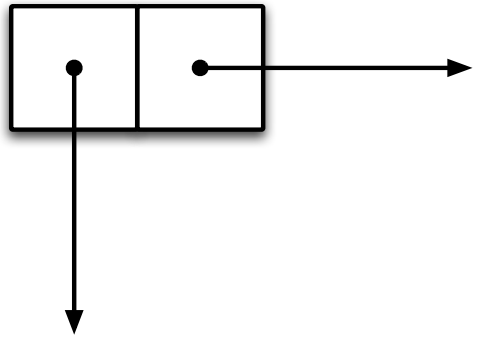
セルの海　マクロの空

山本和彦
(株)インターネットイニシアティブ
kazu@iij.ad.jp

セルはどうして
2 つのポインタなのか？



Lisp のマクロは
どんな風に使うのか？

データと関数の区別がない
とはどういうことか？

おしながき

- Lisp ことはじめ
- セルはどうして2 つのポインタなのか？
- Lisp のマクロはどんな風に使うのか？
- データと関数の区別がないとはどういうことか？

Lisp ことはじめ

Lisp の立場

- アメリカの求人広告
  - Gustavo Duarte 氏が Dice.comで調べた結果
    - http://duartes.org/gustavo/blog/post/2008/04/03/ → Programming-Language-Jobs-and-Trends.aspx

16,479件	Java	
8,080件	C++	
7,780件	C#	
6,749件	JavaScript	
5,710件	Perl	
2,641件	PHP	
1,408件	Python	
1,207件	COBOL	
769件	Ruby	
33件	Lisp	Java の 0.2% orz

強がり

- 「Javaスクールの危険」
  - Joel Spolsky 氏
    - http://local.joelonsoftware.com/mediawiki/index.php/Javaスクールの危険

私の知るSchemeとHaskellとCのポインタが使える人はみな、
Javaを使い始めて2日で経験5年のJavaプログラマより
いいコードを書くようになる。

しかしそのことが平均的な頭の鈍い人事部の連中には
理解できないのだ。

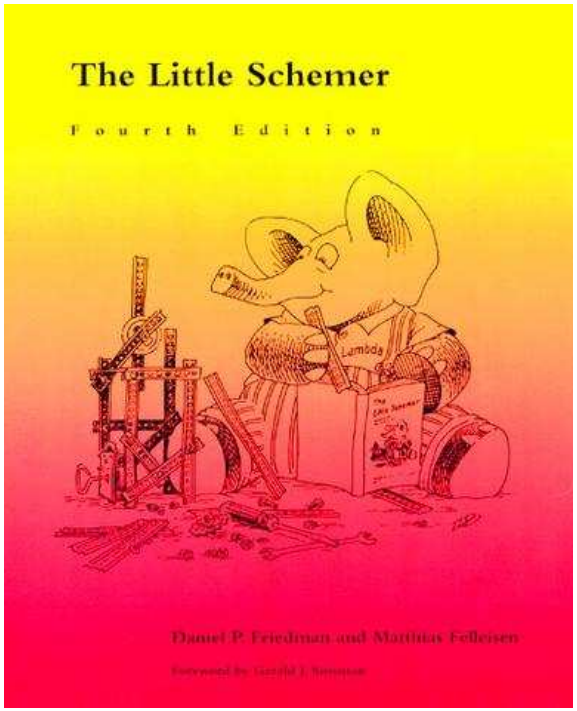
Lisp の流派

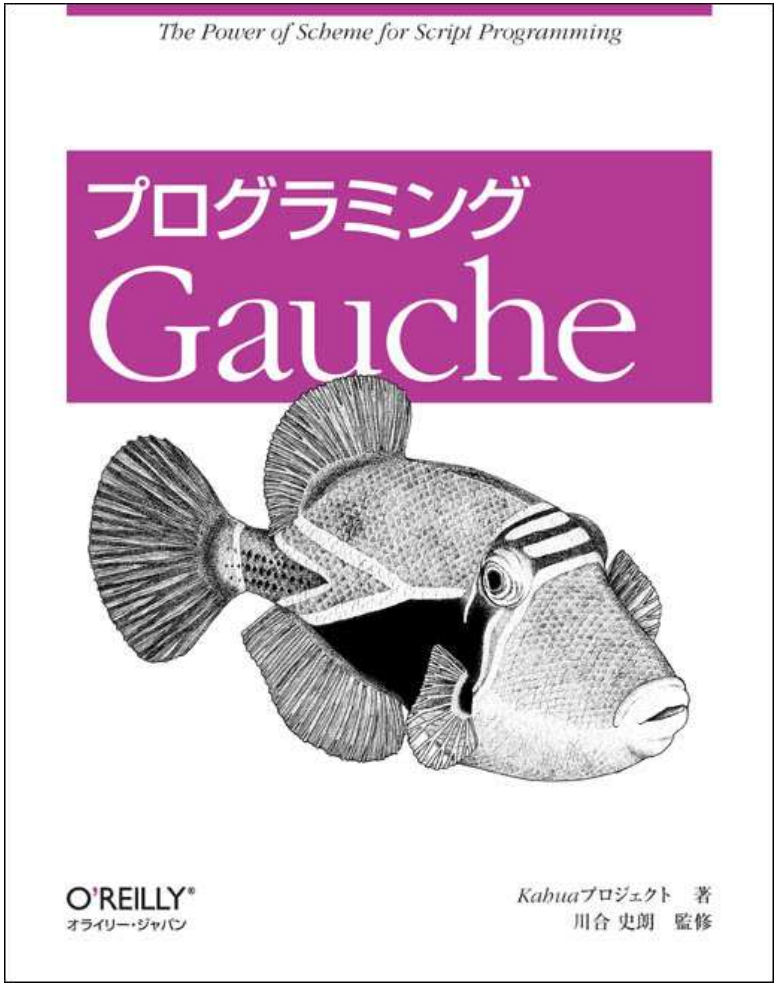
- Lisp には方言が多かった
  - Fortran の次に古い高級言語
- Common Lisp
  - 大きいことはいいことだ
    - (C) 中村正三郎
- Scheme
  - 小さいことはいいことだ
    - (C) 中村正三郎
  - 仕様(R5RS)は、たったの 50 ページ
    - Revised^5 Report on Algorithmic Language Scheme
- Emacs Lisp
  - 使われることはいいことだ
    - (C) 山本和彦

