アルゴリズムとデータ構造入門

2.データによる抽象の構築

2.2.3 並び　2.3 記号データ

奥 乃 博
大学院情報学研究科 知能情報学専攻
知能メディア講座 音声メディア分野
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/07/IntroAlgDs/
okuno@i.kyoto-u.ac.jp

11月27日・本日のメニュー

データによる抽象化

- 2.2.1 Representing Sequences（並び）
- 2.2.2 Hierarchical Structures
- **2.2.3 Sequences as Conventional Interfaces**
- Exercises 2.33~43
- 2.3 Symbolic Data

来週は必修課題の基礎となる図形言語の説明をするので必ず出席すること

祝 無事終了

京都大學
１１月祭

- 後5回 しっかり概念を叩き込んでください
- 学士力＝学修成果重視
- 「何ができるようになるか」

11月27日・本日のメニュー

データによる抽象化

- 復習と補足
- **2.2.1 Representing Sequences（並び）**
- 2.2.2 Hierarchical Structures
- 2.2.3 Sequences as Conventional Interfaces
- Exercises 2.33~43
- 2.3 Symbolic Data

6

数・cons (リスト) の生成関数

データ型	単位元	生成関数
自然数	`0` （零）	`successor`
リスト	`nil`, `()` （空リスト）	`cons`
文字列	`""` （空文字列）	`string-append`

Closure property（閉包性）**of 自然数, cons, 文字列**

データ型	単位元	演算	終了チェック
数	`0`	＋	`zero?`
数	`1`	×	`zero?`
リスト	`nil`	`cons, car, cdr`	`null?`
文字列	`""`	`string-append`	`null-string?`

TUT 無, MIT Scheme有 7

The Fourth Commandment

Always change at least one argument while recurring.
When recurring on a list of atoms, `lat`, use `(cdr lat)`.
When recurring on a number, `n`, use `(sub1 n)`.

教科書では `(- n 1)`

When recurring on a list of S-expressions, `l`, use `(car l)` and `(cdr l)`, if neither `(null? l)` and `(atom? (car l))` are true.

教科書では `(not (pair? (car l)))`

It must be changed to be closer to termination. The changing argument must be tested in the termination condition:

- when using `cdr`, test termination with `null?` and
- when using `sub1`, test termination with `zero?`.

8

The Fifth Commandment

When building a value with `+`, always use `0` for the value of the terminating line, for adding `0` does not change the value of an addition.

When building a value with `*`, always use `1` for the value of the terminating line, for multiplying by `1` does not change the value of a multiplication.

When building a value with `cons`, always consider `()` for the value of terminating line.

9

Lisp/Scheme Programming十戒

6. **The Sixth Commandment**
   Simplify only after the function is correct.
7. **The Seventh Commandment**
   Recur on the *subparts* that are of the same nature
   - On the sublists of a list.
   - On the subexpressions of an arithmetic expression.
8. **The Eighth Commandment**
   Use help function to abstract from representations.
9. **The Ninth Commandment**
   Abstract common patterns with a new function.
10. **The Tenth Commandment**
    Build functions to collect more than one value at a time.

10

Lisp/Scheme Programming十戒

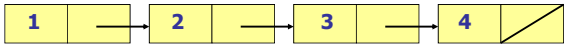
1. **The First Commandment**
   Always ask `null?` as the first question in expressing any function.
2. **The Second Commandment**
   Use `cons` to build lists.
3. **The Third Commandment**
   When building a list, describe the first typical element, and then `cons` it onto the natural recursion.

