Lisp

2022 年 9 月 3 日，NISOC 会合

川俣 吉広

概要

- プログラミング言語 Lisp について紹介します
- 知ってる方には退屈な話かもですが、ご容赦下さい
- わかりやすさ優先のため、厳密な表現になっていない部分があります

Lisp : **Lis**t **P**rocessor

- 汎用プログラミング言語
  1959 ～ 現在

- 現在の主な方言
  - Common Lisp　(ANSI による標準化 (1994))
  - Scheme (IEEE による言語仕様、最新は R^7RS)
  - Clojure (Java VM で動作)
  - その他 (Emacs Lisp, Nyquist, AutoLisp...)

Lisp のデータ = S 式 (S-Expression)

S 式	アトム	数値，文字，文字列，配列，ハッシュ …	2, 2.0, 4/2, #C(2 0), #\a, “a string”, …
		シンボル	X, MAX-NUM, NIL, 2A+ …
	リスト	(S 式 S 式 …) リストの要素は 0 個以上の S 式	(A 2.0) (THIS IS (“version” 2.0)) (PRINT “Hello, world”) ()

S 式の定義の中に S 式が出てくる。循環的 (再帰的) な定義となっている。

Lisp プログラムの実行 = 評価

- Lisp のデータ (S 式) はすべて値を持っている
  その値を求めることを「評価する」と言う
- S 式の評価 = Lisp プログラムの実行
  - 2.0 → 2.0 ; 数値、文字列などは評価すると
  - “foobar” → “foobar” ; 値そのものが返る。
  - (+ (* 2 2) (* 3 3)) → 13 ; (関数名 引数 引数)→ 戻値
  - FOO → ??? ; アトムに束縛 (≒ 代入) されている
    ; 値を返す。

Lisp プログラムの実際の実行 (1/2)

- REPL : Read-Eval-Print Loop
  - S 式の入力・評価・表示を繰り返すしくみ
  - 最近の言語では Lisp 以外にも REPL を持っているものが多い
  - Lisp で REPL を実装すると

```
(LOOP (PRINT (EVAL (READ))))
```

- 【REPL の実演】
  - 電卓としても便利

Lisp プログラムの実際の実行 (2/2)

- REPL を直接使う … rlwrap で入力するとちょっと楽

- 開発環境を導入する
  - **定番** : Emacs + SLIME + Common Lisp 処理系
  - Vim + Lisp プラグイン : slimev
  - IDE の Lisp プラグイン : VSCode,Eclipse,IntelliJ IDEA

  - おすすめ : **Portacle** … https://portacle.github.io
    Emacs+SLIME+SBCL+QuickLisp+Git… 定番環境の詰合せ
    Windows, Mac, Linux 版を提供、インストールも簡単