僕は Emacs Lisper ですが

今日は Scheme でお話しします

なぜ Scheme か？


The Little Schemer
Fourth Edition
Lambda
Daniel P. Friedman and Matthias Felleisen


The Power of Scheme for Script Programming
プログラミング
Gauche
O'REILLY®
オライリー・ジャパン
Kahuaプロジェクト 著
川合 史朗 監修

Lisp に目を慣らす

Scheme のプログラム例

- 関数定義

```
(define atom?
  (lambda (x)
    (and (not (pair? x))
         (not (null? x)))))

(define member*
  (lambda (a l)
    (cond
     ((null? l) #f)
     ((atom? (car l))
      (or (eq? (car l) a)
          (member* a (cdr l))))
     (else
      (or (member* a (car l))
          (member* a (cdr l)))))))
```

- 評価 (実行)

```
(member* 'foo '(boo (foo woo) goo))
→ #t
```

Lisp の読み方、書き方

- 読み方
  - 括弧なんてみない
  - インデントだけみる

```
(define member*
  (lambda (a l)
    (cond
     ((null? l) #f)
     ((atom? (car l))
      (or (eq? (car l) a)
          (member* a (cdr l))))
     (else
      (or (member* a (car l))
          (member* a (cdr l)))))))
```

- 書き方
  - インデントは、エディタにお任せ
  - 括弧の対応関係も、エディタにお任せ

Lisp の基礎

アトム

- すべてのオブジェクトは、まず評価される
- 真偽値
  - `#t` → `#t`
  - `#f` → `#f`
- 数値
  - `10` → `10`
- 文字列
  - `"Hello, World"` → `"Hello, World"`
- シンボル
  - `counter` → `10` ; データ
  - `member*` → `(lambda ... )` ; 関数
- シンボル自身を表すときは quote する
  - `'counter` → `counter`
  - `'member*` → `member*`
  - `(quote member*)` → `member*`

データとしてのリスト

- リストは "("と ")" で囲む
- クオートが前に付いていればデータ
- リスト
  - `'(0 1 2 3)` → `(0 1 2 3)`
  - `'(+ 0 1 2 3)` → `(+ 0 1 2 3)`
- 空リスト
  - `'()` → `()`

関数呼び出しとしてのリスト

- クオートが付いてなければ関数呼び出し
- 最初の要素が関数。残りの要素が引数
  - (+ 1 2 3) → 6
  - *cf* $1 + 2 + 3$
- 関数呼び出しの前に引数が評価される
  - x が 3 のとき

```
    (+ (* x x) x 1)
  → (+ (* 3 3) 3 1)
  → (+ 9 3 1)
  → 13
```

  - *cf* $x^2 + x + 1$
- 評価されない引数がある特殊形式
  - quote, cond, if, and, or
    - and, or, cond は if で作れる
  - define, lambda
    - lambda は、なぜか特殊形式とは呼ばれない

関数定義

- 先頭の要素がシンボル `lambda` であるリストは関数
  - `(lambda (n) (+ n 1))`
  - lambda は quote されている感じ
- 最後の式が返り値
  - すべては式
  - 文はない
  - `return` なんて書かない
- 名前がなくても実行できる
  - `((lambda (n) (+ n 1)) 2)` → `3`
- 名前は `define` で付ける

```
(define plus1
 (lambda (n)(+ n 1)))
```

  - 関数名であるシンボル `plus1` は quote されている

再び Scheme のプログラム例

- 関数定義

```
(define atom?
  (lambda (x)
    (and (not (pair? x))
         (not (null? x)))))

(define member*
  (lambda (a l)
    (cond
     ((null? l) #f)
     ((atom? (car l))
      (or (eq? (car l) a)
          (member* a (cdr l))))
     (else
      (or (member* a (car l))
          (member* a (cdr l)))))))
```

- 評価 (実行)

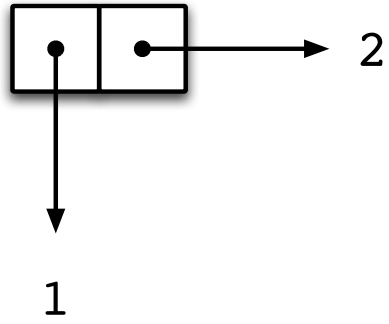
```
(member* 'foo '(boo (foo woo) goo))
→ #t
```