(Friedman, et al. "*The Little Schemer*", MIT Press)

11

List-ref by cdring down

- (list-ref items n)
  if n=0, list-ref is car
  otherwise, (n-1)st item



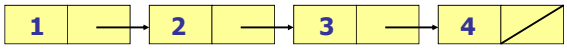

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1) )))
```

- cdring down the list (cdr down)
- Tail recursion に注意（自動的に iteration に変換）

12

length by cdring down

- (length items)
  if items is null?, length is 0
  otherwise, 1 + length of the rest




```
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items)) ))
```

- (+ (length (cdr items)) 1) との違い！

13

append by consing up while cdring down

- (append list1 list2)
  if list1 is null?, append is list2
  otherwise, cons the 1st item of list1 and append of the rest of list1 and list2

```
(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1)
            (append (cdr list1) list2)
      )))
```

14

reverse by consing up while cdring down

- (reverse items)
  if items is null?, reverse is nil
  otherwise, append reverse of the rest of items and list of the 1st item of items

```
(define (reverse items)
  (if (null? items)
      nil
      (append (reverse (cdr items))
              (list (car items)) )))
```

15

Apply transformation to each element

```
(define (scale-list items factor)
   (if (null? items)
       nil
       (cons (* (car items) factor)
             (scale-list (cdr items) factor) )))
```

⇓

```
(define (map proc items)
   (if (null? items)
       nil
       (cons (proc (car items))
             (map proc (cdr items))
       )))
```

```
(map abs (list -10 2.5 -11.6 17))
```

⇒ (10 2.5 11.6 17)

```
(define (scale-list items factor)
  (map (lambda (x) (* x factor)) items) ))
```

16

11月27日・本日のメニュー

データによる抽象化

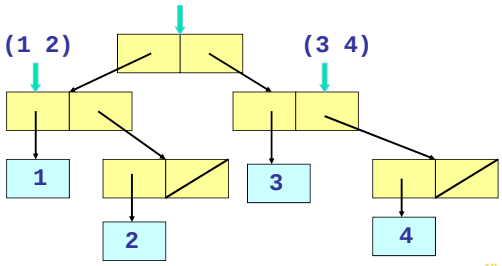
- 2.2.1 Representing Sequences（並び）
- **2.2.2 Hierarchical Structures**
- 2.2.3 Sequences as Conventional Interfaces
- Exercises 2.33~43
- 2.3 Symbolic Data

17

2.2.2 Hierarchical Structures

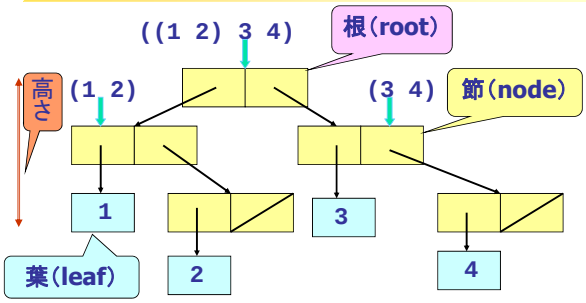
- Tree（木）と捉えると

```
(cons (list 1 2) (list 3 4))
```



18

木の定義



高さ(height)：rootからnodeまでのリンク数
木の高さ:leafの高さの最大値

19

count-leaves

(count-leaves x)
If x is null?, count-leaves is 0
else if x is a leaf (not pair?), count-leaves is 1
otherwise add count-leaves of the car of x and
count-leaves of the cdr of x

```
(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                 (count-leaves (cdr x))
              ))))
```

21

max-height

(max-height x)
If x is null?, max-height is 0
else if x is a leaf (not pair?), max-height is 1
otherwise add 1 to max of
max-height of the car of x and
max-height of the cdr of x

```
(define (max-height x)
   (cond ((null? x) 0)
         ((not (pair? x)) 1)
         (else
           (+ 1 (max (max-height (car x))
                     (max-height (cdr x))
                )))))
```

22

木の写像(leafに倍数をかける)

```
(define (scale-tree tree factor)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (* tree factor))
        (else
          (cons (scale-tree (car tree) factor)
                (scale-tree (cdr tree) factor)
          ))))
```

木をたどって手続きを適用するmapを使用すると:

```
(define (scale-tree tree factor)
  (map
    (lambda (sub-tree)
       (if (pair? sub-tree)
           (scale-tree sub-tree factor)
           (* sub-tree factor) ))
    tree ))
```

23

Ex2.32 powerset

- A = (1 2 3)
- 2^A = (() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))

```
(define (powerset a)
  (if (null? a)
      (list nil)
      (let ((rest (powerset (cdr a)))
        (append
           rest
           (map
             (lambda (x)
               (append (list (car a)) x))
             rest )))))
```

24

11月27日・本日のメニュー

データによる抽象化

- 2.2.1 Representing Sequences（並び）
- 2.2.2 Hierarchical Structures
- 2.2.3 Sequences as Conventional Interfaces
- **Intermission**
- Exercises 2.33~43
- 2.3 Symbolic Data

27

Bugを最初に見つけたのは

- **Grace Murray Hopper (Dec. 9, 1906 – Jan. 1, 1992)**
- **1st women to receive a Ph.D in mathematics.**
- **Sep. 9, 1945. Mark II の Relay #70, Panel F で発見。**
- **bug はそれ以前から使用されていた。**
- **コンパイラ開発にも従事。**
- **NavyでCOBOL言語の検証ソフト開発**

29

A Bug (moth) in the Mark-II

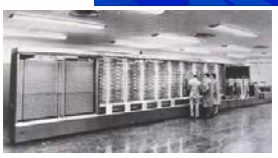
Photo #: NH 96566-KN (Color)

U.S. Naval Historical Center Photograph.

実験ノートをつけること。ルーズリーフはだめ

The Harvard Mark-I

Grace M. Hopper working on the Harvard Mark-I, developed by IBM and Howard Aiken. The Mark-I remained in use at Harvard until 1959, even though other machines had surpassed it in performance, providing vital calculations for the navy in World War II.



10大Bugの3個（by Wired News）

1. 1962年7月22日　火星探査機『マリナー1号』：マリナー1号は打ち上げ時に予定のコースを外れたが、これは飛行ソフトウェアのバグが原因だった。地上の管制センターは大西洋上でロケットを破壊した。事後 調査により、鉛筆で紙に書かれた数式（$\bar{R}_n$）をコンピューターのコードに置き換えるときにミス（－を忘れた）が起き、これが原因でコンピューターが飛行コースの計算を誤ったこと が判明した。
2. 1982年　旧ソ連のガス・パイプライン：シベリアを横断するガス・パイプラインの管理に旧ソ連が購入したカナダ製のコンピューターシステムに、米中央情報局（CIA）のスパイがバグを仕掛けたことがあるという。旧ソ連は当時、米国の機密技術を密かに購入しようと――または盗み出そうと――しており、このシステムを入手したのもその一環だった。だが、計画を察知したCIAはこれを逆手にとり、旧ソ連の検査は問題なく通過するが、いったん運転に入ると機能しなくなるように仕組んだとされる。この結果起きたパイプライン事故は、核爆発以外では地球の歴史でも最大規模の爆発だったという。
