情報実験IIのためのScheme入門

鈴木 徹也

2017年10月1日改訂

目 次

1 はじめに

本資料の目的は, 情報実験 II で実装する Lisp 言語のイメージを持ってもらうことである. 実験で実装する Lisp の仕様は Lisp の方言の一つである Scheme を参考にしている. そこで Scheme の処理系 Gauche[1, 2] を使いながら Scheme の基本を説明する.

本資料では Linux に Gauche がインストールされていると仮定して説明する. Scheme 処理系 Gauche のインストール方法は文献 [1, 2] に任せる. ただし, 情報実験 II 向けに用意した Linux 環境には Gauche がインストールされているので, それを利用すれば Gauche を自分でインストールする必要はない.

2 Gaucheの実行方法

2.1 対話的操作

シェルのコマンドラインにGaucheのコマンド名`gosh`を打つとGaucheが起動し, プロンプトが表示される. 以降, Gaucheは入力された式を評価[1]し, その評価値を表示することを繰り返す.

```
% gosh
gosh>
```

ここで入力する式は**S式** (S-expression) と呼ばれる. 最も簡単なS式として数値, 文字列, 真理値 (`#t`が真, `#f`が偽) を入力してみよう. 数値の評価結果は数値, 文字列の評価結果は文字列, 真理値の評価結果は真理値である.

```
gosh> 1
1
gosh> 2
2
gosh> "Hello, world"
"Hello, world"
gosh> #t
#t
gosh> #f
#f
```

Lispでは式は前置記法で記述する. 例えば中置記法の式$1+2$は, 前置記法では$+\ 1\ 2$となる. ただしLispでは演算子や数値の列を括弧でくくって`(+ 1 2)`と記述する.

```
gosh> (+ 1 2)
3
```

より複雑な式$(1+2)*(5-2)$は, 括弧を入れ子にして`(* (+ 1 2) (- 5 2))`と記述する.

```
gosh> (* (+ 1 2) (- 5 2))
9
```

`write`という手続きでデータを表示できる.

```
gosh> (write "Hello, world")
Hello, world#<undef>
```

上の例では文字列"`Hello, world`" が表示された. `#<undef>`は, 手続き`write`が未定義値

[1] ここでいう "評価" とは, 式の値を計算すること.

を返したことを意味している. 手続き write は改行しないので, 出力した文字列の直後に返り値が表示されている.

対話的な操作を終えるには手続き exit を使う.

```
gosh> (exit)
%
```

2.2 バッチ処理

あらかじめファイルにまとめて記述しておいたプログラム (式) を実行することもできる. 例えば次の内容をもつファイルをファイル名 sample.scm で保存したとする。セミコロンから行末まではコメントである.

ファイル sample.scm

```
;; 1 + 2
;; Hello, world
(+ 1 2)
(write "Hello, world")
```

gosh のコマンドライン引数にファイル名を指定すると, 指定したファイルに記述された式が順に評価される.

```
% gosh sample.scm
Hello, world
```

- 式 (+ 1 2) は評価されるが, その評価値は表示されない.
- 式 (write "Hello, world") は評価されると, 手続き write によって文字列が表示されるが, その評価値 (未定義値) は表示されない.

2.3 まとめ

- Gauche の実行方法には対話的な方法とバッチ処理とがある.
  - 対話的な方法では処理系は, 式を読む, 式を評価する, 式の評価値を表示する, を繰り返す.
  - バッチ処理では処理系は, コマンドラインに指定されたファイルを読み, そこに記述された式を順に評価する.
- Lisp が扱う式を S 式という.
- 数値, 文字列, 真理値は最も基本的な式である.
- 計算式は前置記法で記述する.

3　記号と変数

記号 (symbol) もまた基本的なS式である. 文字列と違い, 同じ綴りの記号は区別されない. 記号はデータにつける名前, つまり変数として利用される.

3.1　変数の考え方

C言語とLispとで変数の考え方が異なる.

- C言語では変数とは値を記録する容器 (メモリ領域) として考える. したがって, 変数に値を設定することを代入 (substitution) という.
- Lispでは変数とはデータに関連づけられた記号と考える. そこで変数を値に関連づけることを**束縛** (binding) という.

3.2　変数定義と記号の評価

記号をデータに束縛するには `define` を用いる. 次の例では記号 `x` を数値1に束縛する.

変数定義

```
gosh> (define x 1)
x
```

記号の評価値は, 記号が束縛されているデータである. 数値や文字列の評価値がそのデータそのものであったのとは異なる.

記号の評価

```
gosh> (define x 1)
x
gosh> x
1
```

S式そのものを評価値とするには `quote`(引用) を使う. (`quote` *exp*) の評価値は式 *exp* になる. これは'*exp* と略記できる. 下の例では記号 `x` に `quote` を適用している.

記号の評価

```
gosh> (define x 1)
x
gosh> (quote x)
x
gosh> 'x
x
```

手続きの引数や局所変数も考慮した上での記号の評価については, 第7節で詳しく説明する.

3.3 まとめ

- Lisp では変数は記号であり, 変数をデータに束縛すると考える.
- 変数定義には `define` を用いる.
- S 式の引用には `quote` を用いる. `quote` には単一引用符を使った略記法がある.

4　算術演算

算術演算をするための手続きを紹介する.

4.1　手続き +

書式 (+ exp_1 exp_2 $\cdots$ exp_n)

意味 0 個以上の式を引数にとり, その和を計算する.

手続き +

```
gosh> (+)
0
gosh> (+ 1)
1
gosh> (+ 1 2)
3
gosh> (+ 1 2 3)
6
```

4.2　手続き -

書式 (- exp_1 exp_2 $\cdots$ exp_n)

意味
- 引数が 1 個のときは, その引数の符号を反転した値
- 引数が 2 個以上のときは, exp_1 - exp_2 $\cdots$ - exp_n

手続き -

```
gosh> (- 1)
-1
gosh> (- 1 2)
-1
gosh> (- 1 2 3)
-4
```

4.3　手続き *

書式 (* exp_1 exp_2 $\cdots$ exp_n)

意味 0 個以上の式を引数にとり, その積を計算する.