セルはどうして
2 つのポインタなのか？

Scheme 用語では「ペア」

セルと3つの基本関数

- cons はセルを生成する
  - malloc あるいは new に相当
  - (cons 1 2) → (1 . 2)
  - ドット対表記



- car はセルの第一要素を取り出す
  - `(car '(1 . 2))` → `1`
- cdr はセルの第二要素を取り出す
  - `(cdr '(1 . 2))` → `2`

つまらない

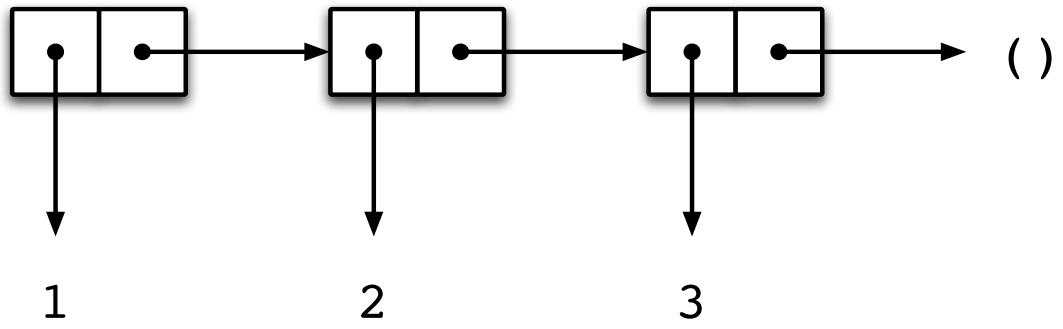
or

だからなに？

Lisp の最初の落とし穴

セルの連鎖

- セルの連なりで、最後が空リスト
  - `(cons 1 (cons 2 (cons 3 '())))`
    → `(1 . (2 . (3 . ())))`



- これはリスト
  - `(1 2 3)`

簡単な表現があるなら
先に教えてよ！

Lisp の第二の落とし穴

再帰

- セルの連なりをもう一度見る
  - `(1 . (2 . (3 . ())))`
- 入れ子だ！
- 入れ子は再帰で処理するのが自然だ！

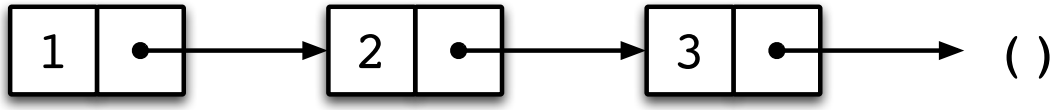
```
(define member?
  (lambda (a l)
    (cond
     ((null? l) #f)            ; ()か検査し
     ((eq? (car l) a) #t)      ; carに仕事をして
     (else
      (member? a (cdr l))))))  ; cdrに再帰する

(member? 2 '(1 2 3)) → #t
(member? 4 '(1 2 3)) → #f
```

表示された関数を
理解しようとは
思わないで下さい！

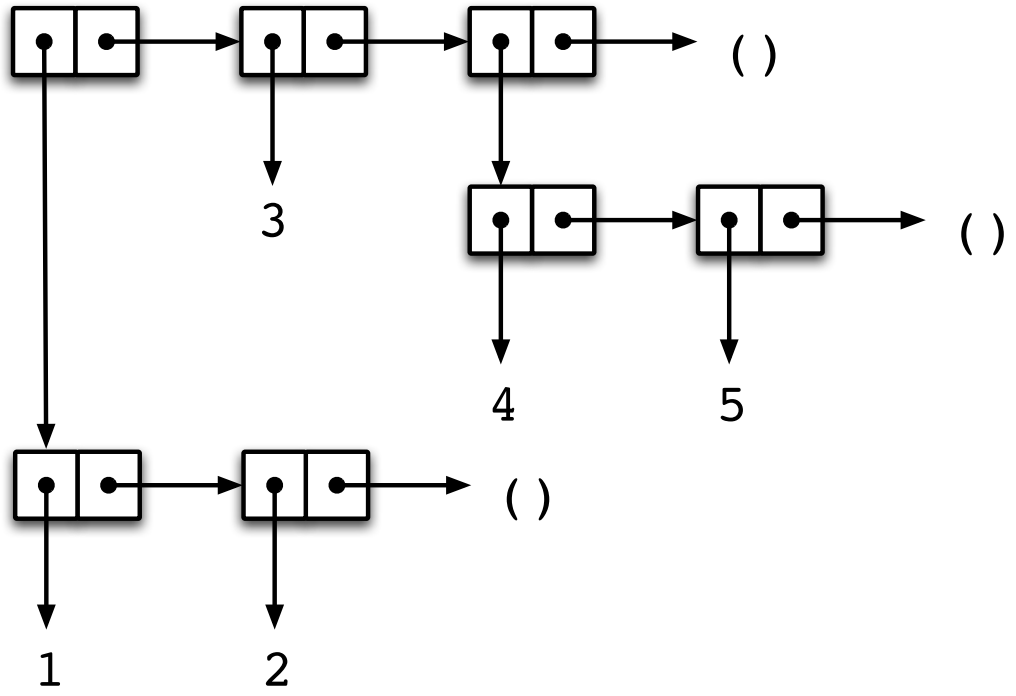
セルが1つのポインタなら？

- リストを作っておしまい



セルは2つのポインタだから

- car もセルを指せる



- リストがさらに入れ子になる
  - リストのリスト
  - ((1 2) 3 (4 5))