3. 1985～1987年　セラック25：複数の医療施設で放射線治療装置が誤作動し、過大な放射線を浴びた患者に死傷者が出た。セラック25は2 種類の放射線――低エネルギーの電子ビーム（ベータ粒子）とX線――を照射できるよう、既存の設計に「改良」を加えた治療装置だった。セラック25では電 子銃と患者の間に置かれた金属製のターゲットに高エネルギーの電子を打ち込み、X線を発生させていた。セラック25のもう1つの「改良」点は、旧モデル『セラック20』の電気機械式の安全保護装置をソフトウェア制御に置き換えたことだった。ソフトウェアの方が信頼性が高いとの考えに基づく判断だった。

   しかし、技術者たちも知らなかった事実があった――セラック20およびセラック25に使われたOSは、正式な訓練も受けていないプログラマーが1人で作成したもので、バグが非常にわかりにくい構成になっていたのだ。「競合状態」と呼ばれる判明しにくいバグが原因で、操作コマンドを素早く打ち込んだ場合、セラック25ではX線用の金属製ターゲットをきちんと配置しないまま高エネルギーの放射線を照射する設定が可能になっていた。これにより少なくとも5人が死亡し、他にも重傷者が出た。

10大Bugの3個（by Wired News）

4. 1988年――バークレー版UNIX(BSD)のフィンガーデーモンによるバッファー・オーバーフロー：最初のインターネットワームとなった通称『モーリス・ワーム』は、バッファー・オーバーフローを悪用し、1日足らずで2000台から6000台のコンピューターに感染した。原因となったのは、標準入出力ライブラリー・ルーチン内の「gets()」という関数のコードだ。「gets()」関数はネットワーク越しにテキストを1行取得するように設計された。しかし、残念ながら「gets()」関数は入力を制限する ようには作られていない。そのため、あまりにも大きな入力があった場合には、接続可能なあらゆるマシンをワームが占拠する元凶になった。

   プログラマーは「gets()」関数を使用コードから排除することで問題に対処しているが、C言語の標準入出力ライブラリーからこれを削除することは拒否しており、この関数は現在も存在している。
5. 1988～1996年――『ケルベロス』の乱数生成アルゴリズム：ケルベロスは暗号を使ったセキュリティーシステムだが、乱数発生器に与えるシード（種）が適切でなく、真にランダムな乱数が生成されていなかった。

   その結果、ケルベロスによる認証を用いているコンピューターについて、非常に簡単な方法で侵入可能な状態が8年間にわたって続いた。このバグが実際に悪用されたかどうかは、今も定かではない。
6. 1990年1月15日―― 米AT&T社のネットワーク停止：米AT& T社の長距離電話用交換機『4ESS』を制御する最新版のソフトウェアにバグが入りこんだ。このため、4ESSは隣接するマシンの1つから、ある特定の メッセージを受け取るとクラッシュするようになってしまった――そしてそのメッセージとは、クラッシュした交換機が復帰した際に、隣接する交換機に送信するものだった。

   ある日、ニューヨークの交換機がクラッシュし再起動した。するとそれが原因で隣接する複数の交換機がクラッシュし、これらの交換機が再起動すると隣接する複数の交換機がさらにクラッシュし、この現象が延々と続いた。しばらくすると、114台の交換機が6秒ごとにクラッ シュと再起動を繰り返すようになった。この影響でおよそ6万人の人々が9時間にわたって長距離通話サービスを利用できなくなった。修復のため、技術者たち は1つ前のソフトウェアをロードした。

10大Bugの3個（by Wired News）

7. 1993年――インテル社製『Pentium』(ペンティアム)による浮動小数点数の除算ミス：米インテル社が大々的に売り出したPentiumチップが、特定の浮動小数点数の除算で誤りを引き起こした。たとえば、4195835.0／3145727.0を計算させると、正しい答えの1.33382ではなく1.33374となる。0.006%の違いだ。
