プログラム言語論
(Lisp)

櫻井彰人

Lisp, 1960

✓ Historical Lisp
- 展望
  - 古いアイデアには古いものもある
  - 古いアイデアには新しいものもある
- エレガントな、極めてコンパクトな言語
- Cとは異世界: 別の考え方をするよい機会
- 言語設計における多くの一般的課題を含む

✓ 参考文献
- McCarthy, Recursive functions of symbolic expressions and their computation by machine, *Communications of the ACM,* Vol 3, No 4, 1960.

John McCarthy

✓ AI の先駆者
- 常識推論を定式化
- ....

✓ そして
- 時分割方式の提案
- Mathematical theory
- ....

✓ Lisp
記号計算への興味から生まれる
(math, logic)

例

(+ 4 5)
値は 9

(+ (+ 1 2) (+ 4 5))
まず1+2, 次に 4+5, 次に 3+9 を評価し値をえる

(cons (quote A) (quote B))
アトム A と B のドット対

(quote (+ 1 2))
評価するとリスト (+ 1 2) になる

'(+ 1 2)
(quote (+ 1 2)) の略記

Lisp 要約

✓ 多くの方言
- Lisp 1.5, Maclisp, …, Scheme, ...
- CommonLisp には多くの付加機能あり
- 本講義: Lisp 1.5 の一部を近似的に
  なお、静的/動的スコープについてはあとで

✓ 単純な構文
(+ 1 2 3)
(+ (* 2 3) (* 4 5))
(f x y)
解析が容易. (先回りすると: データとしてのプログラム)

アトムとドット対

✓ アトムとは、数や分解できない文字列である
<atom> ::= <smbl> | <number>
<smbl> ::= <char> | <smbl><char> | <smbl><digit>
<num> ::= <digit> | <num><digit>

✓ ドット対(dotted pairs)
- 対を (A . B) と記す
- 記号式(symbolic expressions), 略して *S-式*:
  <sexp> ::= <atom> | (<sexp> . <sexp>)

基本関数

✓アトムとドット対に対する関数:
cons car cdr eq atom
✓宣言と制御:
cond lambda define eval quote
✓例
(lambda (x) (cond ((atom x) x) (T (cons 'A x))))
function f(x) = if atom(x) then x else cons("A",x)
✓副作用のある関数
rplaca rplacd set setq

式の評価

✓Read-eval-print ループ
✓関数呼び出し (function arg_1 ... arg_n)
- 引数一つ一つを評価
- 引数値のリストを関数に渡す

✓特殊形式(special form)は引数評価をしない
- 例 (cond (p_1 e_1) ... (p_n e_n))
  - 左から右へ進む
  - 値が真となる最初の p_i をみつけ, この e_i をeval
- 例 (quote A) は A を評価しない

例

(+ 4 5)
値は 9
(+ (+ 1 2) (+ 4 5))
まず1+2, 次に 4+5, 次に 3+9 を評価し値をえる
(cons (quote A) (quote B))
アトム A と B のドット対
(quote (+ 1 2))
評価するとリスト (+ 1 2) になる
'(+ 1 2)
(quote (+ 1 2)) の略記

例(M式)

plus[4; 5]
値は 9
plus[plus[1; 2]; plus[4; 5]]
まず1+2, 次に 4+5, 次に 3+9 を評価し値をえる
cons[A; B]
アトム A と B のドット対
(PLUS 1 2)
定数リスト (PLUS 1 2)

McCarthy の 1960 論文

✓興味深い論文
- 言語に対する良いアイデアと正確な記述
- 歴史的背景が感じられる
- 言語設計プロセスが垣間見える

✓重要な概念
- 記号計算への興味が設計に影響した
- 単純な計算機構モデル
- 理論的裏付けへの配慮
  帰納関数論, ラムダ計算
- 様々なよいアイデア: データとしてのプログラム, ゴミ集め (garbage collection)