再帰の再帰

- 入れ子の入れ子は、再帰の再帰

```
(define member*
  (lambda (a l)
    (cond
     ((null? l) #f)
     ((atom? (car l))
      (or (eq? (car l) a)
          (member* a (cdr l))))
     (else
      (or (member* a (car l)) ; 内部のリストへ再帰
          (member* a (cdr l)))))))

(member* 5 '((1 2) 3 (4 5))) → #t
(member* 6 '((1 2) 3 (4 5))) → #f
```

連想リスト

- リストのリストは連想リスト

```
(("boo" "ワラの家")
 ("foo" "木の家")
 ("woo" "レンガの家"))
```

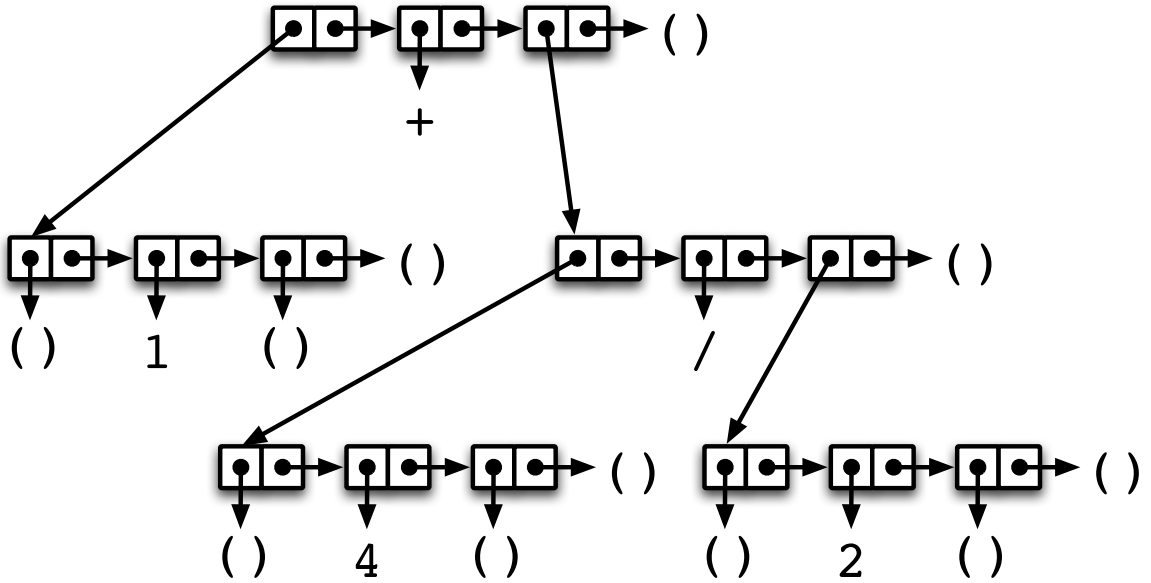
- 連想リストはハッシュの代わり

```
(assoc "foo" '(("boo" "ワラの家")
               ("foo" "木の家")
               ("woo" "レンガの家")))
→("foo" "木の家")
```

他のデータ構造

- たとえば二分木

(左の子 値 右の子)



補助関数を使ったプログラム

- 補助関数

```
(define node-get-left
  (lambda (node) (car node)))
(define node-get-value
  (lambda (node) (car (cdr node))))
(define node-get-right
  (lambda (node) (car (cdr (cdr node)))))
```

- 帰りがけ順に表示するプログラム

```
(define postorder
  (lambda (node)
    (cond
     ((null? node) '())
     (else
      (postorder (node-get-left node))
      (postorder (node-get-right node))
      (display (node-get-value node))))))

(postorder '((() 1 ()) + ((() 4 ()) / (() 2 ()))))
→ 1 4 2 / +
```

本質と効率

- 分けて考えよう
  - 本質的に必要なもの
  - 効率のために必要なもの
- 「珠玉のプログラミング」
  - Jon Bentley

  効率は、問題になるまで、
  考えない方がよいでしょう
- リスト
  - 遅ければ、配列を実装すればいい
- 連想リスト
  - 遅ければ、ハッシュを実装すればいい

Lisp のマクロは
どんな風に使うのか？

Lisp のデザイン・パターン

- 関数は小さく
- 副作用のある関数とない関数を分けろ
  - Scheme では副作用のある関数名に "!" が付く (e.g `set!`)
- 再帰しろ
- 高階関数を使え
  - 関数を引数として取る
    - 「仕事」と「データの走査」を分ける
    - map & reduce
  - 関数を返す
    - クロージャ
- マクロを書け
  - C のマクロとは別次元
  - C のマクロは嫌われ者だが、Lisp のマクロはヒーロー

例題

- FizzBuzz 問題

  1から100までの数をプリントするプログラムを書け。
  ただし
  3の倍数のときは数の代わりに□Fizz□と、
  5の倍数のときは□Buzz□とプリントし、
  3と5両方の倍数の場合には□FizzBuzz□とプリントすること。

- 「Lisp脳」の謎に迫る
  「プログラミング Gauche」のコラム

- FizzBuzz の進化
  - ループ版
  - 高階関数版
  - 直交版
  - マクロ版

FizzBuzz ループ版

- ループの中で print する

```
(define fizzbuzz
  (lambda (n)
    (do ((x 1 (+ x 1)))
        ((>= x n) x)
      (cond
       ((= (modulo x 15) 0)
        (print "FizzBuzz"))
       ((= (modulo x  3) 0)
        (print "Fizz"))
       ((= (modulo x  5) 0)
        (print "Buzz"))
       (else
        (print x))))))
```

