ソフトウェアアーキテクチャ
第6回　LISP処理系

環境情報学部
萩野　達也

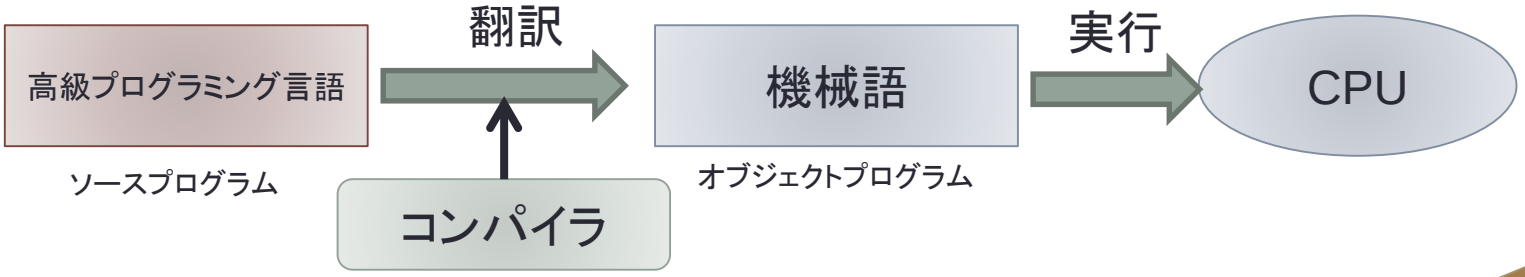
lecture URL

https://vu5.sfc.keio.ac.jp/slide/

コンパイラとインタープリタ

事前翻訳

- コンパイラ方式（compiler）
  - プログラムを機械語に変換して実行
  - プログラムを機械語に変換（翻訳）するプログラムをコンパイラ

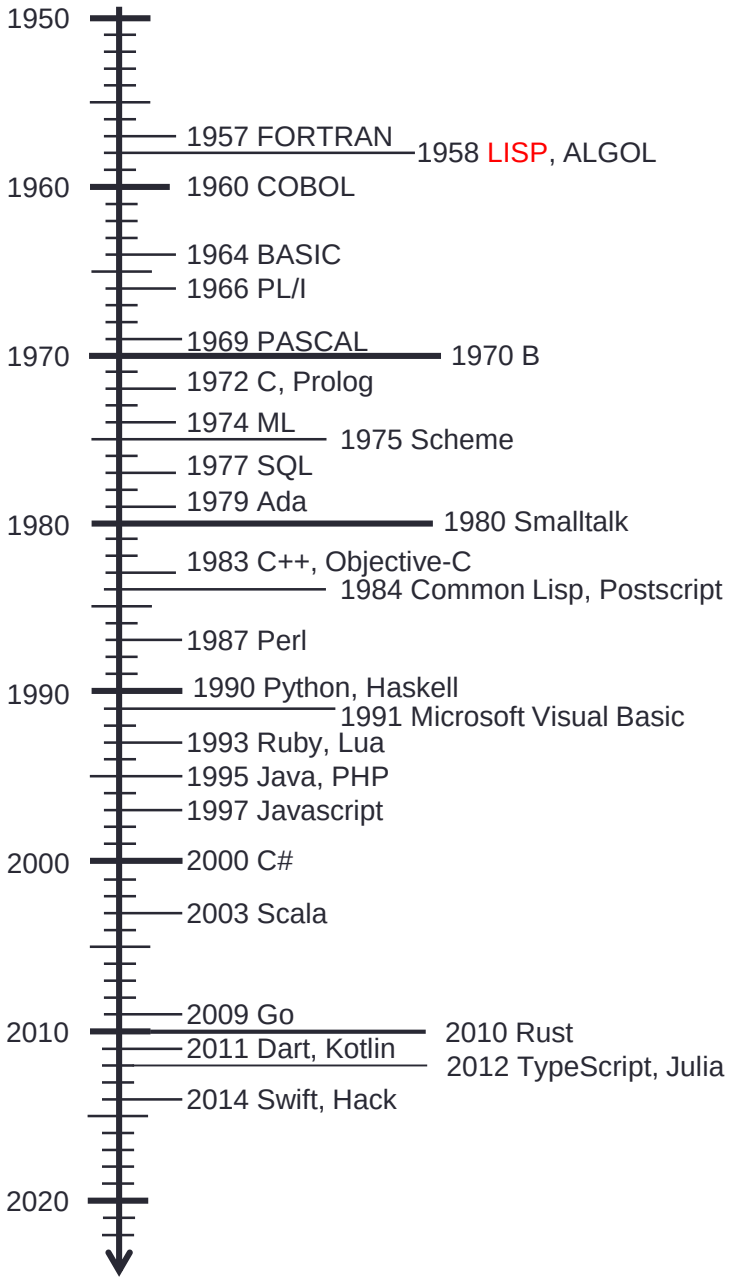


同時通訳

- インタープリタ方式（interpreter）
  - プログラムを変換や翻訳せずに，そのまま直接実行
  - コンパイラに比べて実行速度は落ちる
  - プログラムを変更してから実行までは早い

LISPとは

- LISP（LIST Processing）
  - John McCarthyによって考案
  - 2番目に古いプログラミング言語
    - 一番古いのはFORTRAN
- 特徴
  - ラムダ計算を実装した言語
  - 記号処理向き
    - FORTRANは数値計算向き
  - 人工知能的な分野で多く利用
- 多くの方言が存在
  - MACLisp
  - Common Lisp
  - Emacs Lisp
  - Scheme