手続き *

```
gosh> (*)
1
gosh> (* 1)
1
gosh> (* 1 2)
2
gosh> (* 1 2 3)
6
```

4.4 手続き /

書式 (/ exp_1 exp_2 $\cdots$ exp_n)

意味
- 引数が 1 個のときは, その引数の逆数
- 引数が 2 個以上のときは, exp_1 / exp_2 $\cdots$ / exp_n

手続き /

```
gosh> (/ 6)
1/6
gosh> (/ 6 3)
2
gosh> (/ 6 3 2)
1
```

1/6 は有理数 $\frac{1}{6}$ である.

4.5 手続き modulo

書式 (modulo exp_1 exp_2)

意味 剰余を求める.

手続き modulo

```
gosh> (modulo 9 5)
4
gosh> (modulo 9 3)
0
```

4.6 まとめ

- 算術演算を行う手続き+, -, *, /, modulo を紹介した.
- これらの演算を組み合わせるには, 第 2 節で示したように, 括弧でまとめた部分式を別の演算の引数にする.

5　ドット対とリスト

数値, 文字列, 真理値, 記号は Lisp では基本的な S 式である. 本節では S 式の組 (ドット対) を作る方法を説明する. この方法によってデータの列, 木構造, 構造体などを表現できる.

5.1　ドット対

Lisp では**ドット対**によって二つの S 式の組を構築できる. ドット対は S 式へのポインタを二つ持っていると考えるとよい. ドット対は **cons セル**とも呼ばれる. 構築されたドット対もまた S 式である.

ドット対は手続き `cons` によって構築できる. 次の例では二つの数値 1 と 2 の対 `(1 . 2)` を構築している (図 1).

ドット対の構築

```
gosh> (cons 1 2)
(1 . 2)
```

数値 `1` とドット対 `(2. 3)` との対は `(1 . (2 . 3))` と記述できる (図 2).

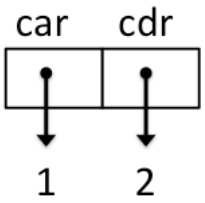

図 1: ドット対 (1 . 2)

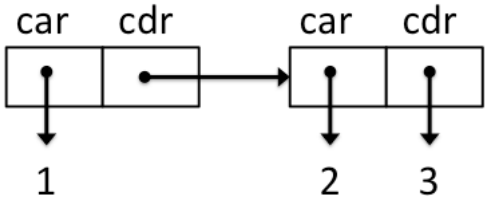

図 2: ドット対 (1 . (2 . 3))

ドット対の記述では, ドットと左括弧'.　(' とそれに対応する右括弧')' との組を省略できる. したがってドット対 `(1 . (2 . 3))` は, 下線部を省略して, `(1 2 . 3)` とも書いてよい. 本資料ではこの記法を**ドット対の省略記法**と呼ぶことにする.

次の例で用いている単一引用符 (') は引用 (`quote`) の略記法である.

ドット対の構築

```
gosh> '(1 . (2 . 3))
(1 2 . 3)
```

ドット対の前の部分を **car 部**, 後ろの部分を **cdr 部**という[2]. そしてドット対の car 部を取り出す手続きは `car`, cdr 部を取り出す手続きは `cdr` である.

[2]car はカー, cdr はクダーと読む.

car 部と cdr 部の取得

```
gosh> (car '(1 . 2))
1
gosh> (cdr '(1 . 2))
2
gosh> (car '((1 . 2) . 3))
(1 . 2)
gosh> (car (car '((1 . 2) . 3)))
1
gosh> (cdr (car '((1 . 2) . 3)))
2
gosh> (cdr '((1 . 2) . 3))
3
```

5.2 リスト

空リストも S 式の基本的なデータである.

空リスト

```
gosh> ()
()
```

この空リストと cdr 部がリストであるドット対とを**リスト**という. 例えば, S 式 (1 . ()) の cdr 部は空リストであり (図 3), 空リストは定義からリストである. したがって S 式 (1 . ()) はリストである.

リストの例

```
gosh> (cdr '(1 . ()))
()
```

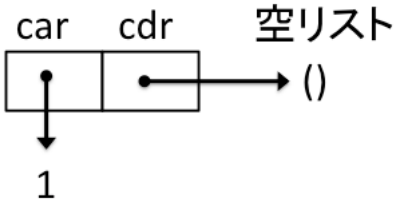

図 3: ドット対 (1 . ())

S 式 (cons 1 (cons 2 ())) を評価して得られるドット対を, ドットを使って記述すると (1 . (2 . ( ))) となる. ドット対の省略記法を使うと一重下線の組と二重下線の組とを削除できるので, この式は (1 2) と記述できる.

5.3 連想リスト

対を要素とするリストを**連想リスト**という. ここでは記号と値との対からなる連想リストを扱う.

連想リストの例

```
gosh> '((x . 10) (y . 10))
((x . 10) (y . 10))
```

連想リストの各要素をキーと値との組とみなして, キーを使って検索できる. 下の例ではキー x を持つ対を, 手続き assoc を使って連想リストから検索している.

連想リストからの検索

```
gosh> (define lst '((x . 10) (y . 20)))
lst
gosh> (assoc 'x lst)
(x . 10)
```

書式 (assoc *key list*)

意味 連想リスト *list* からキー *key* をもつ対を検索する. もし指定されたキーを持つ対があればそれを結果とする. もし見つからなければ#f を結果とする.

5.4 まとめ

- ドット対は car 部と cdr 部とからなり, S 式の組を実現できる.
- ドット対によってデータの列, 木構造, 構造体に相当するデータ構造を構築できる.
- 空リストと cdr 部をたどって空リストに到達するドット対とをリストという.
- リストの表記法にはドットを使わない方法もある.

6　リスト処理

手続き car, cdr, cons 以外のリストを扱う手続きを紹介する．紹介する手続きの中には真理値を返すものがある．そのような手続きを**述語**という．述語の名前は?で終わるのが慣例である．

6.1　述語 null?

書式　(null? *exp*)

意味
- 式 *exp* が空リストであれば真 (#t)
- そうでなければ偽 (#f)

述語 null?

```
gosh> (null? ())
#t
gosh> (null? '(1 2))
#f
gosh> (null? 'x)
#f
gosh> (null? 1)
#f
```

6.2　述語 list?

書式　(list? *exp*)

意味
- 式 *exp* がリストであれば真 (#t)
- そうでなければ偽 (#f)

述語 list?

```
gosh> (list? '(1 2 3))
#t
gosh> (list? '(1 2 . 3))
#f
gosh> (list? 'x)
#f
gosh> (list? 1)
#f
```

6.3 述語 pair?