- 実行

```
(fizzbuzz 100)
→1 2 "Fizz" 4 "Buzz" "Fizz" 7 8 "Fizz" "Buzz"
  11 "Fizz" 13 14 "FizzBuzz" 16 ... "Buzz"
```

高階関数への指針

- 「仕事」と「データの走査」を分ける
- 数のリストを用意

```
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ... 100)
```

- それぞれの要素に仕事をする

```
(1 2 "Fizz" 4 "Buzz" "Fizz" 7 8 "Fizz" "Buzz"
 11 "Fizz" 13 14 "FizzBuzz" 16 ... "Buzz")
```

数のリストを用意

```
(define count
  (lambda (beg end)
    (if (= beg end)
        (cons beg '())
        (cons beg (count (+ beg 1) end)))))

(count 1 100)
→(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ... 100)
```

仕事

```
(define fizzbuzz
  (lambda (x)
    (cond
     ((= (modulo x 15) 0) "FizzBuzz")
     ((= (modulo x  3) 0) "Fizz")
     ((= (modulo x  5) 0) "Buzz")
     (else x))))

(fizzbuzz  1) → 1
(fizzbuzz  3) → "Fizz"
(fizzbuzz  5) → "Buzz"
(fizzbuzz 15) → "FizzBuzz"
```

FizzBuzz 高階関数版

- データの走査は map にお任せ

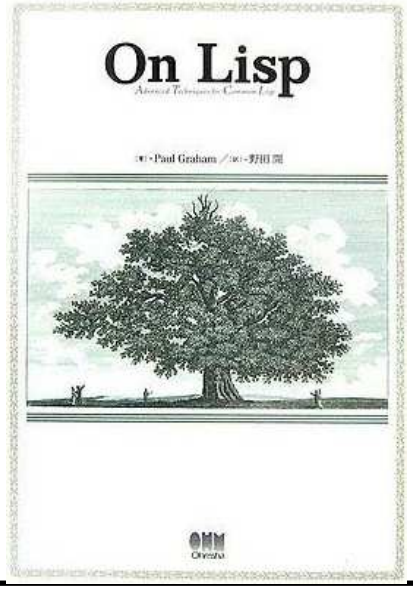
```
(count 1 100)
→(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ... 100)

(map fizzbuzz (count 1 100))
→(1 2 "Fizz" 4 "Buzz" "Fizz" 7 8 "Fizz" "Buzz"
   11 "Fizz" 13 14 "FizzBuzz" 16 ... "Buzz")
```

直交性

- 「On Lisp」
  - Paul Graham

直交的なプログラミング言語とは、
少数のオペレータを多数の様々な方法で結合させることで、
多様な意味が表現できるもののことだ。
おもちゃのブロックは極めて直交的だが、
プラモデルはほとんど直交的でない。



直交性の原則違反

- ぜんぜん直交してない！

```
(define fizzbuzz
  (lambda (x)
    (cond
     ((= (modulo x 15) 0) "FizzBuzz")
     ((= (modulo x  3) 0) "Fizz")
     ((= (modulo x  5) 0) "Buzz")
     (else x))))
```

直交性への指針

- 数のリストを用意

```
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ... 100)
```

- 内部データを用意

```
((1 "") (2 "") (3 "") (4 "") (5 "") ... (100 ""))
```

- Buzz を適応

```
((1 "") (2 "") (3 "") (4 "") (5 "Buzz")
 (6 "") (7 "") (8 "") (9 "") (10 "Buzz")
 (11 "") (12 "") (13 "") (14 "") ... (100 "Buzz"))
```

- Fizz を適応

```
((1 "") (2 "") (3 "Fizz") (4 "") (5 "Buzz")
 (6 "Fizz") (7 "") (8 "") (9 "Fizz") (10 "Buzz")
 (11 "") (12 "Fizz") (13 "") (14 "")
 (15 "FizzBuzz") (16 "") ... (100 "Buzz"))
```

- 結果を取り出す

```
(1 2 "Fizz" 4 "Buzz" "Fizz" 7 8 "Fizz" "Buzz"
 11 "Fizz" 13 14 "FizzBuzz" 16 ... "Buzz")
```

内部データ

- 数字と文字列からなる構造体

```
(define make-numstr
  (lambda (num str) (cons num (cons str '()))))

(define numstr-get-num
  (lambda (x) (car x)))

(define numstr-get-str
  (lambda (x) (car (cdr x))))
```

- 使い方

```
(make-numstr 3 "Fizz")
→(3 "Fizz")

(numstr-get-str '(3 "Fizz"))
→"Fizz"
```

内部データの生成

- 内部データの初期化

```
(define num-str-pair
  (lambda (x)
    (make-numstr x "")))

(map num-str-pair (count 1 100))
→((1 "") (2 "") (3 "") (4 "") ... (100 ""))
```

直交した仕事(1)

```
(define fizz?
  (lambda (n)
    (= (modulo n 3) 0)))

(define fizz
  (lambda (x)
    (if (fizz? (numstr-get-num x))
        (make-numstr
         (numstr-get-num x)
         (string-append "Fizz" (numstr-get-str x)))
        x)))

(fizz '(3 ""))
→(3 "Fizz")
(fizz '(5 ""))
→(5 "")
```

直交した仕事(2)

```
(define buzz?
  (lambda (n)
    (= (modulo n 5) 0)))

(define buzz
  (lambda (x)
    (if (buzz? (numstr-get-num x))
        (make-numstr
         (numstr-get-num x)
         (string-append "Buzz" (numstr-get-str x)))
        x)))

(buzz '(3 ""))
→(3 "")
(buzz '(5 ""))
→(5 "Buzz")
```