プログラミング言語の種類

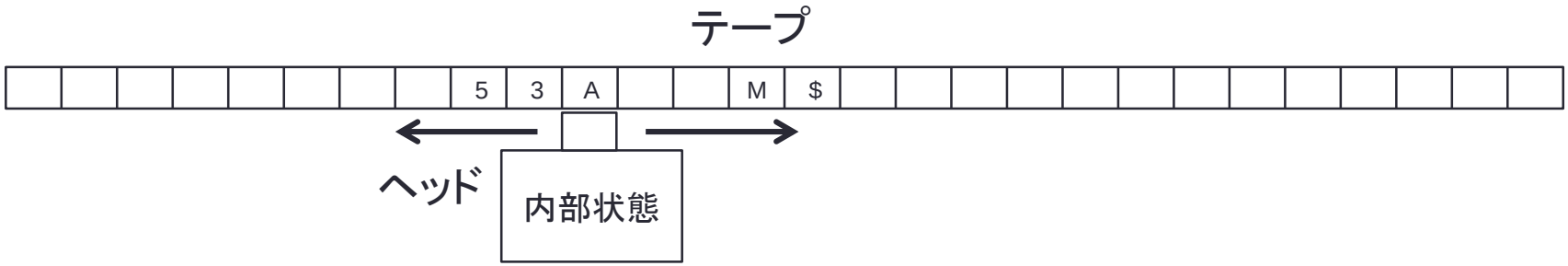
- 数値計算が目的
  - FORTRAN
- 事務処理が目的
  - COBOL
- オペレーティングシステムの記述
  - C言語
- 記号処理や人工知能処理
  - LISP
- Webプログラミング
  - JavaScript
  - PHP
- スクリプト
  - シェルスクリプト
  - perl
  - ruby
  - python
- 文字列処理
  - SNOBOL
- シュミレーション
  - SIMULA
- 機器への組み込み目的
  - Java
- 教育用
  - PASCAL

プログラミング言語のパラダイム

- 手続き型（procedural）
  - プログラムは命令の列
  - 書かれている順番に処理をしていく
  - ほとんどのプログラミング言語が手続き型
  - フォンノイマン型コンピュータのストアドプログラミングの原理と合う
- 関数型（functional）
  - プログラムは関数
  - 関数を組み合わせることで処理を行う
  - 計算の順序は関係ない
  - 数学での問題解決に近い
- 論理型（logical）
  - 論理式を書くことでプログラミングを行う
  - 規則を書き並べるだけ
  - 規則の適用順は関係ないことが多い

計算機のモデル

- チューリング機械
  - 1936年イギリスの数学者Alan Turingの論文「計算可能数について ― 決定問題への応用」で発表
  - 無限のセルで区切られた**テープ**を持つ
  - テープ上のセルの内容を読み書きする**ヘッド**がある
  - 一度に一つのセルの内容しか読み書きできない
  - ヘッドは有限の状態を持つ
  - セルの内容とヘッドの内部状態によって，セルの内容を書き換え，ヘッドを右か左に移動させることができる

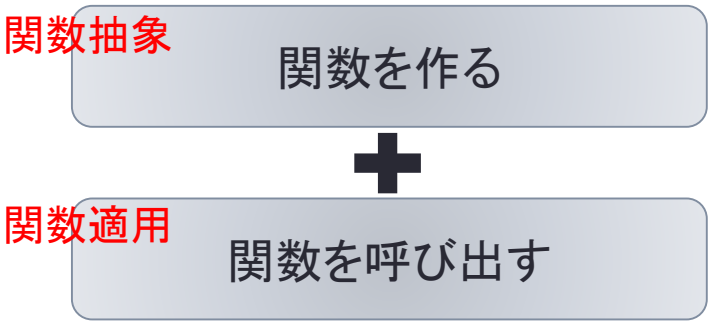


計算機のモデル

- 帰納的関数
  - 原始帰納的関数にμ作用素を追加したもの
  - 原始帰納法
    - 0! = 1
    - (n+1)! = (n + 1) × n!
  - 原始帰納法では書けない計算可能な関数があり，μ作用素が必要
    - Ack(0, n) = n + 1
    - Ack(m, 0) = Ack(m - 1, 1)
    - Ack(m, n) = Ack(m - 1, Ack(m, n - 1))

- レジスタ機械
  - 有限個のレジスタ（ただし無限の大きさの値を格納可能）を持つ単純な計算機

- ラムダ計算
  - 関数を抽象化したもの
  - 関数抽象と関数適用
  - αβ変換を計算の過程と考える



ラムダ式

- 数学では関数に名前を付けて定義し，利用する
  - $f(x) = x + 5$
  - $f(3)$
- ラムダ計算では名前を付けずに関数を記述する
  - $f = \lambda x.x + 5$
  - $f(3) = (\lambda x.x + 5)(3)$

```
<λ式> ::= <変数> | '(λ' <変数> '.' <λ式> ')' | '(' <λ式> <λ式> ')'
```

- 例
  - $(\lambda x.x)$
  - $((\lambda x.(x\ y))(\lambda z.z))$
  - $((\lambda x.(x\ x))(\lambda x.(x\ x))$

