TUTScheme 入門

京都大学情報学研究科
通信情報システム専攻
湯淺研究室 D3　平石 拓
2007/10/9

講義の内容

- Lisp (Scheme)の基本事項の解説
  - SICPの1.1～1.1.6
  - 1.1.6までに載ってないSchemeの基本事項
    - 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)

Scheme

- Scheme
  - Lispの方言の一つ（cf. Common Lisp, Emacs Lisp）
  - リスト処理
  - 対話環境・・・デバッグしやすい
  - 小さい言語仕様（50ページ）+高い拡張性.
    - （2007/09に成立した新仕様で3倍以上に増えた）
    - C（C99）は538ページ
    - Common Lisp（第2版）は1029ページ
  - GC（ごみ集め）
  - 継続オブジェクト
  - . . .

TUTScheme

- Schemeの実装の1つ.
  - 湯淺先生，小宮先生（電気通信大，元湯淺研）開発
  - 利用手段
    - メディアセンターの端末
    - http://www.spa.is.uec.ac.jp/~komiya/download/ から入手
      - Windows（Cygwin版もあり）
      - Mac OS X
      - RedHat Linux
      - Fedora Core

起動・終了

```
$ tus ---------- 起動
TUTScheme version 1.4g
(C) Copyright Taiichi Yuasa …
SC> (+ 3 4)
7
SC> (bye) ---------- 終了
Bye.        処理系によっては(exit), (quit)
$
```

講義の内容

- Lisp (Scheme)の基本事項の解説
  - SICPの1.1～1.1.6
  - 1.1.6までに載ってないSchemeの基本事項
    - 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)

“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

“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)

1.1 言語の要素

- **primitives(基本式, 原始式)**
- **means of combinations(合成法)**
- **means of abstraction(抽象化法)**

- **Creating procedure objects (手続きの作成法)**
- **Viewing the rules of evaluation from a computational perspective (計算という観点からの評価規則)**

Scheme (Lisp) の基本

- **式(expression)は単純なものから構築.**
- **ほぼすべての式は, 値(value)を持つ.**
- **式は評価(evaluate)されて値を返す.**
- **すべての値には型(type)がある.**

このやり方は日常生活では普通にやっている.
ものは分類されて, 使われる.
e.g., 用途別の道具.

基本式(primitives)

- **Self-evaluating primitives(評価すると自分自身の値)**
  - **Numbers (数): 38, 3.80, 1.4141, 2.3e-4, 3/5**
  - **Strings (文字列): “moji”, “a”,**
  - **Booleans (論理式): #t, #f**
- **Built-in procedure (組込み手続き)**
  - **基本オブジェクト(primitive objects)の処理**
  - **Numbers(数): +, -, *, /, <, =, <=, ...**
  - **Strings(文字列): string-length, string=?**
  - **Booleans(論理式): and, or, not, xor, nand,**
- **Names for built-in procedures**
  - **+ ⇒ ＋という組込み手続き**

合成法(combinations)

- **Primitives を使って式を合成**
- **(+ 3 5)**
  (演算子 引数 . . .)

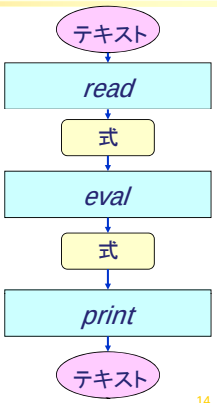
- **評価法**
  - **部分式(subexpressions)の評価し値を得る.**
  - **演算子(operator)を引数(arguments)に適用.**
  - **procedure application (手続き適用)という**

名前と値との関連付け

- `define` を使って式を合成
- `(define foo (+ 3 5))`
  `foo` の値は 8
- `(define bar +)`
  `bar` は + と同じ手続き
- `(bar 3 5)`

1.1.1 Expressions(式)

- **An interpreter （解釈系）: *read-evaluate-print ループ（REPL）を繰り返す***
- **read: テキスト表現の式を内部表現に変換**
- **evaluate:式を評価**
- **print: 評価結果の式を内部表現からテキスト表現に変換，出力**



1.1.2 Naming (名前) Environment (環境)

- The critical aspect of a programming language is the means it provides for using a name to refer to computational object.
- The name identifies a **variable** whose **value** is the object.
- **What is a computational object?**
  - from simple data such as numbers
  - to Complex structures
- The **environment** provides some sort of memory that keeps track of the name-object pairs.

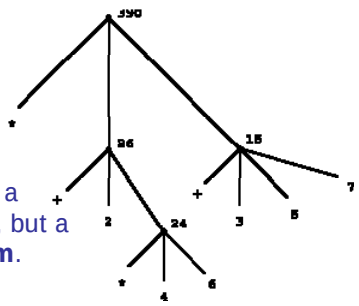
1.1.2 Naming and the Environment

- (define size 2)　処理系依存
- size
  2
- (* 5 size)
  10
- (define pi 3.14159)
- (define radius 10)
- (* pi (* radius radius))
  314.159
- (define circumference (* 2 pi radius))
- circumference
  62.8318

環境
size: 2
pi: 3.14159
radius: 10
circumference: 62.8318

1.1.3 Evaluating Combinations

- (* (+ 2 (* 4 6))
  (+ 3 5 7))



- define is not a combination, but a **special form**.