関数定義：Lisp プログラミングの中核

例：階乗を計算する

定義：　　n! = 1×2× … ×n

あるいは：　n! = 1　　　　: n=1
　　= n×(n-1)! : n>1

再帰的定義による階乗

Lisp では２番目の定義（再帰的定義）を素直に書ける

定義：　n! = 1　　　　: n=1
　　= n×(n-1)! : n>1

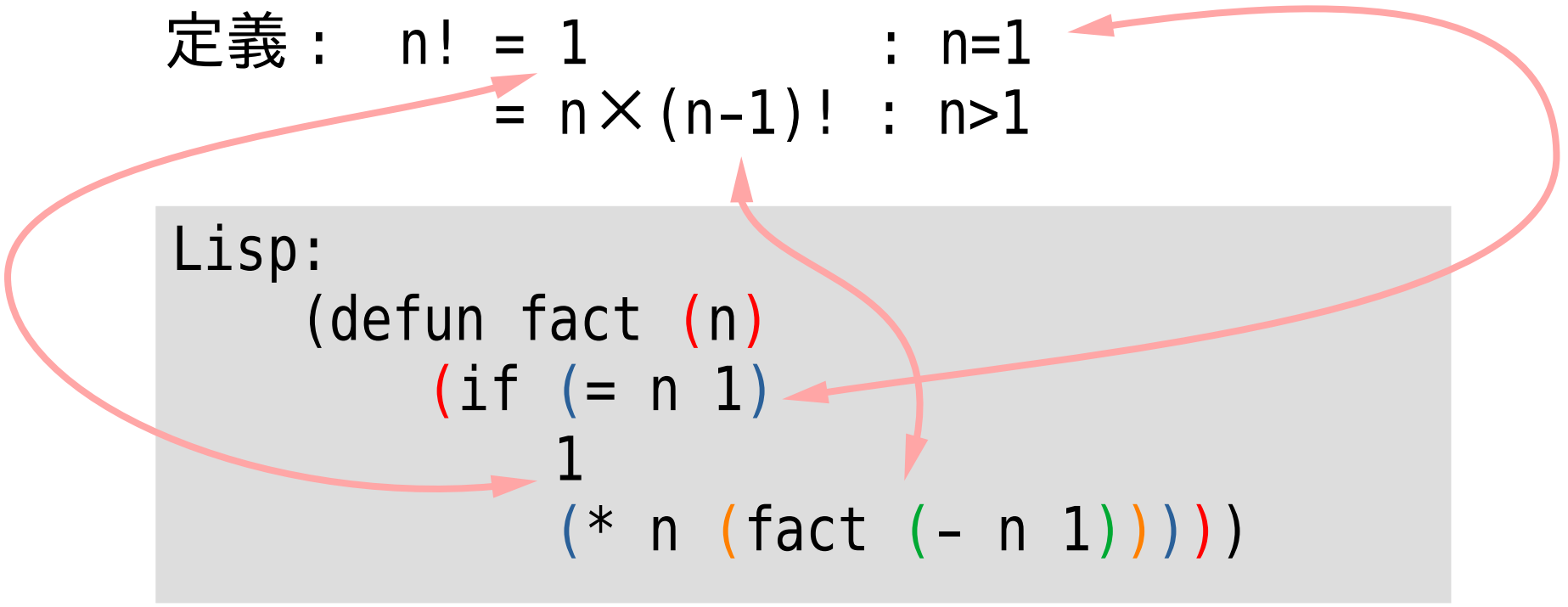
Lisp:

```
    (defun fact (n)
        (if (= n 1)
            1
            (* n (fact (- n 1)))))
```

再帰的定義による階乗

Lisp では２番目の定義（再帰的定義）を素直に書ける

定義：　n! = 1　　: n=1
　　= n×(n-1)!　: n>1



Lisp:

```
(defun fact (n)
    (if (= n 1)
        1
        (* n (fact (- n 1)))))
```

定義した関数の実行 (1/2)

REPL で関数を定義し、それを呼び出す。

```
* (defun fact (n)
      (if (= n 1)
          1
          (* n (fact (- n 1)))))
FACT
* (fact 4)
24
*
```

定義した関数の実行 (2/2)

トレース

```
* (trace fact)
(FACT)
* (fact 4)
  0: (FACT 4)
    1: (FACT 3)
      2: (FACT 2)
        3: (FACT 1)
        3: FACT returned 1
      2: FACT returned 2
    1: FACT returned 6
  0: FACT returned 24
24
* (untrace fact)
T
```

(fact 4) の動作

```
(fact 4)
→ (* 4 (fact 3))
→ (* 4 (* 3 (fact 2)))
→ (* 4 (* 3 (* 2 (fact 1))))
→ (* 4 (* 3 (* 2 1)))
→ (* 4 (* 3 2))
→ (* 4 6)
→ 24
```

繰り返しによる階乗

1番目の定義： n! = 1×2× … ×n

Lisp:

```
(defun fact-loop (n)
  (do ((i 1 (1+ i))
       (ret 1 (* i ret)))
      ((< n i) ret)))
```

繰り返しによる処理は、手続き型言語で慣れているけど ...

実は**結構面倒**？

参考：C言語：

```
int fact_loop (int n) {
  int i, ret;
  for (i=1, ret=1;
       i<n;
       i++, ret*=i) {
  }
  return ret;
}
```

プログラム例 - 整数演算

整数の四則演算を Lisp の組込み関数を使わず、独自に実装してみる（「K 整数システム」と**勝手に**命名）。

データ表現

- リストを使い、その中の要素の個数を数値と見做す
- 要素が何であるかは問わない
- 要素の個数なので、表現できるのは 0 を含む自然数のみ
- 例：`()` … 0
  `(a b c)` … 3
  `(a (b c))` … 2

プログラム例 - K 加算

```
; a と b を繋げたリストを返す
(defun k+ (a b)
  (if a
      (k+ (cdr a) (cons 'k b))
      b))
```

←a が
　空リストでない：真
　空リスト： 偽

```
リスト操作関数 car, cdr, cons
    (car  '(a b c d))     → a
    (cdr  '(a b c d))     → (b c d)
    (cons 'x '(a b c d)) → (x a b c d)
```