FizzBuzz 直交版

- 結果を取り出す関数

```
(define fizzbuzz-print
  (lambda (x)
    (if (equal? (numstr-get-str x) "")
        (numstr-get-num x)
        (numstr-get-str x))))
```

- 実行

```
(map fizzbuzz-print
     (map fizz
          (map buzz
               (map num-str-pair
                    (count 1 100)))))
→(1 2 "Fizz" 4 "Buzz" "Fizz" 7 8 "Fizz" "Buzz"
   11 "Fizz" 13 14 "FizzBuzz" 16 ... )
```

DRY

- 達人プログラマー
  - Andrew Hunt & David Thomas

Don’t Repeat Yourself
重複を避けること

DRY の原則違反

- 構造体の定義
  - 木構造
  - FizzBuzz の内部データ

  → マクロ
- fizz? と buzz? の定義

  → クロージャ
- fizz と buzz の定義

  → マクロ

マクロ

- Scheme のマクロ
  - define-syntax
- 伝統的なマクロ
  - defmacro
    - Common Lisp & Emacs Lisp
  - define-macro
    - Gauche の拡張
- 今日は伝統的なマクロを使います
  - 3 つの Lisp で使える
  - 機能の制約がない

構造体を定義するマクロ(1)

```
(define symbol-false-list
  (lambda (spec)
    (map (lambda (symbol) (list symbol #f)) spec)))

(define-macro (define-structure type . spec)
  `(begin
     (define-constructor ,type ,@spec)
     (define-accessors ,type ,@spec)
     ',type))

(define-macro (define-constructor type . spec)
  `(define ,(string->symbol
             (string-append
              "make-"
              (symbol->string type)))
     (lambda args
       (let-keywords
        args ,(symbol-false-list spec)
        (list ,@spec)))))
```

構造体を定義するマクロ(2)

```
(define-macro (make-accessor-name type name set?)
  `(string->symbol
    (string-append
     (symbol->string ,type)
     "-" ,(if set? "set" "get") "-"
     (symbol->string ,name)
     ,(if set? "!" ""))))

(define-macro (define-accessors type . spec)
  (let loop ((i 0) (name spec) (defs '()))
    (if (null? name)
        (cons 'begin defs)
        (loop
         (+ i 1)
         (cdr name)
         (cons
          `(define ,(make-accessor-name type (car name) #f)
             (lambda (struct)
               (list-ref struct ,i)))
          (cons
           `(define ,(make-accessor-name type (car name) #t)
              (lambda (struct val)
                (set-car! (list-tail struct ,i) val)))
           defs))))))
```

構造体の定義

- FizzBuzz の内部データ

```
(define-structure numstr num str)
→
  make-numstr
  numstr-get-num
  numstr-set-num!
  numstr-get-str
  numstr-set-str!

(make-numstr :num 3 :str "Fizz")
→(3 "Fizz")

(numstr-get-str '(3 "Fizz"))
→"Fizz"
```

- 前出の木構造

```
(define-structure node value left right)
```

比べてみよう

- 単なる関数定義

```
(define make-numstr
  (lambda (num str) (cons num (cons str '()))))
(define numstr-get-num
  (lambda (x) (car x)))
(define numstr-set-num!
  (lambda (x val) (set-car! x val)))
(define numstr-get-str
  (lambda (x) (car (cdr x))))
(define numstr-set-str!
  (lambda (x) (set-car! (cdr x) val)))
```

- マクロ

```
(define-structure numstr num str)
```

- マクロを使うと問題の記述が簡単になる

クロージャ

```
(define make-predicate
  (lambda (div)
    (lambda (n)
      (= (modulo n div) 0))))

(define fizz? (make-predicate 3))
(define buzz? (make-predicate 5))
```

fizz と buzz の定義

- マクロを使ってみました

```
(define-macro (make-filter test str)
  `(lambda (x)
     (if (,test (numstr-get-num x))
         (make-numstr
          :num (numstr-get-num x)
          :str (string-append
                ,str (numstr-get-str x)))
         x)))

(define fizz (make-filter fizz? "Fizz"))
(define buzz (make-filter buzz? "Buzz"))
```

- クロージャでもいいでしょう

FizzBuzz マクロ版

```
(map fizzbuzz-print
     (map fizz
          (map buzz
               (map num-str-pair
                    (count 1 100)))))
```

```
→(... 67 68 "Fizz" "Buzz" 71 "Fizz"
      73 74 "FizzBuzz" 76 77 "Fizz" ...)
```

FizzBuzzSizz

- 要求は変わる
  - 7の倍数のときは「Sizz」
  - 3 と 7 の両方の倍数のときは 「FizzSizz」
  - 5 と 7 の両方の倍数のときは 「BuzzSizz」
  - 3 と 5 と 7 全部の倍数のときは 「FizzBuzzSizz」
- マクロ版なら対応は簡単

```
(define sizz? (make-predicate 7))
(define sizz (make-filter sizz? "Sizz"))
(map fizzbuzz-print
     (map fizz
          (map buzz
               (map sizz
                    (map num-str-pair
                         (count 1 100))))))
```

