JAKLD の使い方

～ Scheme プログラミング入門

計算機科学コース 湯淺研究室
馬谷 誠二

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 1.1.6 節まで ＋ α
    - REPL
    - 評価と副作用（入出力だけ）
    - 関数呼出し，関数定義，再帰的定義
    - 基本的なデータ型（特にシンボルとリスト）
    - クォート
    - リスト操作
    - データ構造（Cons セル）

はじめに

- 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 :-)）
- 本講義では湯淺先生作成の 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 の利用開始

- 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
  - 湯淺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
$
```

2009年10月13日（火） 1回生後期「アルゴリズムとデータ構造」第2回 4

REPL (Read-Eval-Print Loop)

- プログラムを読み（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 では全てが前置 (prefix) 記法
  - (⟨operator⟩ ⟨operand1⟩ ⟨operand2⟩ ...)

2009年10月13日（火） 1回生後期「アルゴリズムとデータ構造」第2回 5

Hello World

- 新しいプログラミング言語を習う時，まずはじめにする事：
  - “Hello World!” を画面に表示
- Scheme ならこんなに簡単！

```
> (begin
    (display "Hello World!")
    (newline))
Hello World!
#t

>
```

2009年10月13日（火） 1回生後期「アルゴリズムとデータ構造」第2回 6

Hello, World ：つづき

- (display 〈式〉)
  - 〈式〉の評価結果を表示
  - 任意の型を表示可能
- (newline)
  - 改行を表示
- (begin 〈式1〉〈式2〉 ... 〈式n〉)
  - 左から右に順に評価し，最後の式の評価結果を返す

Hello World ：つづき

- 先程の JAKLD の応答結果を見直すと:

```
> (begin
    (display "Hello World!")
    (newline))
Hello World!        ; 画面への表示
#t                  ; (newline) の評価結果
```

- 変数定義 (define 〈変数名〉〈式〉):

```
> (define greeting "Hello World!")
ok

> (begin (display greeting) (newline))
Hello World!
#t
```


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（入出力等）
- (* (+ 1 2) 3) ⇒ (* 3 3) ⇒ 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）

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
```


Hello World その 2

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

```
> (define (HelloWorld-2)
    (display "Hello")
    (display " ")
    (display "World!")
    (newline))
```

```
> (define (HelloWorld-3)
    (define (HW greeting)
      (display greting)
      (newline))
    (HW "Hello World!"))
```


組み込みデータ型

- 整数
- 文字列
- 文字
- リスト
- シンボル

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)))
```

- (let ((⟨変数 1⟩ ⟨式 1⟩)
        ...
        (⟨変数 n⟩ ⟨式 n⟩))
  ⟨本体⟩)
  - 局所的な変数定義：⟨本体⟩でのみ有効

組み込みデータ型

- 整数
- 文字列
- 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）

組み込みデータ型

- 整数
- 文字列
- 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 "Hello" #\W)
(1 "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〉)：先頭以外からなるリストを返す

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

- cadr, cddr, caddr, ...：便利な記法

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 (map display l) (newline))
Hello World!
#t
```

上の関数（前述の list も）は，自分で定義する事も可能（この講義で学びます）

閑話休題 その 1

- ファイルからのプログラムの読込み
- 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（デモ）

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
  - 通常の関数呼出しと異なり，‹式›を評価せず，そのまま返す
- '‹ 式 › = (quote ‹ 式 ›)

Hello World その 3

```
> (define (HelloWorld-7)
    (map display (list "Hello" " " "World!"))
    (newline) )
> (define (HelloWorld-8)
    (map display '("Hello" " " "World!"))
    (newline))
> (define (HelloWorld-9)
    (map display
      '("H" "e" "l" "l" "o" " " "W" "o" "rld!"))
    (newline))
> (define (HelloWorld-10)
    (map display
      '(#\H #\e #\l #\l #\o
        #\  #\W #\o #\r #\l #\d #\!))
    (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!
```