今出てきた '(a b c d) の「'」って何？

'(a b c d) は、

「(a b c d) はデータですよ、関数呼び出しとかしませんよ」という意味。評価するとリストそのものが返る。

```
'(a b c d) → (a b c d)
```

逆に ' を付けないと、「(a b c d) は関数呼び出しですよ。b, c, d を引数として関数 a を呼び出しますよ」という意味になる。

'(a b c d) は (quote (a b c d)) の省略形 (構文糖)

```
'(a b c d) = (quote (a b c d))
```

関数 K+ の実行

トレース

```
* (trace k+)
(K+)
* (k+ '(u v w) '(y z))
  0: (K+ (U V W) (Y Z))
    1: (K+ (V W) (K Y Z))
      2: (K+ (W) (K K Y Z))
        3: (K+ () (K K K Y Z))
        3: K+ returned (K K K Y Z)
      2: K+ returned (K K K Y Z)
    1: K+ returned (K K K Y Z)
  0: K+ returned (K K K Y Z)
(K K K Y Z)
*
```

→ K+ をトレース

→ 3+2 を計算

→ 自分自身を呼び出すたびに
・引数１の先頭要素を削る
・引数２の先頭に要素を足す
→ 引数１が空になったら
引数２を返す

→ 5

プログラム例 - K 減算

```
;; a と b の要素を一つずつ削って、 b が空リストに
;; なったとき残った a の要素の個数が演算結果
(defun k- (a b)
  (if b
      (k- (cdr a) (cdr b))
      a))
```

関数 K- の実行

トレース

```
* (trace k-)
(K-)
* (k- '(u v w) '(y z))
  0: (K- (U V W) (Y Z))
    1: (K- (V W) (Z))
      2: (K- (W) NIL)
      2: K- returned (W)
    1: K- returned (W)
  0: K- returned (W)
(W)
*
```

→ K+ をトレース

→ 3-2 を計算

→ 自分自身を呼び出すたびに
　・引数 1, 2 共先頭要素を削る
→ 引数 2 が空になったら
　引数 1 を返す

→ 1

プログラム例 - K 乗算

```
;;  乗算 …  a 中の要素の回数、 b を繋げてゆく
(defun k* (a b)
  (k*sub a b '()))

;;  補助関数 … 引数 c に計算過程を積んでゆく
(defun k*sub (a b c)
  (if a
      (k*sub (cdr a) b (k+ b c))
      c))
```

プログラム例 - K 等値判定

```
;;  等値判定 …  a が空リストのとき b も空リストか？
(defun k= (a b)
  (if a
      (if b
          (k= (cdr a) (cdr b))
          nil)
      (if b
          nil ; Lisp では NIL は偽、それ以外は
          t))); 真と解釈される ( シンボル T で表す )
```

プログラム例 - K 大小比較

```
;;  大小比較 …  a が空リストになったとき b は？
(defun k< (a b)
  (if a
      (if b
          (k< (cdr a) (cdr b))
          nil)
      (if b
          t
          nil)))
```

K整数の演算例

```
* (k+ '(a a) '(b b b))
(K K B B B)
* (k* '(a a) '(b b b))
(K K K K K K)
* (k< '(a a) '(b b b))
T
* (k< '(a a a) '(b b))
NIL
* (k= '(a a a) '(b b))
NIL
* (k= '(a a) (k- '(b b b) '(b)))
T
```

```
2+3
 5
2×3
 6
2 < 3
  真
3 < 2
  偽
3 = 2
  偽
2 = 3 - 1
  真
```

K 整数を使って階乗計算を定義

```
* (defun kfact (n)
    (if (k= n '(k))
        '(k)
        (k* n (kfact (k- n '(k))))))
```

P9-10 で説明した普通の階乗

```
  (defun fact (n)
    (if (= n 1)
        1
        (* n (fact (- n 1)))))
```

↑
使用している関数
← が異なるが、定義
の形は同じ。

K 階乗の実行

```
* (defun kfact (n)
      (if (k= n '(k))
          '(k)
          (k* n (kfact (k- n '(k))))))
KFACT
* (kfact '(a a a a a a a))
(K K K K … 5040 個の要素 … K K K K)
* (fact 7)  ← 普通の階乗
5040
```

K 整数システムを実装してみて

特徴

- リストによる新しいデータ表現 → シンプルかつ容易
  - リスト処理だけで基本演算を定義
  - 64 行程度で実装 (C 言語を使うとしたら…？)
- (この例では) 実行効率は度外視
  大きな数 → 長いリスト → メモリを喰う

