TUTScheme の利用

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 M2　平石 拓

TUTScheme

- Scheme
  - Lispの方言の一つ（cf. Common Lisp, Emacs Lisp）
  - リスト処理
  - 対話環境・・・デバッグがしやすい
- TUTScheme
  - Schemeの実装の1つ（湯淺先生開発）。
  - 現在、京大メディアセンターの端末（Red Hat Linux）で利用可能。

起動・終了

```
$ tus ---------- 起動
TUTScheme version 1.4g
(C) Copyright Taiichi Yuasa …
SC> (+ 3 4)
7
SC> (bye) ---------- 終了
Bye.
$
```

対話環境の利用

```
SC> (+ 3 4)
7 ---------- (+ 3 4)の評価値
SC> (* (+ 1 2) (- 10 7))
9 ---------- (* (+ 1 2) (- 10 7))の評価値
SC> 1234
1234 ---------- 1234の評価値
SC> "abc"
"abc" ---------- "abc"の評価値
```

変数定義 (1)

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

変数定義 (2) : 関数

```
SC> (define (square x) (* x x))
square
SC> (square 10)
100
SC> (square (* 3 4))
144
SC>
```

変数定義 (3) : 関数

- 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
```

記号 (1)

- (quote 〈記号〉) ・・・スペシャル・フォーム

```
SC> (quote x)
x
SC> (quote cat)
cat
SC> (define x (quote y))
x
SC> x
y
```

記号 (2)

- ’⟨記号⟩ でも同じ意味

```
SC> ’x
x
SC> ’cat
cat
SC> (define x ’y)
x
SC> x
y
```

リスト (1)

- 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 (2)

```
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 (3)

- リストの要素がわかっている場合は、(quote ⟨記号⟩) でもよい。

```
SC> ’(x y) ---------- (quote (x y)) と同じ
(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))
```

リストを扱う関数(car, cdr)

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

リストを扱う関数 (cons)

- 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)
```

その他のリスト処理関数

- (length 〈リスト〉) : リストの要素数を返す

  SC> (length '((a b) c (d e f)))
  3
- (append 〈リスト1〉 ... 〈リストn〉) : リストを結合する

  SC> (append '(a b) '(c (d e)) '(f))
  (a b c (d e) f)
- (null? 〈データ〉) : 〈データ〉が空リストなら#t, そうでなければ#fを返す

  SC> (null? '())
  #t
  SC> (null? '(a b c))
  #f

リスト処理関数の定義例

```
SC> (define (my-length x)
        (if (null? x)
            0
            (+ 1 (my-length (cdr x)))))
my-length
SC> (my-length '(a b c d))
4
```

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。
- システムが評価できるデータを「フォーム」という。
- 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 (1)

- リスト・フォーム
  - 関数適用 ・・・ (〈関数式〉 〈式1〉 … 〈式n〉)
    1. 〈関数式〉 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
    - マクロを表す記号で始まるリスト (詳細は省略)

フォームの分類 (2)

- 記号
  - 変数の値が評価値になる

SC> (define x 30)

x

SC> x

30

SC> +

#<function +>

フォームの分類 (3)

- その他のデータ
  - 数値や文字列などは、それ自身が評価値となる

SC> 123

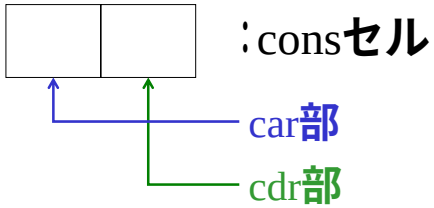
123

SC> "abcde"

"abcde"

consセル (1)

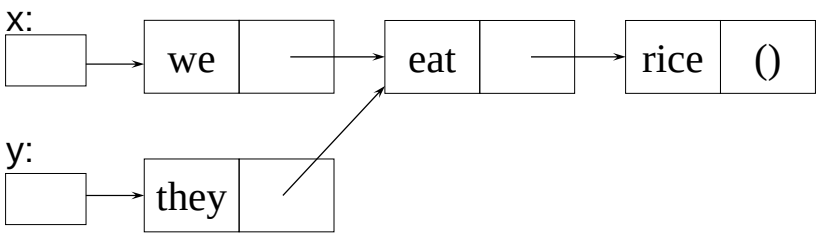
- 例1: (we eat rice) の内部表現



consセル (2)

- 例2:

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



consセル (3)

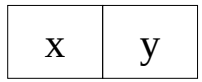
- (cons ⟨データ1⟩ ⟨データ2⟩): car部とcdr部がそれぞれ ⟨データ1⟩ , ⟨データ2⟩ であるコンス・データを返す。

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

consセル (4)

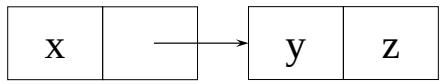
- ドット・ペア

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



- ドット・リスト

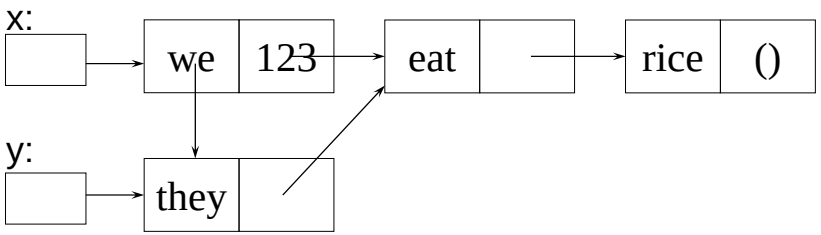
```
SC> (cons 'x (cons 'y 'z))
(x y . z)
```



consセル (5)

- **破壊的操作**

SC> (set-cdr! x 123)
(we . 123)
SC> (set-car! x y)
((they eat rice) . 123)



入出力関数 (1)

- **出力関数**

SC> (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
SC> (begin (write '(a b c)) (newline))
(a b c) write関数の出力
(a b c) システムの出力 (評価値)
SC>

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
SC>
```

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> (fact (read))
11 ← (キーボードから入力)
39916800
SC>

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.

- **〈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出力する。

```
(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

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
```

代表的な組み込み関数（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
- 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⟩)

代表的な組み込み関数（リスト）

- (nth ⟨インデックス⟩ ⟨リスト⟩)
- (nthcdr ⟨インデックス⟩ ⟨リスト⟩)
- (last ⟨リスト⟩)
- (length ⟨リスト⟩)
- (append ⟨リスト1⟩ … ⟨リストn⟩)
- (reverse ⟨リスト⟩)

代表的な組み込み関数（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】

代表的な組み込み関数（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⟩)
- (function? ⟨データ⟩)
- (string? ⟨データ⟩)

その他の組み込み関数

- システムとの対話の記録
  - (transcript-on ⟨ファイル名⟩):システムとの対話をファイル⟨ログファイル⟩に記録する
  - (transcript-off):ログファイルへの記録をやめる
- ヘルプ機能
  - (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

⟨参考⟩「TUTSchemeのマニュアル」

今回扱わなかった関数等の説明。

http://www.yuasa.kuis.kyoto-u.ac.jp/~komiya/tus-man/tus/

http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/04/IntroAlgDs/