Lisp を作った動機

✓Advice Taker (1958)
- 平叙文が入力されるごとに処理し、論理的推論を行う

✓記号積分・微分
- 式の表現 --> 積分の表現
  (integral `(lambda (x) (times 3 (square x))))

✓応用を動機とする、良い言語設計
- 重要な目標を念頭に
- 目立つけど本質ではないアイデアは捨てる

Lisp	記号計算, 論理, 実験的な言語.
C	Unix OS
Simula	シミュレーション
PL/1	あらゆるものの統合, 長期的には不出来

実行モデル (抽象機械)

- ✓言語の意味が定義されるべき
  - 具体的すぎると
    - プログラムは移植不能, 個別アーキテクチャと不即不離
    - 最適化の邪魔 (e.g., C では式の評価順序が未定義)
  - 抽象的すぎると
    - 実行時間、容量等の見積りが容易でない
- ✓Lisp: IBM 704, しかし限られた概念のみ …
  - 番地, 減分レジスタ -> 2部分をもつセル
  - GCによってメモリを抽象化

抽象機械

- ✓ 抽象機械の概念:
  - 理想化した計算機, プログラムの直接実行
  - 実行に対するプログラマの心的イメージを反映
  - 具体的すぎず, 抽象的すぎず
- ✓ 例
  - Fortran
    - 構造のないレジスタマシン; 線型番地のメモリ配置
    - スタックなし, 再帰なし.
  - Algol family
    - スタックマシン, スコープの等高線(contour)モデル, ヒープ
  - Smalltalk
    - オブジェクト, メッセージ通信による交信.

理論的根拠

- ✓" … scheme for representing the partial recursive functions of a certain class of symbolic expressions."
- ✓Lisp が用いたもの
  - 計算可能 (帰納的部分) 関数
    - *全*ての計算可能関数を表現したい
  - 関数の表現
    - ラムダ計算から (提唱者 A. Church)
    - ラムダ計算はTuring機械と等価, しかしより使いやすい構文と計算規則を提供してくれる

Lisp 設計における革新

- ✓式-指向
  - 関数式
  - 条件式
  - 帰納関数
- ✓メモリの抽象化
  - 番地が付いた場所の列ではなく、セル
  - ゴミ集め(garbage collection)
- ✓データとしてのプログラム
- ✓高階関数

品詞:文法を定義に用いる用語

- ✓文 load 4094 r1
  - 命令
  - アクセス可能メモリの内容を変更する
- ✓式 (x+5)/2
  - 構文的な要素。評価の対象
  - 値をもつ。アクセス可能メモリを変更するとは限らない
  - もし変更するなら, 副作用(*side effect*)があるという
- ✓宣言 int x
  - 新しい識別子の導入
  - 識別子に値を束縛することもある。型を指定、等

関数式

- ✓ 例:
  (lambda (*parameters*) (*function_body*))
- ✓ 構文はラムダ計算の構文に基づく:
  λx.λy. fxy ⇒ (lambda (x y) (f x y))
  λf. λx. f (f x)
  (lambda (f) (lambda (x) (f (f x))))

関数式は関数を定めるが、それに名前をつけることはしない.

```
( (lambda (f) (lambda (x) (f (f x))))
  (lambda (y) (+ 2 y)))
)
```

Lisp の条件式

✓一般化された if-then-else

(cond $(p_1\ e_1)$ $(p_2\ e_2)$ … $(p_n\ e_n)$)

– 条件 p_1 … p_n を左から右に評価する

– もし p_i が最初に真となる条件なら, e_i を評価する

– e_i iの値がこの条件式の値となる

未定義(undefined)になるのは、真となる p_i がないか

p_1 … p_i が偽で p_{i+1} が未定義, か
当該 p_i が真で e_i は未定義

条件文はアセンブラにある
条件式が初めて現れたのは、Lisp

例

```
(cond ((< 2 1) 2)  ((< 1 2) 1))
```

の値は 1

```
(cond ((<2 1 ) 2)  ((<3 2) 3))
```

は未定義

```
(cond (diverge 1) (T 0))
```

は未定義, 但し diverge が未定義とする

```
(cond (T 0) (diverge 1))
```

の値は 0

Strictness

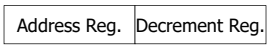
✓演算子や式形が *strict*（厳密、正格）であるのは、全ての被演算項や部分式が値を持つときに限り、それが値をもつ場合である。