→`(... 67 68 "Fizz" "BuzzSizz" 71 "Fizz"`
`73 74 "FizzBuzz" 76 "Sizz" "Fizz"..)`

- さらにマクロを書いてもよい

Lisp の先進性

- 「普通のやつらの上を行け」
  - Paul Graham
    - http://practical-scheme.net/trans/beating-the-averages-j.html

1960年頃にLispによって導入された
ガベージコレクションは、近年では良い技術だと
広く認められるようになった。

実行時型判定も同じくポピュラーになりつつある。

レキシカルクロージャは1970年代はじめに
Lispによって導入されたが、
ようやくレーダーの端に捉えられはじめた。

1960年代中頃にLispが導入したマクロは、
まだ未知の世界だ。

データと関数の区別がない
とはどいうことか？

Lisp のインタープリタ

- Read-Eval-Print-Loop
- Read
  - 文字列を式として読み込む
- Eval
  - 式を評価する
- Print
  - 式を文字列として書き出す

eval を作る

- 「LISP 1.5 Programmer's Manual」の eval
- 制限された Lisp
  - リスト処理はできる
  - 素直な数値計算などはできない
- 作る関数の名前は全部大文字

```
(EVAL
 '(REPL (QUOTE (a b c d)) (QUOTE c) (QUOTE e))
 ENV)
→(a b e d)
```

- Read と Print は Scheme にお任せ

材料

- 5つの関数
  - car
  - cdr
  - cons
  - eq
  - atom
- 2つの特殊形式
  - quote
  - cond
- 関数定義の機能
  - lambda
  - label
    - define のこと

おまじない

- 関数定義の機能を実装

```
(use srfi-1) ; for delete

; シンボルとその値の連想リスト
(define ENV '((#t . #t)
              (#f . #f)
              (() . ())
              (else . #t)))

(define-macro (LABEL func lamb)
  `(begin
     (set! ENV (delete (assoc ',func ENV) ENV))
     (set! ENV (cons (cons ',func ,lamb) ENV))
     (define ,func
       (eval (cons 'lambda (cdr ,lamb))
             (interaction-environment)))))

(define-macro (LAMBDA . body)
  `(cons 'LAMBDA ',body))
```

基本関数

```
(define CAR    car)
(define CDR    cdr)
(define CONS   cons)
(define EQ     eq?)
(define ATOM   (lambda (x) (not (pair? x))))

(define QUOTE  quote)
(define COND   cond)
```

伝統的な補助関数

```
(LABEL CADR  (LAMBDA (x) (CAR (CDR x))))
(LABEL CDAR  (LAMBDA (x) (CDR (CAR x))))
(LABEL CAAR  (LAMBDA (x) (CAR (CAR x))))
(LABEL CADDR (LAMBDA (x) (CAR (CDR (CDR x)))))
(LABEL CADAR (LAMBDA (x) (CAR (CDR (CAR x)))))
```

```
(CADR  '((a b) (c d) (e f))) → (c d)
(CDAR  '((a b) (c d) (e f))) → (b)
(CAAR  '((a b) (c d) (e f))) → a
(CADDR '((a b) (e d) (e f))) → (e f)
(CADAR '((a b) (c d) (e f))) → b
```

本当は (QUOTE ...) と書くべきです。。。

null

- 空リストか？

```
(LABEL NULL
  (LAMBDA (x)
    (COND
     ((EQ x (QUOTE ())) #t)
     (else #f))))

(NULL '()) → #t
(NULL 'symbol) → #f
```

equal

- 同等のリストか？

```
(LABEL EQUAL
  (LAMBDA (x y)
    (COND
     ((ATOM x)
      (COND
       ((ATOM y)
        (EQ x y))
       (else #f)))
     ((EQUAL (CAR x) (CAR y))
      (EQUAL (CDR x) (CDR y)))
     (else #f))))

(EQUAL '(a (b c) d) '(a (b c) d)) → #t
(EQUAL '(a (b c) d) '(a (b e) d)) → #f
```

assoc

- 連想リストの検索

```
(LABEL ASSOC
  (LAMBDA (x a) ; 鍵 連想リスト
    (COND
     ((EQUAL (CAAR a) x)
      (CAR a))
     (else
      (ASSOC x (CDR a))))))

(ASSOC 'foo '((boo straw)
              (foo wood)
              (woo brick))
→(foo wood)
```

eval

```
(LABEL EVAL
  (LAMBDA (e a) ; 式 連想リスト
    (COND
     ((ATOM e) ; シンボルなら
      (CDR (ASSOC e a))) ; 変数名・関数名の検索
     ((ATOM (CAR e)) ; リストの第一要素がシンボルなら
      (COND
       ((EQ (CAR e) (QUOTE QUOTE))
        (CADR e)) ; QUOTE を外す
       ((EQ (CAR e) (QUOTE COND))
        (EVCON (CDR e) a))
       (else
        ; 関数呼び出し
        (APPLY (CAR e) (EVLIS (CDR e) a) a))))
     (else ; リストの第一要素がリストなら
      ; 無名関数呼び出し
      (APPLY (CAR e) (EVLIS (CDR e) a) a)))))
```

もう一度言いますが

表示された関数を
理解しようとは
思わないで下さい！

evcon

- cond の本体を評価

```
(LABEL EVCON
  (LAMBDA (c a) ; 条件＋場合の式 連想リスト
    (COND
     ((EVAL (CAAR c) a)      ; 条件
      (EVAL (CADAR c) a))    ; その場合の式を評価
     (else
      (EVCON (CDR c) a)))))

(EVCON '((#f (QUOTE a))
         (#t (QUOTE b)))
       ENV)
→ b
```

evlis

- 関数の引数の評価

```
(LABEL EVLIS
  (LAMBDA (m a) ; 関数の引数 連想リスト
    (COND
     ((NULL m) (QUOTE ()))
     (else
      (CONS (EVAL (CAR m) a) ; 引数を評価
            (EVLIS (CDR m) a))))))
```