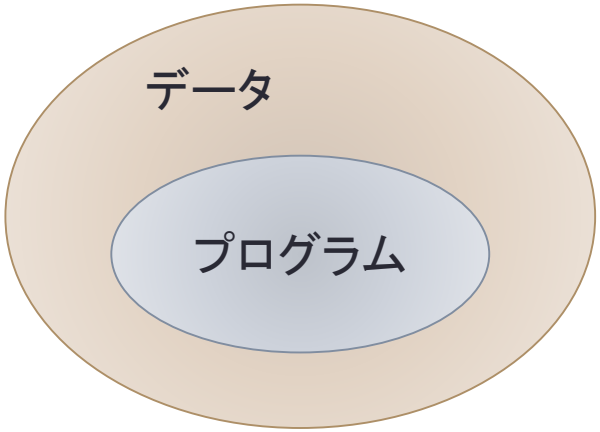
ラムダ計算

- 計算規則は関数適用のみ（β変換）
  - $((\lambda x.M)N) \rightarrow M[N/x]$
- 例
  - $(\lambda x.x)y \rightarrow y$
  - $(\lambda x.xx)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow \lambda y.y$
- 自然数もλ式で表す
  - $\mathbf{0} \equiv \lambda xy.y$
  - $\mathbf{1} \equiv \lambda xy.xy$
  - $\mathbf{2} \equiv \lambda xy.x(xy)$
  - $\mathbf{3} \equiv \lambda xy.x(x(xy))$
  - $\ldots$
- 四則演算もλ式で表すことができる
  - $plus \equiv \lambda xyzw.(xz)((yz)w)$

$$plus\ \mathbf{1}\ \mathbf{2} \equiv \left(\lambda xyzw.(xz)\big((yz)w\big)\right)(\lambda xy.xy)\big(\lambda xy.x(xy)\big) \rightarrow \cdots \rightarrow \lambda xy.x(x(xy))$$

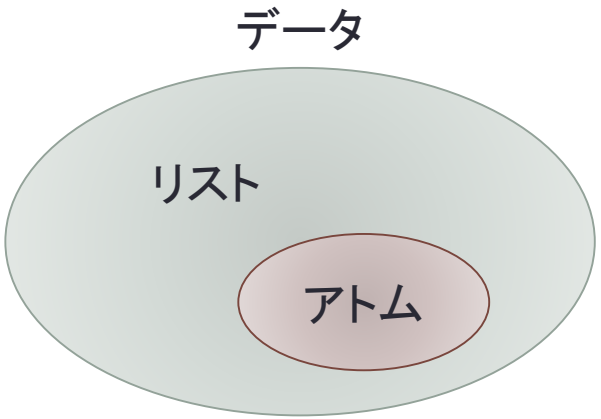
LISPオブジェクト

- データとプログラムの区別がない
  - プログラムもデータである
  - フォンノイマン計算機　＝　ストアドプログラム



- データ
  - アトム(atom)
  - リスト(list)

- アトム
  - 数値: 0, 123, 3.14
  - 文字列: "abc", "Hello World!"
  - シンボル: x, abc, hello

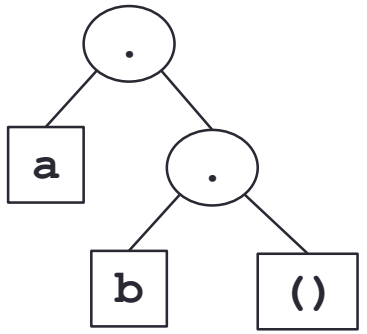


リスト

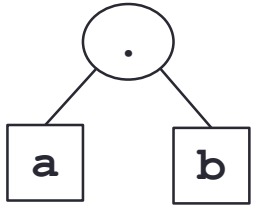
- リスト（あるいはS 式）

```
<リスト> ::= '()' | <アトム> | '(' <リスト> '.' <リスト> ')'
```

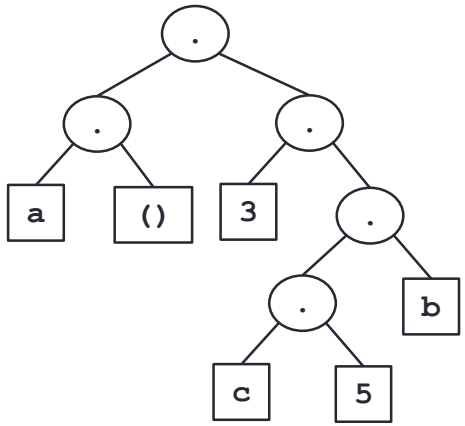
- () は空のリスト，nil とも書く



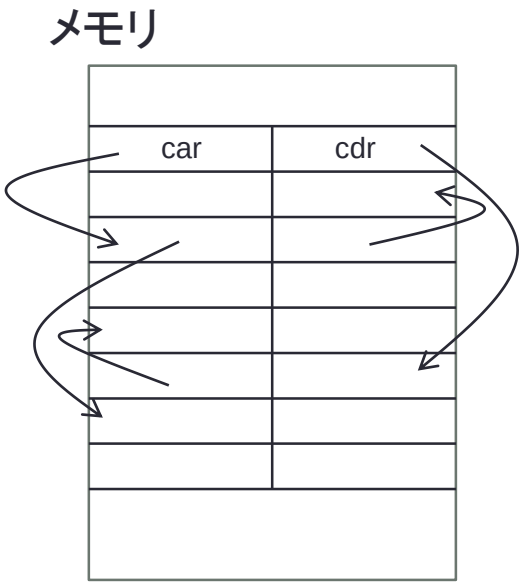
- リストはアトムを葉にもつ2分木



- 例
  - (a . b)
  - (a . (b . ()))
  - ((a . ()) . (3 . ((c . 5) . b)))