✓Lisp の cond はstrict ではない, しかし加算は strict である

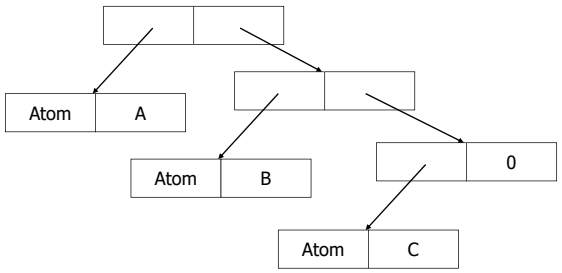
- (cond (true 1) (diverge 0))
- (+ e_1 e_2)

Lisp のメモリモデル

✓Cons セル

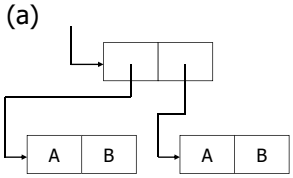


✓セルで表したアトムとリスト

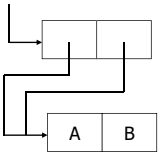


共有

(a)



(b)



✓印刷すれば、どちらも、 (A.B).(A.B)

✓どちらが、下式の評価の結果か？

(cons (cons 'A 'B) (cons 'A 'B)) ?

ゴミ集め(Garbage Collection)

✓ ゴミ(garbage):
プログラム *P* の実行時のある時点で、メモリ内の場所 *m* がゴミ (*garbage*) であるのは、この時点以降 *P* を継続して実行しても、場所*m* にアクセスしないことである

✓ ゴミ集め(garbage collection):

- プログラムの実行中にゴミを見つける
- GC が起動されるのは、メモリが必要となったとき
- 決定は実行時系(run-time system)によってなされるのであって, プログラムによるわけではない

これは非常に有用である。例: 整書プログラムを作成するとき, プログラマが使う時間の ~40% はメモリ管理に対してである.

例

(car (cons (e_1) (e_2)))

e_2 の評価時に作られたセルはゴミかもしれない,
e_1 や他のプログラム部分と共有していない限り

((lambda (x) (car (cons (… x…) (... x ...)))
'(Big Mess))

この cons セルの car と cdr は、重複した構造を指している可能性がある.

Mark-and-Sweep アルゴリズム

- ✓各データには tag bit が付随していると仮定
- ✓プログラムが用いたヒープの場所のリストが必要
- ✓アルゴリズム:
  - 全ての tag bit を 0 にする.
  - プログラムで直接使用されている場所から開始。全てのリンクを辿って、tag bit を1 にしていく
  - tag = 0 であるセルをすべて、未使用リストに載せる

何故 Lisp でゴミ集めを?

- ✓McCarthy の論文には、この方法は
  - " … more convenient for the programmer than a system in which he has to keep track of and erase unwanted lists."
- ✓この推論は同様に有効に C に適用できるであろうか?
- ✓ゴミ集めはLISPに対しての方が C に対するより "more appropriate" か? 何故?