書式 (pair? *exp*)

意味
- 式 *exp* がドット対であれば真 (#t)
- そうでなければ偽 (#f)

述語 pair?

```
gosh> (pair? '(1 2 3))
#t
gosh> (pair? '(1 . 2))
#t
gosh> (pair? 'x)
#f
gosh> (pair? 1)
#f
```

6.4 手続き append

書式 (append *list* ···)

意味 引数のリストを連結した新しいリストを返す.

手続き append

```
gosh> (append '(1) '(2 3))
(1 2 3)
```

6.5 手続き list

書式 (list *exp* ···)

意味 引数を並べたリストを返す.

手続き list

```
gosh> (list '(1) '(2 3))
((1) (2 3))
```

6.6 まとめ

- 真理値を返す手続きを述語という.
- 述語には?で終わる名前を付けるのが慣例である.
- cons 以外にもリストを作る手続きがある.

7　記号 (変数) の評価

局所変数や手続きの引数のことを考慮すると, 同じ記号であっても, それが記述された場所や手続きの呼び出し状況によって, それに関連づけられた値は異なる. そこで本節では変数に関連づけられた値の求め方を説明する.

7.1　スコープ

Scheme では, 変数のスコープとして**静的スコープ** (static scope)[3]を採用している. つまり参照できる変数はプログラムの記述上のブロックによって決まる. ちなみに Emacs Lisp は**動的スコープ** (dynamic scope) を採用している. 動的スコープでは, ある関数内ではその関数を呼んだ関数のスコープを参照できる.

7.2　フレーム

フレーム (frame) とは, 記号とそれに関連づけられているデータとの対応表である. 各フレームは, それに対応した別のフレーム (外側の環境) を持っている. ただし大域変数の値を記録してる**大域的なフレーム**は, 外側の環境を持っていない (図 4).

大域的なフレームはプログラムの実行開始時に存在する. そこにはあらかじめ記号 (例えば `cons`, `car`, `+`など) に対応する手続きが関連づけられている. 後に説明するように手続きもデータなので, 変数を手続きに束縛できる. 次の例では記号`+`の評価値として`#<subr +>`が表示されているが, それは和を計算する手続きを表している. 記号`+`を別のデータに束縛することもできる.

大域的なフレーム

```
gosh> +
#<subr +>
gosh> (define + 1)
+
gosh> +
1
```

大域的なフレーム以外のフレームは, 局所変数を作るときや手続きの仮引数に実引数を割り当てるときに構築される.

7.3　環境

環境 (environment) とはフレームの列である. 現在のフレーム (そのとき注目しているフレーム) から大域的なフレームまで外側の環境をたどるとフレームの列が得られる. こ

[3]構文スコープ　(lexical scope) ともいわれる.

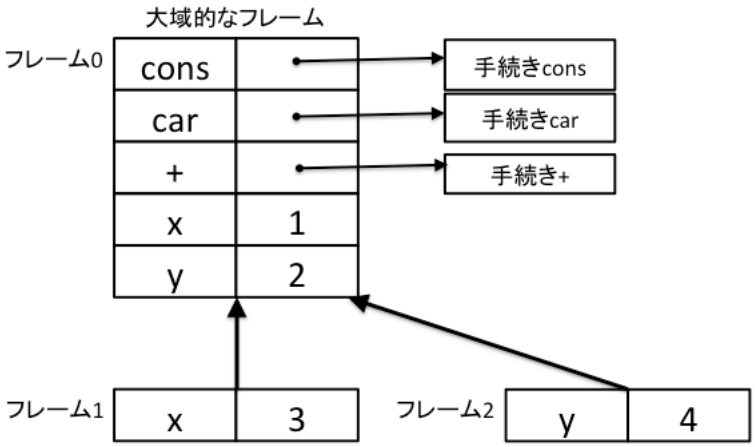

図 4: フレーム

のフレームの列によって変数の値が決まる. つまり, 記号に関連づけられた値は, 現在のフレームから大域的なフレームへ向かって探索してはじめに見つかった値とする.

プログラム実行開始時は, 大域的なフレームが現在のフレームとなる. その後, プログラム実行中に手続きを呼び出したり局所変数を作るたびに新たにフレームが作られ, それが現在のフレームとなる. 手続きが終了したり局所変数が有効な範囲から抜けたりすると, 現在のフレームが一つ前のフレームに戻る. 現在のフレームから大域的なフレームまでの列が, 現在の環境となる.

例えば図 4 のようなフレームがあったとする. 現在のフレームが, フレーム 1 の場合, 記号 x はフレーム 1 に記録されている 3 に束縛されていることになり, 記号 y はフレーム 0 に記録されている 2 に束縛されていることになる. もし現在のフレームがフレーム 2 なら, 記号 x はフレーム 0 に記録されている 1 に束縛されていることになり, 記号 y はフレーム 2 に記録されている 4 に束縛されていることになる.

7.4 まとめ

- Scheme は静的スコープを採用している.
- 変数の値は環境によって決まる.
- 環境とはフレームの列である.
- フレームは記号とデータとの対応表である.

8　リストの評価

リストがどのように評価されるのかを説明する.

8.1　手続きの適用

Lisp 処理系ではリスト (exp_0 exp_1 $\cdots$ exp_n) は次のように評価される.

1. exp_0 を評価した結果を $proc$ とする. もし $proc$ が手続きでなければエラーを出力して終了する.
2. もし $proc$ が組み込み手続きかクロージャ(第 12 節) であれば次を行う.
   (a) 部分式 exp_1, $\cdots$ exp_n をそれぞれ評価する.
   (b) 手続き $proc$ を, 引数 exp_1 の評価値, $\cdots$, exp_n の評価値に適用する.
   (c) 手続き $proc$ を引数に適用して得られた値をそのリストの評価値とする.
3. もし $proc$ が特殊形式 (第 9 節) であれば次を行う.
   (a) 手続き $proc$ を, 引数 exp_1, $\cdots$, exp_n に適用する.
   (b) 手続き $proc$ を引数に適用して得られた値をそのリストの評価値とする.

次の例では, 変数 x が数値 2 に束縛されているとき, x の 2 乗に 1 を加えた値を計算している.