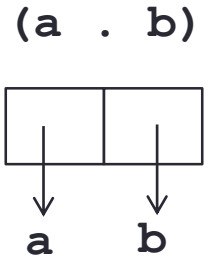


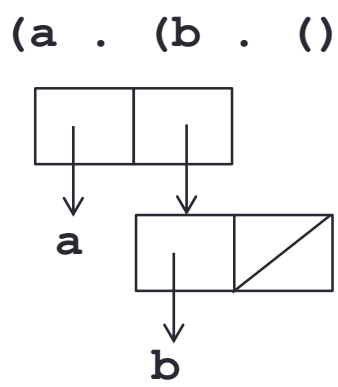
リストの実装

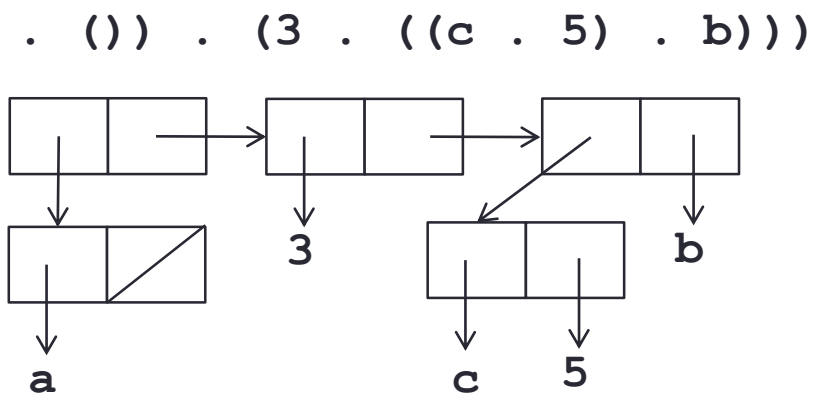


- Cons cell

- consセルはメモリ上の番地のペア
- carとcdrはアトムあるいはconsセルの番地を保持

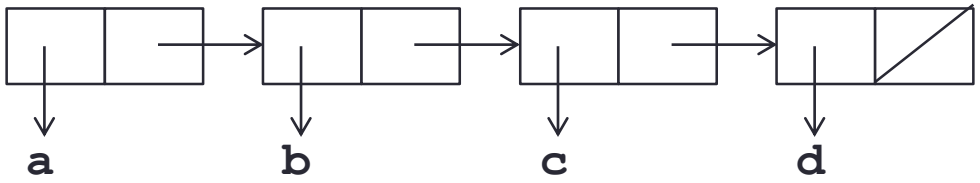




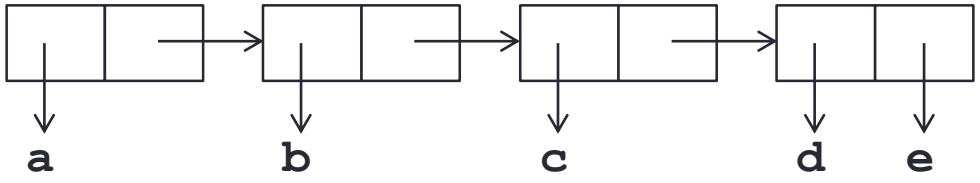


リストの省略形

- (a b c d)は次のS式の省略形
  - (a . (b . (c . (d . ()))))



- (a b c d . e)は次のS式の省略形
  - (a . (b . (c . (d . e))))



式と評価

- LISP のプログラム
  - 与えられたS式を評価すること
  - if文やwhile文などの文はない
- 式は関数と引数をリストにしたもの
  - `(plus 1 2)`
  - `(times (plus 1 2) (plus 2 3))`

$f(x)$

数学

`(f x)`

LISP

中置記法 `1 + 2`

前置記法（ポーランド記法） `+ 1 2`

後置記法（逆ポーランド記法） `1 2 +`

式の評価

- LISP のプログラム
  - 与えられたS式を評価すること
  - 式を単純化していく

```
(plus 1 2)  ⇒  3
```

```
(times (plus 1 2) (plus 2 3))  ⇒  (times 3 (plus 2 3))
                                          ⇓
                                  (times 3 5)  ⇒  15
```

基本関数(1)

- **quote**: 引数そのものが値

```
(quote (a b)) => (a b)
```

  - 省略記法では**quote**を’で表す

```
'(a b) => (quote (a b)) => (a b)
```

- **atom**: アトムであるかどうかを調べ, t またはnil を返す

```
(atom 'abc) => t
(atom '(a b)) => nil
```

- **eq**: 同じオブジェクト(アトムまたはリスト)を指し示すかを調べる

```
(eq 'a 'a) => t
(eq 'a 'b) => nil
(eq '(a b) '(a b)) => nil
```

  - 単純にメモリ上の番地の比較
  - アトムおよび数値は常にユニーク

基本関数(2)

- **cons**: コンスセルを作る

```
(cons 'a 'b) => (a . b)
(cons 'a '(b c)) => (a b c)
```

- **car**: コンスセルの左を返す

```
(car '(a b)) => a
```

- **cdr**: コンスセルの右を返す

```
(cdr '(a b)) => (b)
(cdr '(a b c)) => (b c)
```

- **cond**: 条件にしたがって異なる値を返す

```
(cond ((eq x 'a) 'yes)
      ((eq x 'b) 'no)
      ('t 'unknown))
```