　実際にこの問題の影響を受けるユーザーはごくわずかだったが、ユーザーへの対応から、同社にとって悪夢のような事態につながった。概算で300万～500 万個の欠陥チップが流通していた状況で、インテル社は当初、高精度のチップが必要だと証明できる顧客のみをPentiumチップの交換対象とした。しかし、最終的にインテル社は態度を改め、不満を訴えるすべてのユーザーのチップ交換に応じた。この欠陥は結局、インテル社に約4億7500万ドルの損害を与 えた。

8. 1995年／1996年――『Ping of Death』:[ピング・オブ・デス、不正なピングパケットによる攻撃] 分割送信されたIPパケットの再構成を行なうコードのチェックとエラー処理が不十分だったため、インターネット上の好きな場所から不正な形式のピングパケットを飛ばすことで、さまざまなオペレーティング・システム(OS)をクラッシュさせることができた。影響が最も顕著に現れたのはウィンドウズ搭載マシンで、この種のパケットを受け取ると、「死のブルー・スクリーン」と呼ばれる青い画面を表示して動作が停止してしまう。しかしこのバグを利用した攻撃は、ウィンドウズのみならず、マッキントッシュやUNIXを使ったシステムにも多くの被害をもたらした。

9. 1996年6月4日――『アリアン5』フライト501：欧州宇宙機関の開発したロケット、アリアン5には、『アリアン4』で使われていたコードが再利用されていた。しかし、アリアン5ではより強力なロケットエン ジンを採用したことが引き金となり、ロケットに搭載された飛行コンピューター内の計算ルーチンにあったバグが問題を起こした。エラーは64ビットの浮動小 数点数を16ビットの符号付き整数に変換するコードの中で起こった。アリアン5では加速度が大きいため、64ビット浮動小数点で表現される数がアリアン4 のときよりも大きくなってオーバーフローが起こり、最終的には飛行コンピューターがクラッシュしてしまった。
　フライト501では、最初にバックアップ・コンピューターがクラッシュし、それから0.05秒後にメイン・コンピューターがクラッシュした。その結果、エンジンの出力が過剰になり、ロケットは打ち上げ40秒後に空中分解してしまった。

10大Bugの3個（by Wired News）

10. 2000年11月――パナマ国立ガン研究所：米マルチデータ・システムズ・インターナショナル社(本社ミズーリ州)が製作した治療計画作成用ソフトウェアを使っていたパナマの国立ガン研究所で、放射線治療で照射する放射線量の計算を誤る一連の事故が起きた。
　マルチデータ社のソフトウェアでは、健康な組織を放射線から守るための「ブロック」と呼ばれる金属製のシールドの配置を、コンピューターの画面上に描いて決めるようになっていた。しかし、同社のソフトウェアではシールドが4個しか使えなかったにもかかわらず、パナマ人の技師たちはこれを5個使いたいと考え た。
　技師たちは、真ん中に穴を持つ1個の大きなシールドとして、5個のシールドをまとめて表示させれば、ソフトウェアをだますことができることを発見した。だが、そうした配置にした場合、穴の描き方によってこのソフトウェアが返す計算結果が違ってくることにはまったく気づいていなかった。ある方向に向けて描くと正しい照射量が計算されるが、違う方向に描くと必要な照射量の最大2倍の量を推奨してきたのだ。
　少なくとも8人の患者が死亡し、さらに20人が過剰照射によって深刻な健康被害を受けたとみられている。技師たちは、コンピュータによる計算結果を手作業で再チェックする法的義務を負っていたため、殺人罪で起訴されることになった。

11. 2003年8月14日――米国北東部の大停電：3分間に21の発電所が脱落し、オハイオ州の系統で電圧異常が起こった.

11月27日・本日のメニュー

データによる抽象化

- 2.2.1 Representing Sequences（並び）
- 2.2.2 Hierarchical Structures
- **2.2.3 Sequences as Conventional Interfaces**
- Exercises 2.33~43
- 2.3 Symbolic Data

seq: 慣用インタフェース

- 処理間のインタフェース
- API（Application Program Interface）
- Parameterでの受け渡し
- データ構造をインタフェースに使う。
- sequence を活用
- 例： 素数を求めるための The Sieve of Eratosthenes （エラトステネスの篩）

38

奇数の葉だけ2乗して和を取る