リストの評価 1

```
gosh> x
2
gosh> (+ (* x x) 1)
5
```

この例においてリスト (+ (* x x) 1) は次のように評価される.

1. 記号+を評価して和を計算する組み込み手続きを得る.
2. リスト (* x x) を次の順序で評価し, 4 を得る.
   (a) 記号*を評価し積を計算する組み込み手続きを得る.
   (b) 記号 x を評価し 2 を得る.
   (c) 記号 x を評価し 2 を得る.
   (d) 積を計算する組み込み手続きを二つの引数 2 と 2 に適用し, $4(= 2 \times 2)$ を得る.
3. 数値 1 を評価し, 1 を得る.
4. 和を計算する組み込み手続きを二つの引数 4 と 1 に適用し, 5 を得る.

8.2 エラーとなる例

次の例ではリストの最初の式 1 の評価値が手続きではないのでエラーとなる.

リストの評価 2

```
gosh> (1 2 3)
*** ERROR: invalid application: (1 2 3)
Stack Trace:
_______________________________________
gosh>
```

8.3 まとめ

- リストの各部分式を評価した結果 ($<$ 手続き $>$ $<$ 引数 1 $<$ 引数 2 $>$ $\cdots$ $<$ 引数 n $>$) となる場合, その手続きを引数に適用する.
- リストの最初の部分式の評価値が手続きでないとエラーになる.

9 特殊形式

特殊形式と呼ばれる手続きについて説明する.

9.1 特殊形式の例

define によって変数定義できることはすでに説明した. 下の例では (define x 1) によって変数 x に数値 1 が関連づけられる.

define の評価

```
gosh> (define x 1)
x
gosh> x
1
```

第 8 節で説明したリストの評価手順において上記のリスト (define x 1) は次のように評価される.

1. 記号 define を評価し, 変数に値を関連づける手続きを得る.
2. 変数に値を関連づける手続きを, 二つの引数 (記号 x, 数値 1) に適用する.

第 8 節で示したリストの評価例 (リストの評価 1) の場合と異なるのは, 手続きを適用する前に, 引数が評価されないことである. もし引数の記号 x を評価してしまうと, 変数に値を関連づける手続きに記号 x を渡せなくなってしまう.

この define のように, 適用の前に引数が評価されない手続きを**特殊形式** (special form)[4] という. この節では三つの特殊形式 define, set!, quote を説明する.

9.2 特殊形式 define

書式 (define *symbol exp*)

意味 現在のフレームにおいて記号 *symbol* を *exp* の評価値に束縛する. *symbol* は評価されない.

9.3 特殊形式 set!

書式 (set! *symbol exp*)

意味 定義済みの変数 *symbol* の値を更新する. つまり, その時の環境[5]で見える変数 *symbol* の値を *exp* の評価値に更新する. *symbol* は評価されない.

[4] Scheme では構文 (syntax) という.
[5] 現在のフレームから大域的なフレームまでの列

9.4 特殊形式 quote

書式 (quote *exp*)

書式 (略記) '*exp*

意味 引数のS式 *exp* を評価せずそのまま返す.

9.5 特殊形式 begin

文法 (begin *exp* ...)

意味 左から順番に式 *exp* を評価する. 最後に評価した式の評価値を, この特殊形式の評価値とする.

第13節で説明する特殊形式 let とは違い, begin は新たなフレーム (第7節) を作らない.

9.6 まとめ

- 通常の手続きとは異なり, 全ての引数が評価されるわけではない特殊形式というものがある.

10　条件式

比較のための述語と否定の述語とを紹介する. 述語とは真か偽かのどちらかの値を返す関数である.

10.1　述語 =

書式 (= exp_1 exp_2 $\cdots$)

意味 二つ以上の引数 (全て数値) をとる. 引数が全て等しい ($exp_1 = exp_2 = \cdots$) なら#t, そうでなければ#f.

10.2　述語 <

書式 (< exp_1 exp_2 $\cdots$)

意味 二つ以上の引数 (全て数値) をとる. 引数が単調増加 ($exp_1 < exp_2 < \cdots$) ならば#t, そうでなければ#f.

10.3　述語 <=

書式 (<= exp_1 exp_2 $\cdots$)

意味 二つ以上の引数 (全て数値) をとる. 引数が単調非減少 ($exp_1 \leq exp_2 \leq \cdots$) ならば#t, そうでなければ#f.

10.4　述語 >

書式 (> exp_1 exp_2 $\cdots$)

意味 二つ以上の引数 (全て数値) をとる. 引数が単調減少 ($exp_1 > exp_2 > \cdots$) ならば#t, そうでなければ#f.

10.5　述語 >=

書式 (>= exp_1 exp_2 $\cdots$)

意味 二つ以上の引数 (全て数値) をとる. 引数が単調非増加 ($exp_1 \geq exp_2 \geq \cdots$) ならば#t, そうでなければ#f.

10.6 述語 eq?

書式 (eq? obj_1 obj_2)

意味
- obj_1 と obj_2 とが主記憶の同じ場所に記録される同じ型のデータなら#t
- obj_1 と obj_2 とが両方とも等しい数値, 等しい真理値, 等しい記号, (), ならば#t

述語 eq?

```
gosh> (eq? 1 1)
#t
gosh> (eq? 1 2)
#f
gosh> (eq? 1.0 1)
#f
gosh> (eq? #t #t)
#t
gosh> (eq? #t #f)
#f
gosh> (eq? 'a 'a)
#t
gosh> (eq? 'a 'b)
#f
gosh> (eq? (list 'a) (list 'a))
#f
gosh> (let ((x (list 'a))) (eq? x x))
#t
```

10.7 述語 equal?

書式 (equal? obj_1 obj_2)

意味
- obj_1 と obj_2 とが主記憶上の違う場所に記録されていても, 両者の中身が等しければ#t.
- 数値, 真理値, 文字列, 記号は, 両者の型と値が等しければ#t
- リストであれば再帰的に中身を比較する.

10.8 手続き not

書式 (not *test*)

意味
- *test* の評価値が#f ならば#t
- それ以外ならば#f

not の例

```
gosh> (not #t)
#f
gosh> (not #f)
#t
gosh> (not (= 1 2))
#t
gosh> (not (= 1 1))
#f
```

10.9 まとめ

- 比較のための述語と否定の述語を紹介した.
- 等価性を比較する述語には種類がある.

11 条件分岐

条件分岐の特殊形式を紹介する.