関数の定義

- ラムダ記法

```
(lambda (x y) (cons x (cons y '())))
```

  - 引数: x, y 本体: `(cons x (cons y '()))`

- 関数適用

```
((lambda (x y) (cons x (cons y '()))) 1 2)
   => (1 2)
```

  - 引数に値を束縛して本体の式を評価

- 名前付け

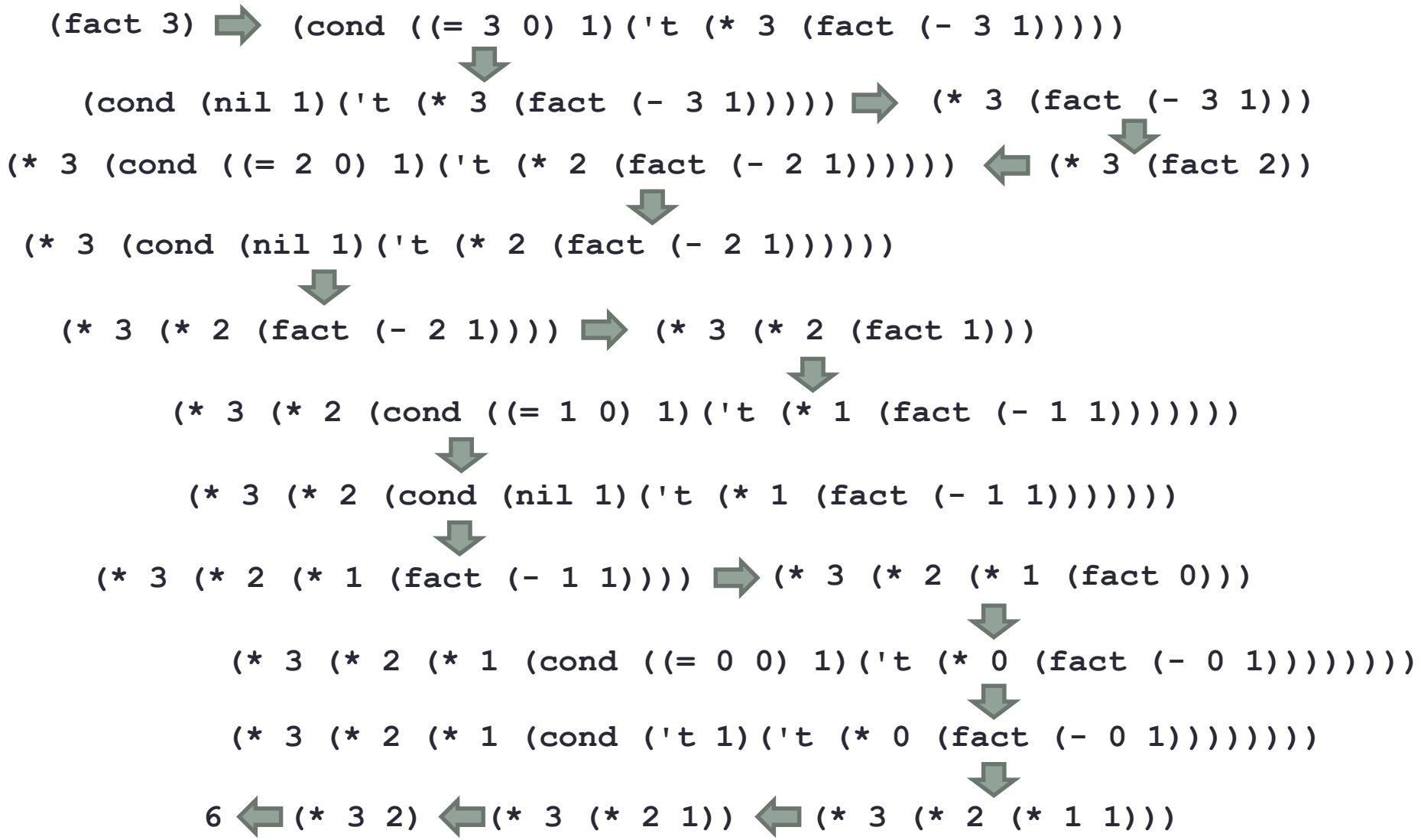
```
(define fact
   (lambda (x)
      (cond ((= x 0) 1)
            ('t (* x (fact (- x 1)))))))
```

  - 再帰呼び出しにより繰り返しを実現

```
(defun fact (x)
   (cond ((= x 0) 1)
         ('t (* x (fact (- x 1))))))
```

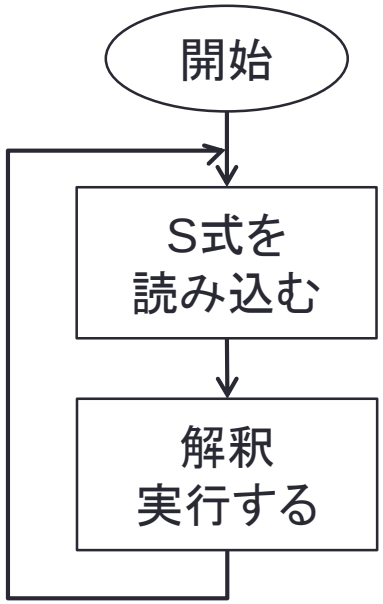
関数の評価

```
(define fact
   (lambda (x)
      (cond ((= x 0) 1)
            ('t (* x (fact (- x 1)))))))
```