```
(define (count-leaves tree)
  (cond ((null? tree) 0)
        ((not (pair? tree)) 1)
        (else (+ (count-leaves (car tree))
                 (count-leaves (cdr tree)) ))
```

を基に，奇数の葉だけ2乗して和を取る

```
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree) (square tree) 0) )
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree)) ))
   ))
```

39

2つの手続きの構成は似ている?!

偶数のFibの和

```
(define (even-fibs n)
  (define (next k)
     (if (> k n)
         nil
         (let ((f (fib k)))
             (if (even? f)
                 (cons f (next (+ k 1)))
                 (next (+ k 1)) ))))
  (next 0) )
```

奇数の葉の二乗和

```
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree) (square tree) 0) )
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree))
              ))))
```

Ex.1.32 Accumulationを思い出そう

```
(define (accumulate combiner null-value term a
   next b)
  (if (> a b)
      null-value
      (combiner (term a)
         (accumulate combiner null-value
               term (next a) next b ))))
```

```
(define (sum term a next b)
  (accumulate + 0 term a next b) )
(define (product term a next b)
  (accumulate * 1 term a next b) )
```

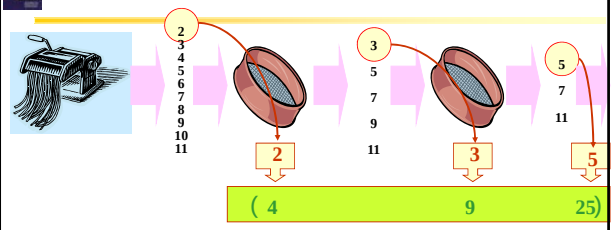
$$\sum_{i=a,next(i)}^{b} f(i)$$

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b)) ) )
```

$$\prod_{i=a,next(i)}^{b} f(i)$$

42

共通性の視点：素数の2乗を求める



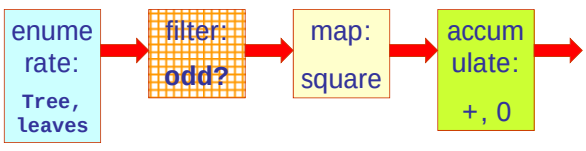
共通点を見る4つの基本手続き

- 数え上げ(enumerate)
- フィルタ(filter)
- 写像(map)
- 集約(accumulate)

43

4つの基本手続きから眺めると

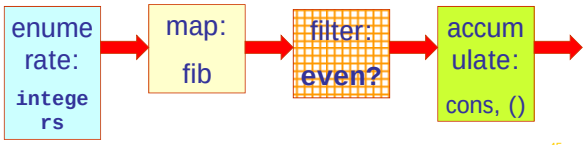
```
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree) (square tree) 0) )
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree))
              ))))
```



44

4つの基本手続きから眺めると

```
(define (even-fibs n)
  (define (next k)
     (if (> k n)
         nil
         (let ((f (fib k)))
             (if (even? f)
                 (cons f (next (+ k 1)))
                 (next (+ k 1)) ))))
  (next 0) )
```



45

4つの基本手続きをプログラム

```
(map square (list 1 2 3 4 5))
    ⇒

(define (filter predicate sequence)
  (cond ((null? sequence) nil)
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate
                       (cdr sequence)) ))
        (else (filter predicate
                      (cdr sequence))) ))

(filter odd? (list 1 2 3 4 5))
    ⇒
```

46

4つの基本手続きをプログラム(続)

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial
                      (cdr sequence) ))))

(accumulate + 0 (list 1 2 3 4 5))
    ⇒

(accumulate * 1 (list 1 2 3 4 5))
    ⇒

(accumulate cons nil (list 1 2 3 4 5))
    ⇒
```

47

4つの基本手続きをプログラム(続)

整数の並びの数え上げ(enumerate): e.g. even-fibs

```
(define (enumerate-interval low high)
  (if (> low high)
      nil
      (cons low
            (enumerate-interval
                 (+ low 1)
                 high ))))

(enumerate-interval 2 7)
    ⇒
(accumulate * 1 (enumerate-interval 1 5))
    ⇒
(accumulate + 0 (enumerate-interval 1 5))
    ⇒
```

48

4つの基本手続きをプログラム(続)

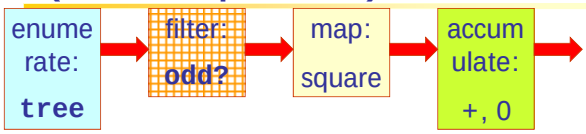
木の数え上げ(enumerate): e.g. even-fibs