データにもなるプログラム

- ✓プログラムはデータと同じ方法で表現される
- ✓Eval 関数はリスト要素の評価に用いられる
- ✓例: z 中の y を x に置き換えて評価する

```
(define substitute (lambda (x y z)
    (cond   ((atom z) (cond ((eq z y) x ) (T z)))
            (T  (cons (substitute x y (car z))
                      (substitute x y (cdr z))))))
(define substitute-and-eval
    (lambda (x y z) (eval (substitute x y z))))
```

再帰関数

- ✓以下の関数 f を式で表現したい

  (f x) = (cond ((eq x 0) 0) (true (+ x (f (- x 1)))))
- ✓試しに

```
(lambda (x) (cond ((eq x 0)  0)  (true  (+ x (f  (- x
  1))))))
```

  としても、関数本体中の f は未定義.
- ✓McCarthy (1960) の解は演算子 "label"

```
(label f
   (lambda (x) (cond ((eq x 0) 0) (true (+ x (f (- x
  1)))))))
```

高階関数

- ✓ 関数であって、次のいずれか
  - 関数を引数とする
  - 関数を結果とする
- ✓ 例: 関数合成

```
(define  compose
    (lambda (f  g)   (lambda (x)  (f  (g   x)))))
```

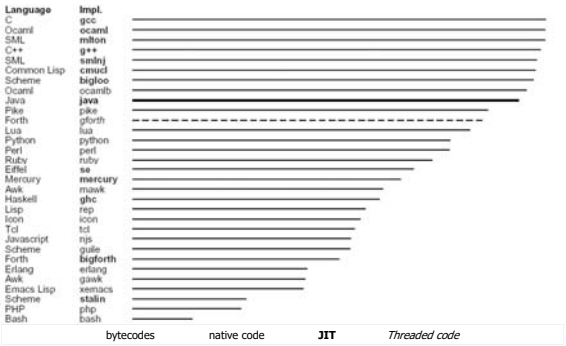
- ✓ 例: maplist

```
(defun  maplist  (f  x)
    (cond  ((null x)   nil)
           (true  (cons (f (car x))  (maplist f (cdr x))))))
```

効率と副作用(Side-Effects)

- ✓Pure Lisp: 副作用なし
- ✓“効率”のために導入された演算
  (rplaca x y) replace car of cell x with y
  (rplacd x y) replace cdr of cell x with y
- ✓ここで“効率”は何を意味するのか?
  - (rplaca x y) は (cons y (cdr x)) より速い?
  - より速いことはいつもより良いことか?

言語の速度(?) Language speeds



http://www.bagley.org/~doug/shootout (link切れ)
http://shootout.alioth.debian.org/ *Completely Random and Arbitrary Point System*

GNU Emacs / Mule と Lisp

- ✓ Emacs Lisp: Emacs / Mule の Lisp インタプリタ
  - Emacs / Mule を引数無しで起動すると、最初に `*scratch*' という名前のバッファがあるはず。
  - モードラインの右側を見て、`(Lisp Interaction)' の表示がない場合、M-x lisp-interaction-mode と入力する。
  - そのバッファ内で Lisp のプログラムを入力し、カーソルが最後の `)' の右にあるときに C-j を押すと実行される。
  - 又は C-x C-e を押すと結果が mini-buffer にでる
- ✓ xyzzy, Emacs (Windows) / Emacs (Mac)
- ✓ GCL - GNU Common Lisp
  - http://www.gnu.org/software/gcl/
- ✓ CMUCL:
  - http://www.cons.org/cmucl/
- ✓ 他の実装(大分と古い)
  - http://jp.franz.com/jlug/ja/resources/implementations.html

もうちょっと真剣に試してみるには

- ✓Ruby 上のものがある。
  - http://taw.github.io/rlisp/
- ✓Gouche (Scheme)
  - http://practical-scheme.net/gauche/index-j.html
- ✓旧 PLT Scheme
  - http://racket-lang.org/

```
ruby rlisp.rb –i とする。ただし、
rlisp.rb 内で、
parser.each_expr{  ...
}
↓
begin
  parser.each_expr{  ...
  }
  rescue => exc
  if [RuntimeError,
    NoMethodError,
    ZeroDivisionError]
        .member?(exc.class) then
    p exc
    retry
  end
end
```