11.1 特殊形式 `if`

書式 (`if` *predicate then else*)

意味
- 式 *predicate* の評価値が偽 (`#f`) 以外ならば, 式 *then* を評価する.
- 式 *predicate* の評価値が偽 (`#f`) ならば, 式 *else* を評価する.

書式 (`if` *predicate then*)

意味
- 式 *predicate* の評価値が偽 (`#f`) 以外ならば, 式 *then* を評価する.
- 式 *predicate* の評価値が偽 (`#f`) ならば, 未定義の値を返す.

`if` の例

```
gosh> (define x 1)
x
gosh> (if (= x 1) "one" "not one")
"one"
gosh> (define x 2)
x
gosh> (if (= x 1) "one" "not one")
"not one"
gosh> (if (= x 1) "one")
#<undef>
```

11.2 特殊形式 `cond`

書式 (`cond` *clause* ...)

- *clause* は (*test exp* ...) か (`else` *exp* ...) のいずれかの形を取る.
- 最後の *clause* のみ (`else` *exp* ...) になりうる.

意味 次のように評価する.

- 左から順番に *clause* の式 *test* を評価する.
- はじめて偽 (`#f`) でない値となる *test* があったら, その *clause* の *exp* を順に評価する. 最後に評価した *exp* の値を, この特殊形式の評価値とする.

- もし全ての *test* の評価値が偽 (`#f`) となり, 最後の *clause* が (`else` *exp* ...) でなければ, 未定義の値をこの特殊形式の評価値とする.
- もし全ての *test* の評価値が偽 (`#f`) となり, 最後の *clause* が (`else` *exp* ...) であれば, その *clause* の *exp* を順に評価する. 最後に評価した *exp* の値を, この特殊形式の評価値とする.

特殊形式 `if` の入れ子が深くなる場合は, 特殊形式 `cond` を使った方が読み書きしやすい.

11.3 特殊形式 `and`

書式 (`and` *test* ···)

意味 *test* が順に評価される.

- 評価値が偽になる式があったら, 残りの式は評価されず, 偽の値が返る.
- すべての式が評価されたら最後の式の評価値が返る.
- 式が与えられなければ`#t` が返る.

`and` の例

```
gosh> (and (= 1 1) (> 2 1))
#t
gosh> (and (= 1 1) (< 2 1))
#f
gosh> (and)
#t
```

11.4 特殊形式 `or`

書式 (`or` *test* ···)

意味 *test* が順に評価される.

- 評価値が真になる式があったら, 残りの式は評価されず, 真 (`#t`) の値が返る.
- すべての式が評価されたら最後の式の評価値が返る.
- 式が与えられなければ偽 (`#f`) が返る.

and の例

```
gosh> (or (> 2 1) (= 1 1))
#t
gosh> (or (< 2 1) (= 2 1))
#f
gosh> (or)
#f
```

11.5 まとめ

- 条件分岐の特殊形式を紹介した.
- 最も基本的な条件分岐 if があれば, その他の特殊形式 cond, and, or は記述できる.

12　手続きの定義

Lisp では手続きもデータである. ここでは手続きの定義方法を説明する.

12.1　特殊形式 lambda

手続きを作成するには特殊形式 lambda を使う.

手続きの作成 1

```
gosh> (lambda (x y) (+ x y))
#<closure #f>
```

特殊形式 (lambda (x y) (+ x y)) を評価すると, 二つの仮引数 x と y の和を計算する手続きが得られる.

手続きの作成 2

```
gosh> ((lambda (x y) (+ x y)) 1 2)
3
```

この場合, 第 8 節で説明したリストの評価方法にしたがい, リストの先頭要素が評価される. その結果 lambda で作成した手続きが得られる. 次に二つの引数を評価し 1 と 2 を得る. 最後に lambda で作成した手続きを二つの引数 1 と 2 に適用し, それらの和 3 を得る.

lambda によって作られる手続きはその内部に次の 3 つの要素を持つ.

- 手続きを作成したときの現在のフレームへのポインタ
- 仮引数
- 手続き本体

このように環境と関連づけられた手続きを**クロージャ**(closure) という. 上の例では大域的なフレームへのポインタを保持する手続きが作成される (図 5).

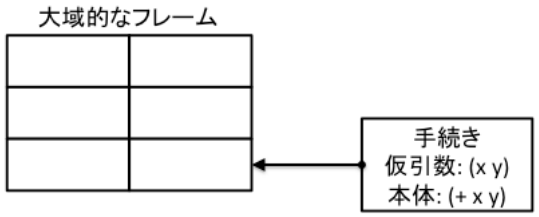

図 5: クロージャのイメージ

特殊形式 lambda の構文を詳しく説明する.

書式 (lambda *formals body* ···)

意味 仮引数 *formals* のあとに 1 個以上の式が並ぶ. 仮引数 *formals* の書き方によって, 手続きが取る引数の個数が変わる.

0 個の引数 (lambda () *body* ···)

$n(\geq 1)$ **個の引数** (lambda ($x_1 \cdots x_n$) *body* ···)
仮引数 x_i $(1 \leq i \leq n)$ が i 番目の実引数に関連づけられる.

仮引数と実引数との対応 ($n(\geq 1)$ 個の引数)

```
gosh> ((lambda (x1 x2) x1) 1 2)
1
gosh> ((lambda (x1 x2) x2) 1 2)
2
```

0 個以上の引数 (lambda x *body* ···)
仮引数 x が実引数リストに関連づけられる.

仮引数と実引数との対応 (0 個以上の引数)

```
gosh> ((lambda x x) 1 2 3)
(1 2 3)
gosh> ((lambda x x))
()
```

$n(\geq 1)$ **個以上の引数** (lambda ($x_1 \cdots x_n$. z) *body* ···)
仮引数 x_i $(1 \leq i \leq n)$ が第 i 番目の実引数に関連づけられる. 仮引数 z が第 $n+1$ 番目以降の実引数リストに関連づけられる.

仮引数と実引数との対応 ($n(\geq 1)$ 個以上の引数)

```
gosh> ((lambda (x1 x2 . z) x1) 1 2 3 4)
1
gosh> ((lambda (x1 x2 . z) x2) 1 2 3 4)
2
gosh> ((lambda (x1 x2 . z) z) 1 2 3 4)
(3 4)
gosh> ((lambda (x1 x2 . z) z) 1 2)
()
```

12.2 クロージャの適用

手続きの適用によって実引数を与えられたクロージャはその本体を次のように評価する.