LISPインタープリタ

- インタープリタはプログラムを直接解釈実行する



S式を読み込みリストにする

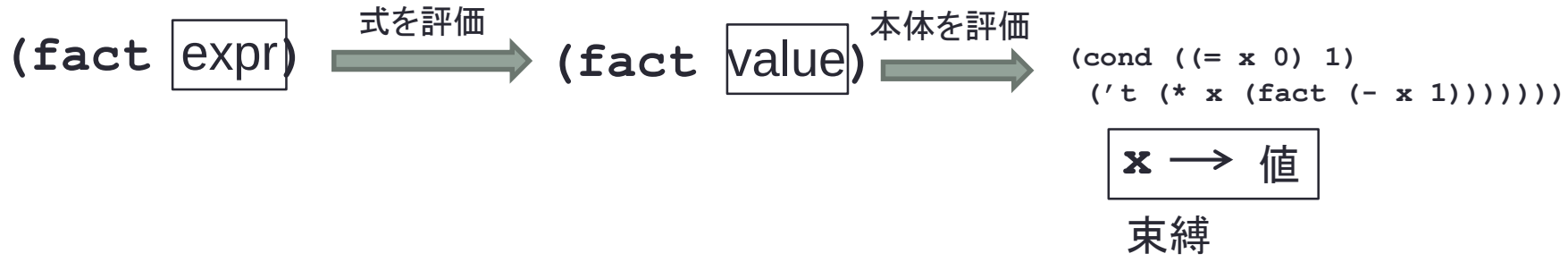
S式を簡略化・関数の適用を繰り返す

解釈実行

- 基本関数（組み込み関数）の場合
  - 引数を評価して値にする
  - 基本関数を実行する

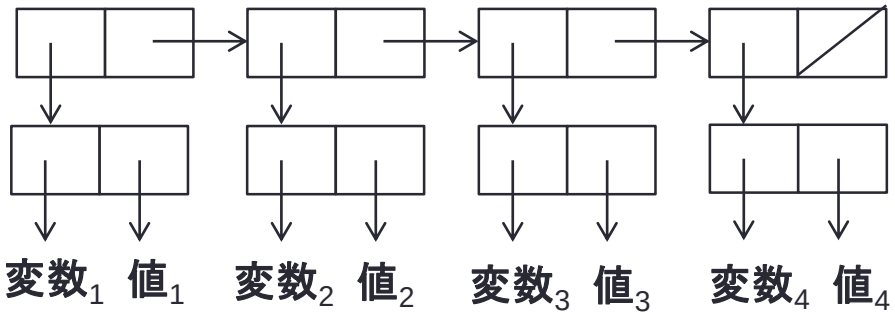
- ユーザ定義の関数の場合
  - 引数を評価して値にする
  - 仮引数の変数に値をバインドして関数本体を評価する

```
(define fact
   (lambda (x)
      (cond ((= x 0) 1)
            ('t (* x (fact (- x 1)))))))
```

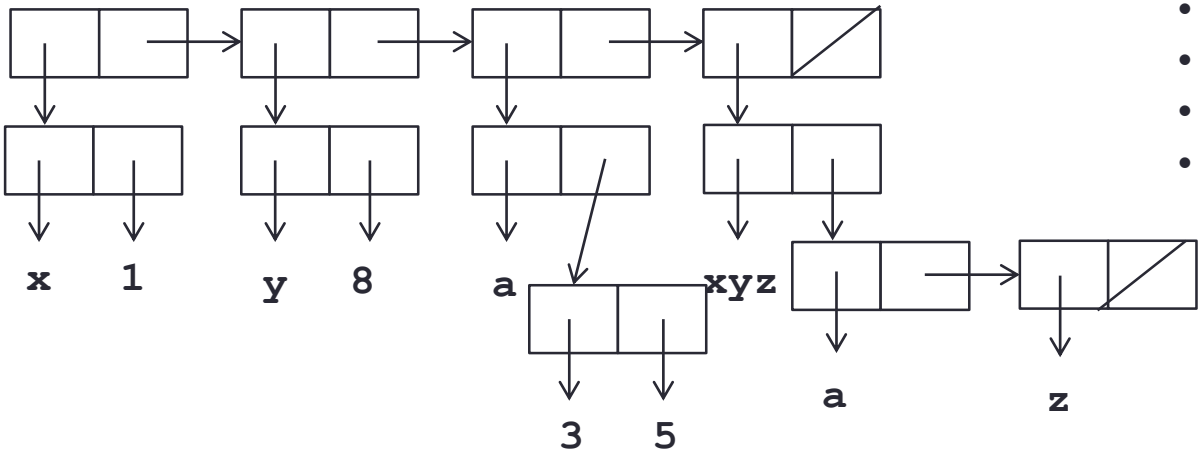


連想配列

- 式は変数が値に束縛された状態で評価される
  - 変数が値に束縛されている状態を環境と呼ぶ
  - 環境は，変数の値を保持する
  - 連想リスト
    - 連想メモリ
    - 名前添え字の配列



- 例



- xの値は1
- yの値は1
- aの値は(3 . 5)
- xyzの値は(a z)

LISPインタープリタ in LISP（1）

```
(define null (lambda (x) (eq x nil)))

(define and (lambda (x y) (cond (x (cond (y 't) ('t nil))) ('t nil))))

(define not (lambda (x) (cond (x nil)('t 't))))

(define append (lambda (x y)
  (cond ((null x) y)
  ('t (cons (car x) (append (cdr x) y))))))

(define list (lambda (x y) (cons x (cons y nil))))

(define pair (lambda (x y)
  (cond ((and (null x) (null y)) nil)
        ((and (not (atom x)) (not (atom y)))
         (cons (list (car x) (car y)) (pair (cdr x) (cdr y)))))))

(define assoc (lambda (x y)
  (cond ((eq (caar y) x) (cadar y))
        ('t (assoc x (cdr y))))))

(define caar (lambda (x) (car (car x))))
(define cadr (lambda (x) (car (cdr x))))
(define cadar (lambda (x) (cadr (car x))))
(define caddr (lambda (x) (cadr (cdr x))))
(define caddar (lambda (x) (caddr (car x))))
```

LISPインタープリタ in LISP(2)