```
(define (enumerate-tree tree)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (list tree))
        (else (append
                (enumerate-tree (car tree))
                (enumerate-tree (cdr tree))
              ))))

(enumerate-tree
    (list 1 (list 2 (list 3 4)) 5) )
    ⇒
```

49

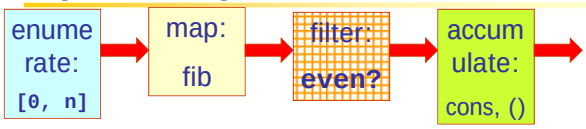
4つの基本手続きを使ってプログラム (Sum-odd-squares tree)




```
(define (sum-odd-squares tree)
  (accumulate
     +
     0
     (map
        square
        (filter
           odd?
           (enumerate-tree tree) ))))
```

50

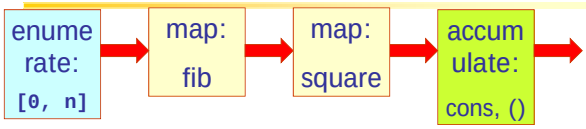
4つの基本手続きを使ってプログラム (even-fibs n)




```
(define (even-fibs n)
  (accumulate
     cons
     nil
     (filter
        even?
        (map
           fib
           (enumerate-interval 0 n) ))))
```

51

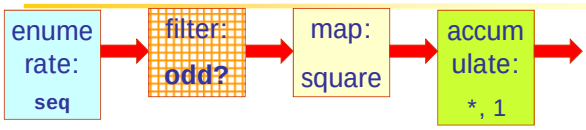
例題: list-fib-squares




```
(define (list-fib-squares n)
  (accumulate
     cons
     nil
     (map
        square
        (map
           fib
           (enumerate-interval 0 n) ))))
```

52

product-of-squares-of-odd-elements

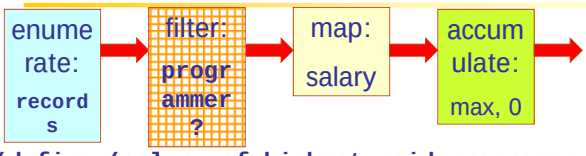



```
(define (product-of-squares-of-odd-elements seq)
  (accumulate
     *
     1
     (map
        square
        (filter odd? seq) )))

(product-of-squares-of-odd-elements
   (list 1 2 3 4 5) ) ⇒
```

53

salary-of-highest-paid-programmer




```
(define (salary-of-highest-paid-programmer
                    records )
  (accumulate
     max
     0
     (map
        salary
        (filter programmer? records) )))
```

データベース問い合わせ(query)言語の原型

54

Ex2.33 Using accumulate

```
■(define (my-map p sequence)
   (accumulate
      (lambda (x y) (cons (p x) y))
      nil
      sequence ))
■(define (my-append seq1 seq2)
    (accumulate cons seq2 seq1) )
■(define (my-length sequence)
    (accumulate
       (lambda (x y)
          (if (null? x) y (+ 1 y)) )
       0
       sequence ))
```

56

Horner's rule(Hornerの方法)

$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ を計算するのに

$(\ldots(a_n x + a_{n-1})x + \ldots + a_1)x + a_0$ と変形する

coefficient-sequence: (a_n … a_3 a_2 a_1 a_0)

```
(define (horner-eval x coefficient-sequence)
  (accumulate
     (lambda (this-coeff higher-term)
        (+ (* higher-term x) this-coeff) )
     0
     coefficient-sequence ))
(horner-eval 2 (list 1 3 0 5 0 1))
```

⇒ 225

57

宿題：12月4日正午締切

- 4つの基本手続の考え方を習得
- 宿題は、次の3題：
- Ex.2.33, 2.34, 2.35.
- 随意宿題　2.42 (n-queens).

DON'T PANIC!



58