1. 仮引数と実引数との対応を記録するフレームを作る. 手続きが保存しているフレームを, そのフレームの外側のフレームとする[6].
2. 作成したフレームを現在のフレームとして, 手続きの *body* が順に評価される.
3. 最後に評価した *body* の値が手続きの評価値となる.

手続きの適用 (`((lambda (x y) (+ x y)) 1 2)`) では図 6 のようにフレームが作られる.

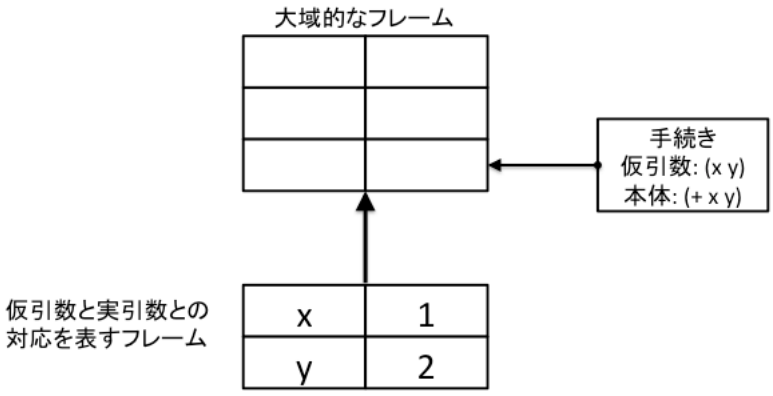

図 6: 手続きの適用 (`((lambda (x y) (+ x y)) 1 2)`) でのフレームの様子

[6]静的スコープを実現する.

仮引数と実引数との対応を記録するフレームは次のように作成できる. ここで手続きは (lambda *formals body* ···) という形式で作成され, 実引数のリストを *args* とする. 例えば手続きの適用 ((lambda (x y) (+ x y)) 1 2) においては, *formals* は (x y), *args* は (1 2) である.

1. 新しい空のフレーム *frame* を作る. 手続きが保持しているフレームを *frame* の外側の環境とする.
2. 次を繰り返す.

   ***formals* が空リスト () である場合** 実引数のリスト *args* も空リスト () ならば, 繰り返し処理を終了する. 実引数のリスト *args* が空リスト () でなければ, 仮引数より実引数が多いので, エラーを出力して終了する.

   ***formals* が記号 *x* である場合** *frame* において *x* を *args* に束縛して, 繰り返し処理を終了する.

   ***formals* がドット対である場合** *frame* において (car *formals*) で得られる記号を (car *args*) に束縛する. ただし *args* が空リストならば, 仮引数より実引数が少ないので, エラーを出力して終了する. *formals* を (cdr *formals*), *args* を (cdr *args*) とする.
3. *frame* を返す.

12.3 手続きへの名前の付け方

手続きはデータであるので 特殊形式 define を使って変数を手続きに束縛できる.

手続きへの名前の付け方 1

```
gosh> (define add (lambda (x y) (+ x y)))
add
gosh> (add 1 2)
3
```

上記の式 (define add (lambda (x y) (+ x y))) は, 式 (define (add x y) (+ x y)) と書いてもよい.

手続きへの名前の付け方 2

```
gosh> (define (add x y) (+ x y))
add
gosh> (add 1 2)
3
```

これは次のように二段階に分けて考えるとよい.

1. 式 (define *func* (lambda *formals body* ···)) を式 (define (*func* . *formals*) *body* ···)) と書き換える.
2. 第 5 節で説明したドット対の略記法を (*func* . *formals*) に適用する. つまり, ドットと左括弧'. (' とそれに対応する右括弧')' とを省略する.

この二段階を式 (define add (lambda (x y) (+ x y))) に適用してみよう.

1. 式 (define add (lambda (x y) (+ x y))) を式 (define (add . (x y)) (+ x y)) と書き換える.
2. (add . (x y)) を (add x y) に書き換えて (下線部を省略して), 式 (define (add x y) (+ x y)) を得る.

以上をまとめると, 取る引数の個数によって, 次のように手続きへの名前の付け方が変わる.

0 個の引数 (define *func* (lambda () *body* ···))
もしくは (define (*func*) *body* ···)

n **個の引数** (define *func* (lambda ($x_1 \cdots x_n$) *body* ···))
もしくは (define (*func* $x_1 \cdots x_n$) *body* ···)

0 個以上の引数 (define *func* (lambda *x body* ···))
もしくは (define (*func* . *x*) *body* ···)

n **個以上の引数** (define *func* (lambda ($x_1 \cdots x_n$. *z*) *body* ···))
もしくは (define (*func* $x_1 \cdots x_n$. *z*) *body* ···)

12.4 まとめ

- 手続きもデータである.
- 特殊形式 `lambda` で手続きを作成できる. 手続き型のデータは, それが定義されたときの現在のフレームへのポインタと手続きとを持っている.
- 引数は決まった個数だけでなく, 可変個数の引数も受け取れる.
- 特殊形式 `define` によって手続きに名前をつけられる.
- 手続きに名前をつけるために, `define` の略記法がある.

13　局所変数

特殊形式 lambda の仮引数を使って局所変数を実現できる.

13.1　特殊形式 lambda の応用

lambda による局所変数

```
gosh> (define (f x) ((lambda (p q) (+ x p q)) 10 20))
f
gosh> (f 30)
60
```

上記の define では手続き f を定義している. 手続き f は一つの引数 x と二つの局所変数 p と q との和を計算する. その p と q は特殊形式 lambda の仮引数であり, それぞれ 10 と 20 に束縛される. したがって式 (f 30) は, ((lambda (p q) (+ 30 p q)) 10 20) を評価した値, つまり (+ 30 10 20) を評価した値 60 となる. このときのフレームと手続きとの様子を図 7 に示す.