1.1.4 Compound Procedures(合成手続き)

- The rule of *procedure definitions*

1. Numbers and arithmetic operations are primitive data and procedures.
2. Nesting of combinations provides a means of combining operations.
3. Definitions that associate names with values provide a limited means of abstraction.

- Use define to associate a procedure with a name:
- “To square something, multiply it by itself.”

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*>)
  - <formal parameter> 仮パラメータ
  - <body> 本体

1.1.5 The Substitution Model for Procedure Application(手続き適用の置換モデル)

- Assume that the mechanism for applying primitive procedures to arguments is built into the interpreter.
- For compound procedures, the application process is as follows:

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

1.1.5 The Substitution Model for Procedure Application(手続き適用の置換モデル)

- (foo 5)

1. foo is the procedure defined before.
2. By retrieving the body of foo
   (sum-of-squares (+ a 1) (* a 2))
3. Replace the formal parameter a by the argument 5: (sum-of-squares (+ 5 1) (* 5 2))
4. Two arguments are evaluated and substituted for the formal parameters x and y:
   (+ (square 6) (square 10))
5. (+ 36 100)
6. 136

1.1.5 The Substitution Model for Procedure Application(手続き適用の置換モデル)

- A model that detemines the meaning of procedure application, insofar as the procedure in this chapter are concerned.
- The purpose of the substitution is just for explanation.
- The substitution model is the first step and we will present a sequence of increasingly elaborate models of how interpreters work.

1.1.5 The Substitution Model 注意

- **(|-∀) ∀右 と (∃|-) ∃左 は、eigenvariable condition が必要。要注意。**
- **a = b |- a = b**
  ---------- (|-∀)
  a = b × ∀x.(x = b)

  a = b |- a = b
  ----------(∃|-)
  ∃x.(x = b) × a = b

  上記は成立しない！

- **a = b |- a = b**
  ---------- (|-∃)
  a = b |- ∃x.(x = b)

  a = b |- a = b
  ----------(∀|-)
  ∀x.(x = b) |- a = b

  は成立！

Applicative order vs. normal order (作用的順序と正規順序)

- 1.1.3: ***applicative-order evaluation***
  “*Evaluate the arguments and then apply*” method called
- **Alternative evaluation model: *normal-order evaluation***
  “*fully expand and then reduce*”
- Exercise 1.5 is an instance of an illegitimate value where these two evaluation methods do not give the same result.

Alternative Model for Procedure Application

- (foo 5)

1. (sum-of-squares (+ 5 1) (* 5 2))
2. (+ (square (+ 5 1)) (square (* 5 2)))
3. (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
4. (+ (* 6 6) (* 10 10))
5. (+ 36 100)
6. 136

The same answer, but different process.
The evaluations of (+ 5 1) and (* 5 2) are each performed twice.

1.1.6 Condtional Expressions and Predicates (条件式と述語)

- 絶対値をcase analysis (場合分け)で定義

$$|x| = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -x & \text{if } x < 0 \end{cases}$$

- (define (abs x)
  (cond ((> x 0) x)
  ((= x 0) 0)
  ((< x 0) (- x))))
- General form of a conditional expression

1.1.6 Condtional 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.
- *<p>* predicate. Its value is interpreted as either **true** or **false**.
- *<e>* consequent expression
- Special <p>: **else**

1.1.6 Condtional Expressions (条件式)

- **(define (abs x)**
  (cond ((< x 0) (- x))
  (else x)))
- **(define (abs x)**
  (if (< x 0)
  (- x)
  x))
- **If** : special form
- **(if** <*predicate*> <*consequent*> <*alternative*>**)**

1.1.6 Predicates (述語)

- **(and** *<e_1> … <e_n>***)** **論理積(左から評価)**
- **(or** *<e_1> … <e_n>***)** **論理和(左から評価)**
- **(not** *<e>***)** **論理否定**

例:

- 5 < x < 10 ⇒
- **(define (>= x y)**
  (or (> x y) (= x y)))
- **(define (>= x y)**
  (not (< x y)))

1.1.6 Condtional Expressions (条件式)

- **(define (abs x)**
  (cond ((< x 0) (- x))
  (else x)))
- **(define (abs x)**
  (if (< x 0)
  (- x)
  x))
- **If** : special form
- **(if** <*predicate*> <*consequent*> <*alternative*>**)**

1.1.6 Predicates (述語)

- **(and $<e_1>$ … $<e_n>$) 論理積(左から評価)**
- **(or $<e_1>$ … $<e_n>$) 論理和(左から評価)**
- **(not $<e>$) 論理否定**

例:

