名〉〈関数〉):
  指定されたファイルへの入力ポートを 引数として
  〈関数〉を呼び出し，その返り値を返す．

➢〈関数〉は1引数の関数でなければならない。
➢実行終了後、ファイルは自動的に閉じられる。

call-with-output-file

- 例：1^2, 2^2, ..., 99^2 の値をファイルに書き出す。

```
(call-with-output-file "square99.out"
  (lambda (out)
    (let loop ((n 1))
      (if (< n 100)
          (begin
            (write (* n n) out)
            (newline out)
            (loop (+ n 1)))))))
```

call-with-input-file

- 例：ファイルから全てのデータを順に読み込んで画面に表示する。

```
(call-with-input-file "infile"
  (lambda (in)
    (let loop ((dat (read in)))
      (if (not (eof-object? dat))
          (begin
            (write dat)
            (newline)
            (loop (read in)))))))
```

プログラムファイルのロード

- (load 〈ファイル名〉):ファイルに書かれているフォームを順に、全て評価する。

```
> (load "square.scm")
Loading square.scm...
Finished.
"square.scm"
> (square 4)
16
```

関数実行のトレース

- (trace 〈関数名〉):関数のトレースを開始
- (untrace 〈関数名〉):トレースをやめる

標準ではないが，たいていの処理系で使える.

```
> (trace fact)
> (fact 2)
1>(fact 2)
  2>(fact 1)
    |3>(fact 0)
    |3<(fact 1)
  2<(fact 1)
1<(fact 2)
2
```

主な組み込み関数(1)

- 加減乗除
  - (+ 〈数値1〉 … 〈数値n〉)
  - (− 〈数値1〉 … 〈数値n〉)
  - (* 〈数値1〉 … 〈数値n〉)
  - (/ 〈数値1〉 … 〈数値n〉)
  - (remainder 〈整数1〉 〈整数2〉) :割り算の余り (cf. modulo)
- 比較
  - (= 〈数値1〉 … 〈数値n〉)
  - (< 〈数値1〉 … 〈数値n〉)
  - (> 〈数値1〉 … 〈数値n〉)
  - (<= 〈数値1〉 … 〈数値n〉)
  - (>= 〈数値1〉 … 〈数値n〉)

主な組み込み関数(2)

- リスト関係
  - (length 〈リスト〉)
  - (append 〈リスト1〉 … 〈リストn〉)
  - (reverse 〈リスト〉)
- 等号
  - (eq? 〈データ1〉 〈データ2〉)
  - (eqv? 〈データ1〉 〈データ2〉)
  - (equal? 〈データ1〉 〈データ2〉)
- 論理演算
  - (not 〈データ1〉)
  - (and 〈データ1〉 … 〈データn〉)【特殊フォーム】
  - (or 〈データ1〉 … 〈データn〉) 【特殊フォーム】

主な組み込み関数(データ型述語)

- (number? 〈データ〉)
- (integer? 〈データ〉)
- (symbol? 〈データ〉)
- (pair? 〈データ〉)
- (list? 〈データ〉)
- (null? 〈データ〉)
- (string? 〈データ〉)

その他の組み込み関数

- Schemeの仕様書(R^{5}RS，最新よりこちらがおすすめ)
  - http://www.schemers.org/Documents/Standards/R5RS/
  - 50ページしかないので(全ての理解は無理でも)初心者でも読める
  - 日本語版もある
- tus, guile等では以下のヘルプ機能も使える
  - (apropos 〈文字列〉)
    - 〈文字列〉を含む組み込み関数、スペシャルフォーム、マクロの一覧を表示する.

```
> (apropos "list")
dolist
get-host-list
list
list*
list-ref
list-tail
list->string
...
```

練習問題(小テスト)

1. 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

2. 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

宿題:10月26日正午締切

1. Fibonacci数の再帰型と繰り返し型手続きについて，それぞれファイルを作成せよ．
   fib.scm と fib-iter.scm
2. 2種類のFibonacci数の手続きを実行し，fib(10), fib(20), fib(30) の出力結果を求めよ．
3. 2種類のFibonacci数の手続きを使ったfib(n) 実行時に fib が呼ばれる回数を，それぞれ解析的に求めよ
4. 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．(紙でも可)
5. プログラムファイルとレポートのpdf を 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)