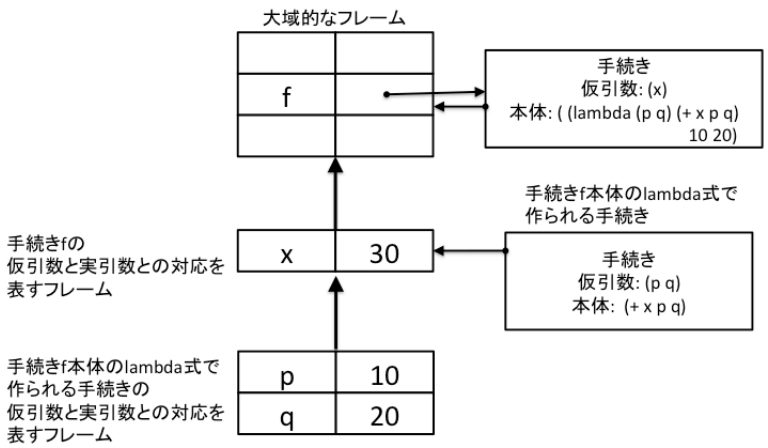

図 7: 手続き f の適用でのフレームと手続きの様子

つまり, 式 ((lambda (x_1 x_2 ... x_n) $body$ ...) v_1 v_2 ... v_n) を評価すると, 局所変数 x_i に値 v_i を関連づけた上で $body$ ... を順に評価することになる.

13.2 特殊形式 let

特殊形式 let を使うと, 式 ((lambda (*x*1 *x*2 ... *xn*) *body* ...) *v*1 *v*2 ... *vn*) を (let ((*x*1 *v*1) (*x*2 *v*2) ... (*xn* *vn*)) *body* ...) と記述できる.

let による局所変数

```
gosh> (define x 1)
x
gosh> (define (f x) (let ((p 10) (q 20)) (+ p q x)))
f
gosh> (f 30)
60
```

13.3 まとめ

- 特殊形式 lambda の仮引数を局所変数として利用できる.
- 局所変数を記述するための特殊形式 let がある. これは lambda を使った記述に変換できる.

14 高階関数

引数に関数 (手続き) をとったり関数を返したりする関数を**高階関数**という.

14.1 手続き apply

書式 (apply *proc arg* ··· *args*)

意味 *proc* は手続き, *args* はリストとする.

(*proc arg* ··· . *args*) の評価値を結果とする.

map の例

```
gosh> (apply + 1 2 '(3 4))
10
gosh> (apply + '(1 2 3 4))
10
```

14.2 手続き map

書式 (map *proc* $list_1$ $list_2$ ... $list_n$)

意味 *proc* は n 引数をとる手続き, 各 $list_i$ は長さ m のリストとする.

((*proc* $<$ $list_1$の第 1 要素 $>$ $<$ $list_2$の第 1 要素 $>$ ... $<$ $list_n$の第 1 要素 $>$)
(*proc* $<$ $list_1$の第 2 要素 $>$ $<$ $list_2$の第 2 要素 $>$ ... $<$ $list_n$の第 2 要素 $>$)
⋮
(*proc* $<$ $list_1$の第 m 要素 $>$ $<$ $list_2$の第 m 要素 $>$ ... $<$ $list_n$の第 m 要素 $>$)) を評価値とする.

map の例

```
gosh> (map + '(1 2 3))
(1 2 3)
gosh> (map + '(1 2 3) '(4 5 6))
(5 7 9)
gosh> (map (lambda (x y) (* x y)) '(1 2 3) '(4 5 6))
(4 10 18)
```

14.3 手続き eval

書式 (eval *expression env*)

意味 環境指定子 *env* の下で *expression* を評価する．*expression* の評価値を返す．

eval の例

```
gosh> (eval 'x (interaction-environment))
*** ERROR: unbound variable: x
Stack Trace:
---------------------------------------
gosh> (define x "hello")
x
gosh> (eval 'x (interaction-environment))
"hello"
gosh> (eval '(+ 1 2) (interaction-environment))
3
```

(interaction-environment) は Gauche の組み込み手続きとユーザ定義とを全て含んだ環境指定子を返す．

- 最初の eval では変数 x は未定義である．
- 2 番目の eval では，その直前に変数 x に関連づけた文字列が得られている．
- 3 番目の eval では，和を計算している．

14.4 まとめ

- 関数 (手続き) を引数として受け取ったり，関数を返り値とする関数を高階関数という．
- 関数を関数の引数とすることで，機能を組み合わせることができる．

15　応用 1 (クロージャ, 高階関数)

これまでに説明した内容の応用例を示す.

15.1　カウンタ

カウンタ counter.scm

```
(define (make-counter)
  (let ((count 0))
    (lambda () (set! count (+ count 1)) count)))

(define counter (make-counter))

(write (counter)) ; 1
(newline)
(write (counter)) ; 2
(newline)
(write (counter)) ; 3
(newline)
```

プログラムを簡単に説明する.

- 手続き make-counter は手続きを返す.
- その手続きは let で作成されるフレームを保持する.
- その手続きは let で作成されるフレーム内の変数 count をインクリメント (元に値に 1 を追加) する.

図 8 は二つ目の define までを実行した時の手続きと環境との関係を表している.

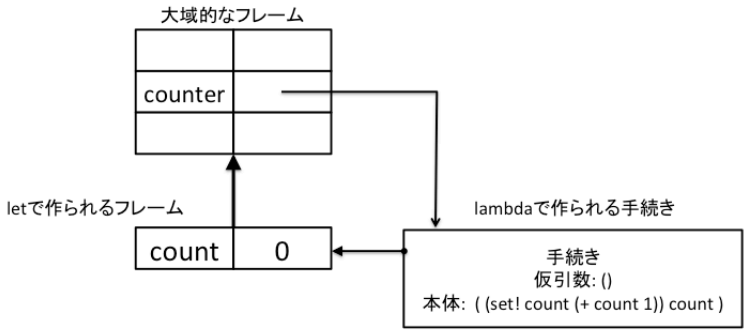

図 8: 手続きと環境との様子

したがってプログラムを実行すると次のようになる.

counter.scm の実行

```
% gosh counter.scm
1
2
3
```

二つ目の define で記号 counter が make-counter が返す手続きに束縛されているので, 式 (counter) を評価するたびにその返り値が 1 ずつ増加している.