- 5 < x < 10 ⇒
- **(define (>= x y)**
  (or (> x y) (= x y)))
- **(define (>= x y)**
  (not (< x y)))

階乗の計算

- n! を求める関数

```
SC> (define (fact n)
      (if (= n 0)
        1 ---------- n=0の時
        (* n (fact (- n 1))))) ---------- n=0でない時
fact
SC> (fact 10)
362800
```

リスト

- Lispにおける最も重要なデータ型の1つ
- データ(要素)の“並び”を表す
  - (1 2 3 4 5 6 7 8 9 10)
  - (we eat rice)
  - ((a b c) x y (1 2))

  など。

リストの生成(1)

```
SC> (list 1 2 3 4)
(1 2 3 4)
SC> (list 'w 'x (list 'y 'z))
(w x (y z))
SC> (define x 4)
x
SC> (list x (* x 5))
(4 20)
SC> (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。

```
SC> '(x y) ---------- (quote (x y)) の略記
(x y)
SC> '((x y) 1 2 (a b c))
((x y) 1 2 (a b c))
SC> '(define (square x) (* x x))
(define (square x) (* x x))
```

リストの要素の参照

```
SC> (car '(a b c d)) ---------- リストの先頭要素
a
SC> (cdr '(a b c d)) ---------- 先頭要素を除いたリスト
(b c d)
SC> (car (cdr (cdr '(a b c d))))
c
SC> (cdr (cdr (cdr (cdr '(a b c d)))))
()
```

リストへの要素追加

- リストの先頭に要素を追加

```
SC> (cons 'we '(eat rice))
(we eat rice)
SC> (cons 'never (cdr '(we eat rice)))
(never eat rice)
SC> (cons '(a b c) '(x y z))
((a b c) x y z)
SC> (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                          ---------- xが()か？
      (+ 1 (my-length (cdr x)))))
```

リストを結合する関数

- (append '(a b c) '(1 2 3)) → (a b c 1 2 3)
- 一般的な定義は？
  - (append () *y*) ＝
  - (append '(*a b* ...) *y*) ＝

リストを逆順に並べ換える関数

- (reverse '(a b c)) → (c b a)
- 一般的な定義は？
  - (reverse ()) ＝
  - (reverse (*a b* ...)) ＝

効率は悪い

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。

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

  - システムがこのリストを「評価」すると、
    "関数呼び出し式"として処理を行い、値を返す。

```
SC> (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。

入出力関数（1）

- 出力関数

```
SC> (write '(a b c))
(a b c)(a b c)
```

(a b c)：システムの出力（評価値）
(a b c)：write関数の出力

```
SC> (begin (write '(a b c)) (newline))
(a b c)
#t
```

(a b c)：write関数の出力
#t：システムの出力（評価値）

入出力関数（2）

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

入出力関数（3）

- 入力関数

```
SC> (read)
abcde ←（キーボードから入力）
abcde ---------- システムの出力（評価値）
SC> (begin (display "input: ") (fact (read)))
input: 11 ←（キーボードから入力）
          （プロンプト）
39916800
```

ファイル入出力（1）

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

ファイル入出力（2）

```
SC> (define in (open-input-file "infile"))
in
SC> (read in) ------- "infile"からデータを1つ読み込む
data
SC> (read in)
#<end-of-file> ------- ファイルの終端に達した場合
SC> (close-input-port in)
#t
```

ファイル入出力（3）

- (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。

ファイル入出力（4）

- 例：1^2, 2^2, ..., 99^2 の値をファイルに書き出す。

```
(call-with-output-file "square99"
  (lambda (out)
    (do ((n 1 (+ 1 n)))
        ((>= n 100))
      (write (* n n) out)
      (newline out))))
```

ファイル入出力(5)

- 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)
    (do ((dat (read in) (read in)))
        ((eof-object? dat))
      (write dat)
      (newline))))
```

プログラム・ファイル

- (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。

```
SC> (load ”square.scm”)
Loading square.scm...
Finished.
”square.scm”
SC> (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.

関数実行のトレース(使用例)

```
SC> (trace fact)
#t
SC> (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(数値)

- 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(リスト)

- (length 〈リスト〉)
- (append 〈リスト1〉 … 〈リストn〉)
- (reverse 〈リスト〉)

主な組み込み関数（等号・論理演算）

- 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

- ヘルプ機能（tus, guileなど）
  - (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.

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

参考資料

- 講義のページ
  http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/07/IntroAlgDs/
- TUT Schemeのマニュアル
  http://www.spa.is.uec.ac.jp/~komiya/tus-man/tus/
- TUT Scheme Tips
  http://www.spa.is.uec.ac.jp/~komiya/download/tus-tips.html

宿題

- Factorialのrecursive版とiterative版のプログラムを実際に作成して実行してみる.
- trace関数による実行ログをプリントアウトして提出
- 締切：10/16（火）正午
- 提出場所：10号館の提出BOX

「recursive版とiterative版」
→第一回目の講義資料