とは違います

関数の評価(1)

```
(LABEL APPLY
  (LAMBDA (fn x a) ; 関数 引数 連想リスト
    (COND
     ((ATOM fn) ; シンボルなら
      (COND
       ((EQ fn (QUOTE CAR))
        (CAAR x))
       ((EQ fn (QUOTE CDR))
        (CDAR x))
       ((EQ fn (QUOTE CONS))
        (CONS (CAR x) (CADR x)))
       ((EQ fn (QUOTE ATOM))
        (ATOM (CAR x)))
       ((EQ fn (QUOTE EQ))
        (EQ (CAR x) (CADR x)))
       (else
        (APPLY
         (EVAL fn a) ; 関数名から関数本体を検索
         x a))))
```

関数の評価(2)

- lambda の評価

  apply の続き -- ここが大切

```
((EQ (CAR fn) (QUOTE LAMBDA))
 (EVAL
  (CADDR fn) ; 関数本体
  (PAIRLIS   ; 引数の登録
   (CADR fn) ; 引数名
   x a)))    ; 評価後の引数 連想リスト
```

```
(CAR   '(LAMBDA (x y) (CONS x y)))
→ LAMBDA

(CADDR '(LAMBDA (x y) (CONS x y)))
→ (CONS x y)

(CADR  '(LAMBDA (x y) (CONS x y)))
→ (x y)
```

pairlis

- 引数名と実際の値の組を連想リストに登録

```
(LABEL PAIRLIS
  (LAMBDA (x y a)
    (COND
     ((NULL x) a)
     (else
      (CONS (CONS (CAR x) (CAR y))
            (PAIRLIS (CDR x) (CDR y) a))))))

(PAIRLIS '(x y) '(1 2) '((#t . #t) (#f . #f)))
→((x . 1) (y . 2) (#t . #t) (#f . #f))
```

cf `((LAMBDA (x y) (CONS x y)) 1 2)`

関数の評価(3)

- label の評価

  apply の続き

```
((EQ (CAR fn) (QUOTE LABEL))
 (APPLY
  (CADDR fn) ; (LAMBDA 関数本体)
  x
  (CONS ; 関数の登録
   (CONS
    (CADR fn)   ; 関数名
    (CADDR fn)) ; (LAMBDA 関数本体)
   a))))))
```

使ってみよう

```
(LABEL REPL
  (LAMBDA (lst old new)
    (COND
      ((NULL lst) (QUOTE ()))
      ((EQ (CAR lst) old)
       (CONS new (REPL (CDR lst) old new)))
      (else
       (CONS (CAR lst)
       (REPL (CDR lst) old new))))))

(EVAL
 '(REPL (QUOTE (a b c d)) (QUOTE c) (QUOTE e))
 ENV)
→(a b e d)
```

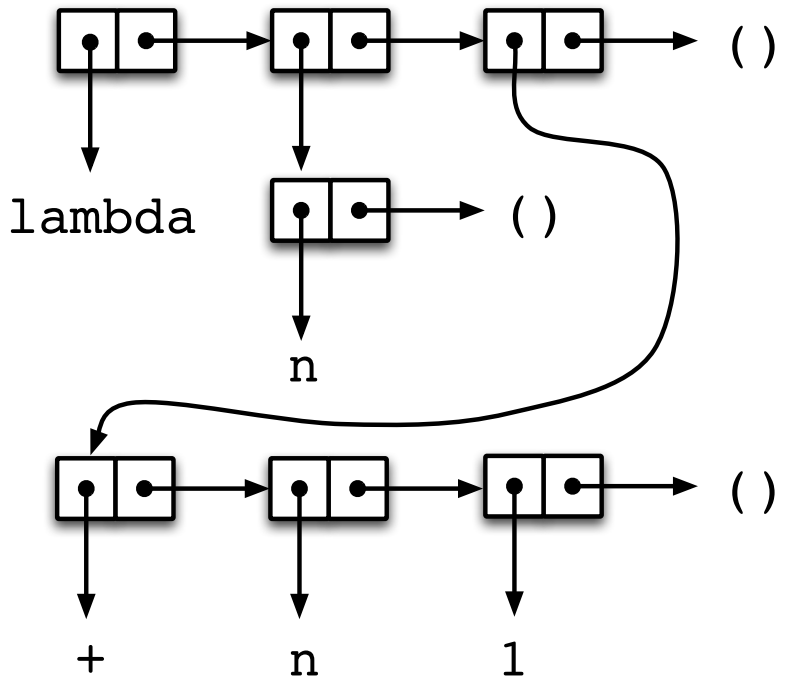
「The Little Schemer」の10章は
もっとちゃんとした
Scheme のインタープリタを
作る話です

evcon & evlis も出てきます :-)

関数も単なるデータ

- 第一要素がシンボル lambda であるリスト

```
(lambda (n) (+ n 1))
```



Lisp の名人

- 「名人伝」
  - 中島 敦

既に、我と彼との別、
是と非との分を知らぬ。
眼は耳の如く、
耳は鼻の如く、
鼻は口の如く思われる

セルはλの如く　λはセルの如く