更なる拡張

- 演算子の追加 (除算が未実装)
- 負の数、有理数、実数に対応

  → データ表現の変更が必要
- 桁の導入
- Lisp の数値型との相互変換

おまけ：ベンチマーク

SINNET に投稿された 30 年前のベンチマークと今の処理系を比較

```
  * Note 3/26(COMN 11) Computer Language
[ RESPONSE 311/940 ]
Title   : Wirth の部屋
SINxxxx [XXXXXX]  92/06/14  13:08:19  サイズ： 1322 バイト
Subtitle: ベンチマーク

  Lisp の有名なベンチマークプログラムに tak というのがあります。tak の定
義は以下のとおりです。
～略～
  この tak 関数を (tak 18 12 6) として呼び出します。この引数で呼び出すと、
tak は 63609 回の呼出しを行い、関数の結果は 7 になります。手持ちの言語処
理系で同様のプログラムを書き、テストしてみました。

言語      処理系              時間      備考
--------  ----------------  -------  --------------------------
Lisp      Lisp-09 2.10      1 分 22 秒
Basic     Basic09           2 分 24 秒   パックあり／なしともほぼ同じ
C         Microware C 1.1.4      4 秒
Modula-2  Modula-2/09 2.2.0      7 秒   スタック、レンジチェックあり

                              ハードウェア： FM-11EX (CPU: MC6809 2MHz)
                                    ＯＳ： OS-9/6809 Level-II 1.2J

  う～ん、やっぱりＣが一番速いのか‥‥（笑）。ちなみに参考文献によると、
Cray-1 PSL というスーパーコンピュータ上の Lisp 処理系では、0.048 秒だそ
うです。みなさんのシステムではいかがでしょうか。
```

```
* (defun tak (x y z)
    (if (>= y x)
      z
      (tak (tak (1- x) y z)
           (tak (1- y) z x)
           (tak (1- z) x y))))

* (tak 18 12 6)
```

↑

tak を SLIME に貼り付けて実行してみる

さらなる Topics

- マクロ
- オブジェクト指向

  CLOS - Common Lisp Object System
- DSL (Domain Specific Language)
  - LOOP
  - FORMAT
- コンディションと再起動
- ジェネリックセッター
- 関数型プログラミング
  - 継続
  - 遅延評価

などなど .....

参考情報 - Web

- Common Lisp Hyperspec
  - オンラインマニュアル、 SLIME から直接参照可
    http://www.lispworks.com/documentation/HyperSpec/Front/
- Common Lisp Programming
  - M.Hiroi 氏による解説
    http://www.nct9.ne.jp/m_hiroi/clisp/
- Portacle: Common Lisp のオールインワン環境
  - 統合開発環境 Portacle のチュートリアル
    https://speakerdeck.com/masatoi/portacle-common-lispfalseoruinwankai-fa-huan-jing
- Practical Scheme
  - Scheme 処理系 Gauche の作者 川合史朗氏のサイト
    https://practical-scheme.net/

参考情報 - Web(読み物)

- JPL （ジェット推進研究所）における Lisp の顛末
  https://postd.cc/lisping-at-jpl/
- NASA Programmer Remembers Debugging Lisp in Deep Space
  https://thenewstack.io/nasa-programmer-remembers-debugging-lisp-in-deep-space/
- Lisp: 読み物
  - Practical Scheme 内のリンク集
    … イマドキの話も沢山
    http://practical-scheme.net/wiliki/wiliki.cgi?Lisp%3A%E8%AA%AD%E3%81%BF%E7%89%A9

参考情報 - 書籍

- Land of Lisp
  Conrad Barski, M.D. 著、川合史朗 訳
- 実践 Common Lisp
  Peter Seibel 著、 佐野匡俊 他 共訳
- ANSI Common Lisp
  Paul Graham 著、久野雅樹、須賀哲夫 訳
- Scheme 手習い 直感で学ぶ Lisp
  Daniel P. Friedman, Matthias Felleisen 著、
  元吉文男、横山晶一 訳

ほとんど絶版なのでオンラインで入手しましょう。

まとめ

- Lisp は OS を問わず、手軽に導入できます
- REPL を使った電卓あそびから、高度なアプリケーションの構築まで、幅広く使えます
- 今回紹介したのは Lisp のほんの入り口です。前出の参考資料などで深堀りしてみて下さい