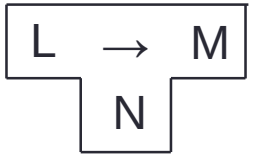
```
(define evcon (lambda (c a)
  (cond ((eval (caar c) a) (eval (cadar c) a))
        ('t (evcon (cdr c) a)))))

(define evlis (lambda (m a)
  (cond ((null m) nil)
        ('t (cons (eval (car m) a) (evlis (cdr m) a))))))

(define eval (lambda (e a)
  (cond ((atom e) (assoc e a))
        ((atom (car e))
         (cond ((eq (car e) 'quote) (cadr e))
               ((eq (car e) 'atom) (atom (eval (cadr e) a)))
               ((eq (car e) 'eq) (eq (eval (cadr e) a) (eval (caddr e) a)))
               ((eq (car e) 'car)(car (eval (cadr e) a)))
               ((eq (car e) 'cdr) (cdr (eval (cadr e) a)))
               ((eq (car e) 'cons) (cons (eval (cadr e) a) (eval (caddr e) a)))
               ((eq (car e) 'cond) (evcon (cdr e) a))
               ('t (eval (cons (assoc (car e) a) (cdr e)) a))))
        ((eq (caar e) 'label) (eval (cons (caddar e) (cdr e))
                                    (cons (list (cadar e) (car e)) a)))
        ((eq (caar e) 'lambda)
         (eval (caddar e) (append (pair (cadar e) (evlis (cdr e) a)) a))))))
```

LISP in LISP

- T図式

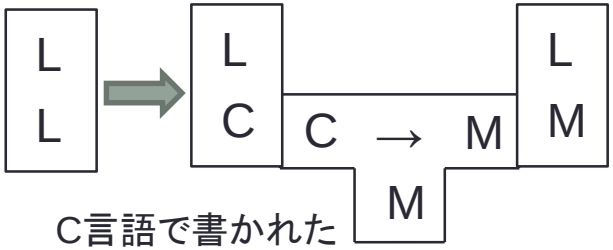


N言語で書かれた
言語Lから言語M
へのコンパイラ

N言語で書かれた
言語Lのインタープリタ

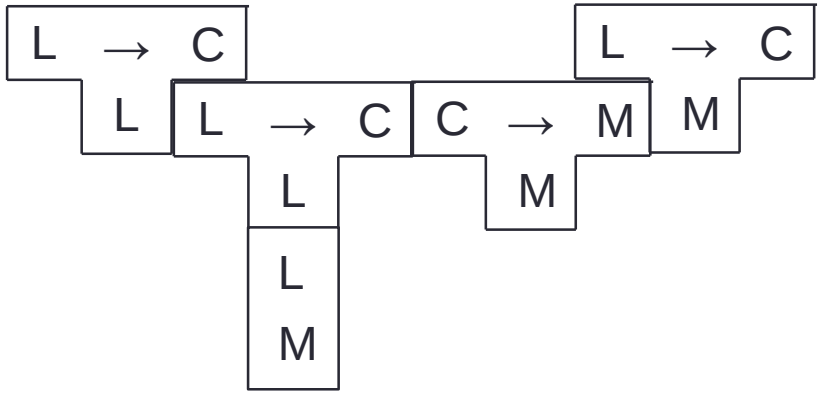
- LISPの仕組みをLISPで説明
  - **eval**が評価する関数
  - **(eval e a)**: 状態**a**(変数へのバインド)で式**a**を評価