- 逆にプログラムをデータとして見るとデータになる

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

→ メタプログラミング（プログラムを操作するプログラム）

シンボル（記号）：つづき

- 備考：

```
> (map display '("Hello" " " "World!"))
```

- map，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 map ⟨手続きオブジェクト⟩)
  (define display ⟨手続オブジェクト⟩)
  → 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"))
      (map display
        (append
          '(He llo)
          (list space wor)
          '(ld!)))
      (newline)))
ok
> (HelloWorld-11)
Hello World!
#t
```

再帰的な手続き定義

- 「中身を知らないmap 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
- 再帰的な定義の書き方についても，奥乃先生の第 1 回講義のとおり
  - 復習: factorial

```
> (define (fact n)
    (if (<= n 0)
        1
        (* n (fact (- n 1)))))
> (fact 16)
20922789888000
```


Hello World その 5

- リストを反復的に辿りながら要素を順に表示

```
> (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
```


閑話休題 その 2

- JAKLD のバックトレースで末尾再帰最適化を確認
- 非末尾再帰版 factorial

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

>(fact 8)
RuntimeException: undefined function foo
Backtrace: if < fact < if < fact < if < fact < if < fact < if <
fact < if < fact < if < fact < if < fact < if < fact < top-level

>
```

閑話休題 その 2

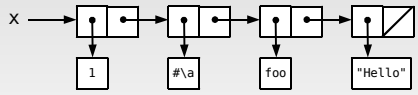
- 末尾再帰版 factorial

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

> (fact-iter 1 1 8)
RuntimeException: undefined function foo
Backtrace: if < fact-iter < top-level

>
```

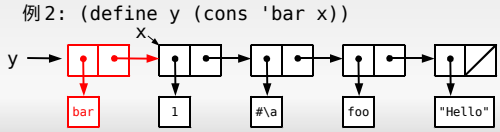

Cons セル

- 任意の 2 つのデータを結合
- リスト表現:

例 1: (define x '(1 #\a foo “Hello”))



例 2: (define y (cons 'bar x))



発展的な内容

(注意:
- もし時間に余裕があれば, ごく簡単に説明します
- 興味を持たれた方は, 後日, 読みなおしてみてください)

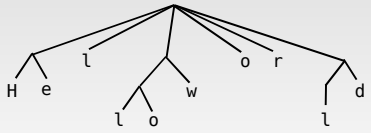
内容:

1. より一般的なデータ構造を表現する方法
   - 本講義の後半で勉強します
2. eval を使った（簡単な）メタプログラミング
   - 2 回生前期の湯浅先生の講義で勉強します

階層データ構造

- リストは別のリストを要素に持つことが可能
  → 線形リスト以外，たとえば木構造も表現可能

  例: '((H e) l ((l o) w) o r ((l) d))



- これを表示するには？（2.2.2 節）
- 慣れた Lisp プログラマは，上のようなシンボルのリスト構造をもっぱら使用（文字や文字列は本当に必要な時のみ）

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 (scheme-report-environment 5))
25
> (define prog2 (list '-
                      (cadr prog1)
                      (caddr prog1)))
ok
> prog2
(- (* 7 3) 4)
> (eval prog2 (scheme-report-environment 5))
17
```


最後に（個人的なコメント）

- Scheme プログラミングにハマってください！
  - 面白い機能が盛りだくさん
    - 強力なマクロ機能，メタプログラミング ：
      Lisp の Lisp たる所以
      <コード> = <データ> = <リスト>
    - 第一級継続 ：Scheme の特色の一つ
  - 後々，必ず役に立ちます
    - 一生，Lisp だけでプログラムを書き続ける Lisp ハッカーになる
      → 尊敬されます :-)
    - そうでなくても，Ruby，Python，C#，Java，Scala，..
      すべて Lisp の遺伝子を受け継いでいます
      → あたらしい言語の習得が非常に楽
      （はじめに学んだ言語が C だった馬谷の私見）

補足事項

- この資料は，奥乃先生の講義ホームページに掲載してもらう予定です
- 質問やコメント等ありましたら：

  umatani@kuis.kyoto-u.ac.jp

  まで