プログラム例

```
(defun fact (n)
   (cond ((= n 0) 1)
         ((< n 0) nil)
         (t       (* n (fact (1- n))))))


(defun insert (x y)
       (cond ((or (null y) (< x (car y))) (cons x y))
             (t (cons (car y) (insert x (cdr y))))))
```

プログラム例

```
(log (sin 1))

(length '(coffee milk (chocolate cake)))
(length '((A (B) (C (D)))) (E () F))

(car '(a b c d))
(cdr '(a b c d))
(cons (car '(a b c d)) (cdr '(a b c d)) )

(defun cadr (x) (car (cdr x)))
(cadr '(a b c d))
(defun caddr (x) (car (cdr (cdr x))))
(caddr '(a b c d))
```

プログラム例

```
(null ())
(null 'a)
(null nil)
```

```
(eq 1 3)
(eq 2 2)
(eq 'a 'a)
(eq 'a 'b)
(eq '(a b) '(a b))
(eq '2 '2)
(eq '(2) '(2))
```

```
(equal 1 3)
(equal 2 2)
(equal 'a 'a)
(equal 'a 'b)
(equal '(a b) '(a b))
(equal '2 '2)
(equal '(2) '(2))
```

プログラム例

```
(setq x 1)
(cond ((eq x 1) (car '(a b))) (t (car (cdr '(a b)))))

(defun test1 (x)
 (cond ((eq x 1) (car '(a b)))
       (t (car (cdr '(a b))))))
(test1 1)
(test1 2)

(atom 1)
(atom ())
(atom nil)
(atom '(a 1))
```

プログラム例 (Scheme)

```
(define x 1)
(cond ((eq? x 1) (car '(a b))) (else (car (cdr '(a b)))))

(define (test1 x)
 (cond ((eq? x 1) (car '(a b)))
       (else (car (cdr '(a b))))))
(test1 1)
(test1 2)

(list? 1)
(list? '())
(define nil '())
(list? nil)
(list? '(a 1))
```

プログラム例

```
(defun dupl (x)
  (cond  ((atom x)  x)
         (t  (cons (dupl (car x)) (dupl (cdr x))))
  )
)
(dupl '(a b (a b)))
```

プログラム例

```
(append '(a b) '(c d) )
(defun app (x y)
  (cond
    ((null x)  y )
    (t  (cons (car x) (app (cdr x) y)) )
)  )
(app '(a b) '(c d) )

(reverse '(a b c d))
(defun rev (x)
  (cond ((null x)  nil)
        ( t  (app (rev (cdr x)) (cons (car x) nil)))
)  )
(rev '(a b c d))
```

プログラム例

```
(defun rev (x)
  (cond ((null x)  nil)
        ( t  (app (rev (cdr x)) (cons (car x) nil)))
)  )

(defun rev (x)  (rev2 x nil))
(defun rev2 ( x y )
  (cond  ((null x) y)
         (t (rev2 (cdr x) (cons (car x) y)))
)  )
(rev2 '(a b)  nil)
(rev2 '(a b)  '(c d))
(rev  '(a b c d))
```

プログラム例

```
(defun hanoi (n)
  (__hanoi n 'a 'b 'c))

(defun __hanoi (n from via to)
  (cond ((eq n 1)
              (print (list 'move 'disk from '=> to)) )
        (t
         (__hanoi (- n 1) from to via)
         (__hanoi 1  from via to)
         (__hanoi (- n 1) via from to))))

(hanoi 3)
```

http://aten.aial.hiroshima-u.ac.jp/~kakugawa/clisp/index-ja.shtml

まとめ: Lisp の貢献

✓ 成功した言語
- 記号計算, 実験向けプログラミング

✓ 言語のアイデア
- 式-指向: 関数と帰納
- 基礎データ構造としてのリスト
- データとしてのプログラム, そして万能関数 `eval`
- 帰納を “public pushdown list” を用いて実装
- ゴミ集めのアイデア.