LISPで書かれた
LISPインタープリタ



C言語で書かれた
LISPインタープリタ

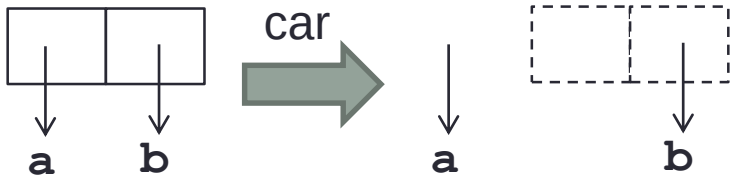
LISPインタープリタを拡張



LISPで書かれた
LISPからCへのコンパイラ

ガーベジコレクション

- ゴミの発生
  - **cons**がコンスセルを消費
  - 参照されないコンスセルはゴミ
  - **(car (cons 'a 'b))**



- ゴミを回収する必要がある
  - ガーベジコレクション（ごみ回収, GC）

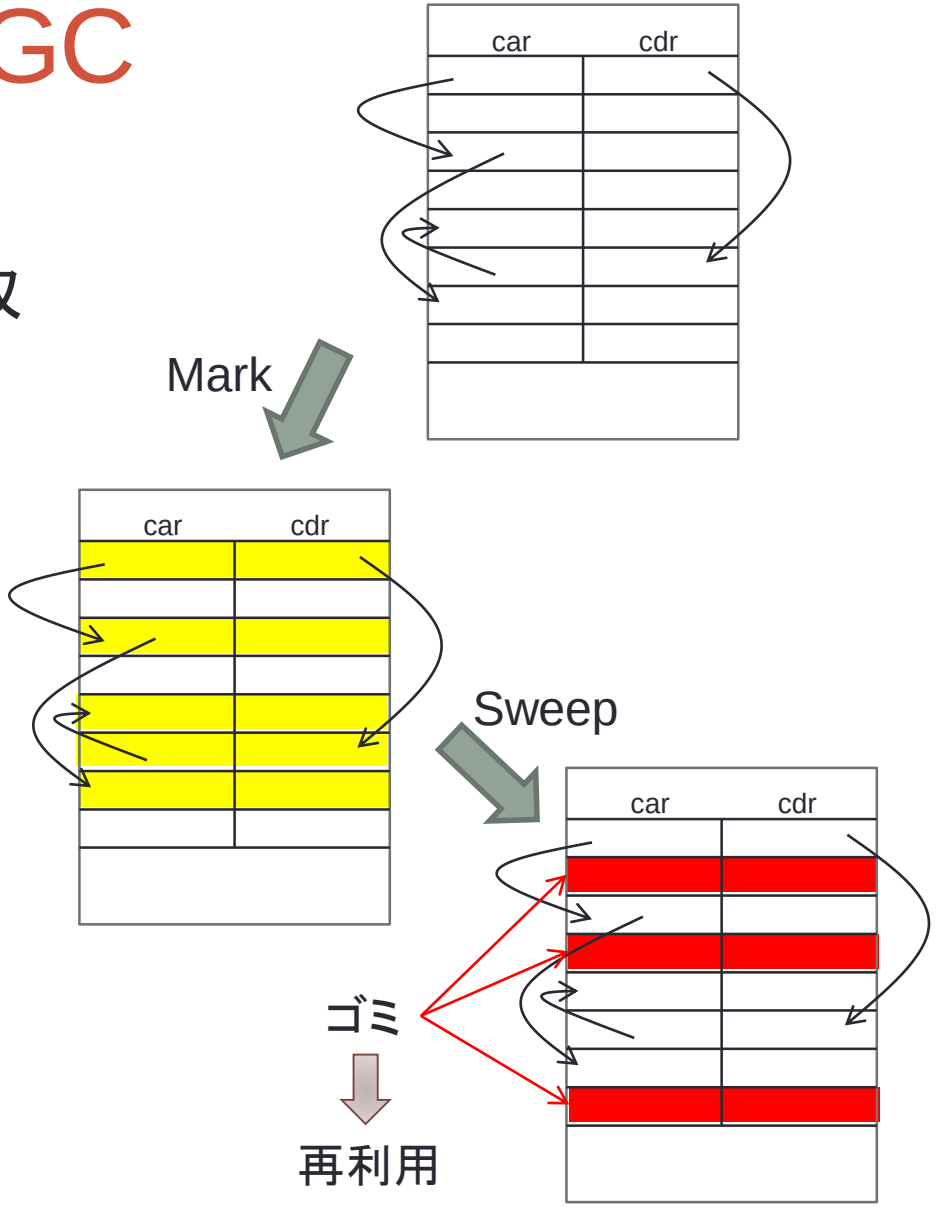
- 色々なガーベジコレクション
  - mark and sweep
  - compaction GC
  - copy GC
  - reference GC
  - realtime GC

Mark and Sweep GC

- 使っているセルに印を付ける
- 印の付いていないセルを回収

```
struct cons {
   struct cons *car;
   struct cons *cdr;
   int marked;
} cell[N], *free;

mark(struct cons *x) {
   if (x->marked) return;
   x->marked = 1;
   mark(x->car);
   mark(x->cdr);
}
sweek() {
   free = 0;
   for (i = 0; i < N; i++) {
      if (!cell[i].marked) {
         cell[i].car = free;
         free = &cell[i];
      }
   }
}
```

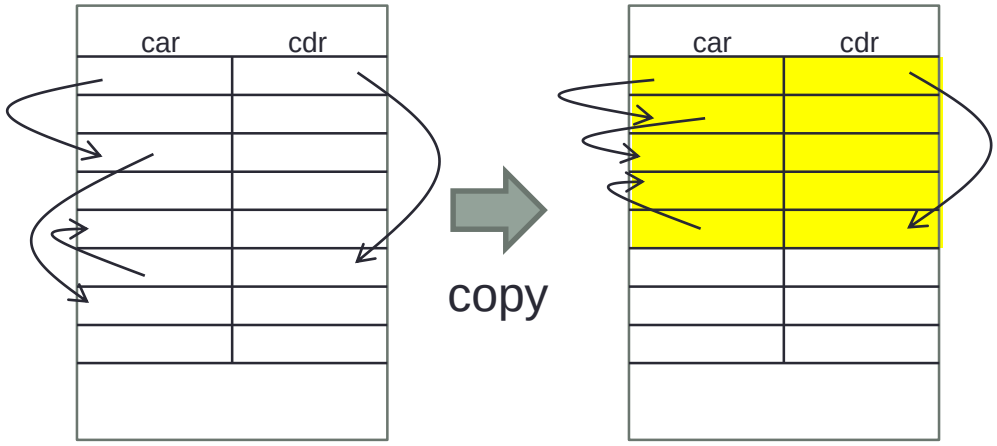


Copying GC

- 使っているセルを別のところにコピーする
- 領域が2倍必要

```
struct cons {
   struct cons *car;
   struct cons *cdr;
   int copied;
};

copy(struct cons *x) {
   struct cons *y, *car, *cdr;
   if (x->copied) return x->car;
   car = x->car;
   cdr = x->cdr;
   x->car = y = new();
   x->copied = 1;
   y->car = copy(car);
   y->cdr = copy(cdr);
   return y;
}
```



その他のGC

- Compaction GC
  - ゴミとなったスペースを詰めていく
  - 時間がかかる
- Reference Count
  - それぞれのセルに参照カウンタを付ける
  - 参照カウンタがゼロになったときに回収
  - サイクルのあるリストは回収できない
- Realtime GC
  - GC 中に処理が止まらないようにする
  - バックグラウンドでGC を処理

まとめ

- LISP
  - 型無しラムダ計算
  - 基本関数
  - ラムダ式
  - LISP インタープリタ
  - ガーベジコレクション
- Haskell
  - 型付きラムダ計算
  - 遅延評価
  - モナド