15.2 クイックソート

クイックソート qsort.scm

```
(define (filter pred? lst)
  (if (null? lst) lst
      (if (pred? (car lst))
          (cons (car lst) (filter pred? (cdr lst)))
          (filter pred? (cdr lst)))))

(define (qsort pred? lst)
  (if (null? lst) lst
   (let ((pivot (car lst))
         (rest (cdr lst)))
     (append
      (qsort pred? (filter (lambda (x) (pred? x pivot)) rest))
      (list pivot)
      (qsort pred? (filter (lambda (x) (not (pred? x pivot))) rest))))))

;; 5未満を抽出する
;; (2 3 4 4 1)
(write (filter (lambda (x) (< x 5)) '(8 5 2 7 3 4 4 10 1)))
(newline)

;; 昇順で並べ替える
;; (1 2 3 4 4 5 7 8 10)
(write (qsort < '(8 5 2 7 3 4 4 10 1)))
(newline)

;; 降順で並べ替える
;; (10 8 7 5 4 4 3 2 1)
(write (qsort > '(8 5 2 7 3 4 4 10 1)))
(newline)
```

qsort.scmの実行

```
% gosh qsort.scm
(2 3 4 4 1)
(1 2 3 4 4 5 7 8 10)
(10 8 7 5 4 4 3 2 1)
```

15.3 FizzBuzz

FizzBuzz

```
(define (make-integer-list start end)
  (if (> start end) ()
      (if (= start end) (list start)
          (cons start (make-integer-list (+ start 1) end)))))

(define (fizzbuzz)
  (map (lambda (x)
         (if (= (modulo x 15) 0) 'FizzBuzz
             (if (= (modulo x 3) 0) 'Fizz
                 (if (= (modulo x 5) 0) 'Buzz x))))
       (make-integer-list 1 100)))

;; 1 から 100 までのリストを得る.
;; (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ... 100)
(write (make-integer-list 1 100))
(newline)

;; 1 から 100 までの整数について,
;; 3 の倍数は Fizz, 5 の倍数は Buzz, 15 の倍数は FizzBuzz に置き換える.
;; (1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz
;;  13 14 FizzBuzz 16 ... Buzzz)
(write (fizzbuzz))
(newline)
```

fizzbuzz.scm の実行

```
% gosh fizzbuzz.scm
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51  52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69
 70 71 72 73 74 75  76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91
 92 93 94 95 96 97 98 99 100)
(1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17
 Fizz 19 Buzz Fizz 22 23 Fizz Buzz 26 Fizz 28 29 FizzBuzz 31 32 Fizz
 34 Buzz Fizz 37 38 Fizz Buzz 41 Fizz 43 44 FizzBuzz 46 47 Fizz 49
 Buzz Fizz 52 53 Fizz Buzz 56 Fizz 58 59 FizzBuzz 61 62 Fizz 64 Buzz
 Fizz 67 68 Fizz Buzz 71 Fizz 73 74 FizzBuzz 76 77 Fizz 79 Buzz Fizz
 82 83 Fizz Buzz 86 Fizz 88 89 FizzBuzz 91 92 Fizz 94 Buzz Fizz 97
 98 Fizz Buzz)
```

16　応用2 (オブジェクト指向プログラミング)

オブジェクト指向プログラミングでのオブジェクトはデータ (状態) と手続きとのまとまりである．　本資料に書かれた内容の範囲でもそれを実現できる[7]．　局所変数に状態を持たせ，それらを操作する手続き群を連想リストによって束ねればよい．　本節ではオブジェクトの構築方法とメソッド呼び出しの方法とを説明する．

16.1　オブジェクトの構築

預け入れと引き出しだけができるシンプルな銀行口座を例にとって，本資料で説明した範囲内でのオブジェクト指向の実現方法を説明する．

簡単な口座オブジェクトのコンストラクタ

```
; 口座オブジェクトのコンストラクタ
; 連想リストとして，キーとメソッドの組を返す．
(define (make-account)
  (let ((amount 0)) ; 残高
    (list (cons 'get-amount (lambda () amount))
          (cons 'deposit    (lambda (x) (set! amount (+ amount x))))
          (cons 'withdraw   (lambda (x) (set! amount (- amount x)))))))
```

この手続き make-account は連想リストを返す関数である．　連想リストのキーがメソッド名，そのキーに対応する値がメソッドである．　どのメソッドも，特殊形式 let で構築されるフレーム上に記録される変数 amount を参照もしくは更新する．

- キー get-amount に対応する手続きは残高を返す．
- キー deposit に対応する手続きは預け入れをする．
- キー withdraw に対応する手続きは引き出しをする．

簡単な口座オブジェクトの構築

```
(define my-account (make-account))  ; 口座開設
```

手続き make-account をこのように使うと，変数 my-account を口座オブジェクトに束縛できる．

[7] ここで説明するオブジェクト指向は, プロトタイプベースのオブジェクト指向である. つまり Java 言語や C++言語のようなクラスベースのオブジェクト指向ではない.

16.2 メソッド呼び出し

オブジェクト指向プログラミングでは，各オブジェクトが互いにメッセージを送りあって (仕事の依頼をしあって)，全体の処理を進めると考える． 依頼された仕事を処理する手続きがメソッドである．

メソッド呼び出しの仕組み

```
; オブジェクトにメッセージを送る.
(define (send-message obj method . args)
  (apply (cdr (assoc method obj)) args))
```

この手続き send-message は，連想リストで実装されたオブジェクトへメッセージを送る手続きである．

- 引数はオブジェクト obj，メソッド名 method，メソッドへの引数 args である．
- (cdr (assoc method obj)) の評価値がメソッドになる．
- 高階関数 apply を使って，そのメソッドを引数 args に適用する．

手続き send-message は，具体的には次のように使用する．

メソッド呼び出しの例

```
(write (send-message my-account 'get-amount)) ; 残高照会 => 0
(newline)

(send-message my-account 'deposit 1000) ; 預け入れ
(write (send-message my-account 'get-amount)) ; 残高照会 => 1000
(newline)

(send-message my-account 'withdraw 400) ; 引き出し
(write (send-message my-account 'get-amount)) ; 残高照会 => 600
(newline)
```

- 最初の send-message では構築直後の口座の残高を参照する．
- 2 番目の send-message では口座に 1000 だけ預け入れする．
- 3 番目の send-message では口座から 400 だけ引き出す．

参考文献

[1] Shiro Kawai. ‘‘Gauche - A Scheme Implementation’’. http://practical-scheme.net/gauche/index-j.html (2014 年 8 月 18 日訪問).

[2] Kahua プロジェクト著, 川合 史朗 監修. プログラミング Gauche. オライリージャパン. 2008 年 03 月.