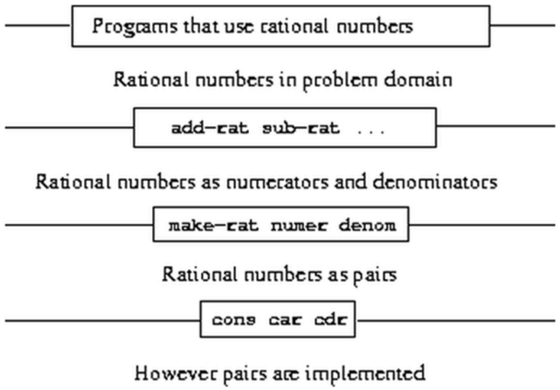
VL Nichtprozedurale Programmierung

Funktionale Programmierung

Andreas Naderlinger
Fachbereich Computerwissenschaften
Software Systems Center
Universität Salzburg

Last time …

- Last time we talked about
  - Functions: (i) Higher-Order Procedures, (ii) Lambda (λ)
  - Data:
    - Data abstraction → Compound Data
      - Based on Pairs
        - » Constructors (cons)
        - » Selectors (car, cdr)
      - Barriers



    - What is meant by data? What do we need to represent data?
      - no data structures, just procedures
      - the ability to manipulate procedures as objects automatically provides the ability to represent compound data
    - Hierarchical Data
      - Lists

Today …

- what inspired Google when making

MapReduce [MapReduce1]

Today ...

- what inspired Google when making

MapReduce [MapReduce1]

- Programming model
  - for processing and generating **large data sets**
  - on clusters with many machines
  - With
    - Automatic parallelization and distribution
    - Fault-tolerance
    - I/O scheduling
    - Status and monitoring

Petabytes, ...

thousands

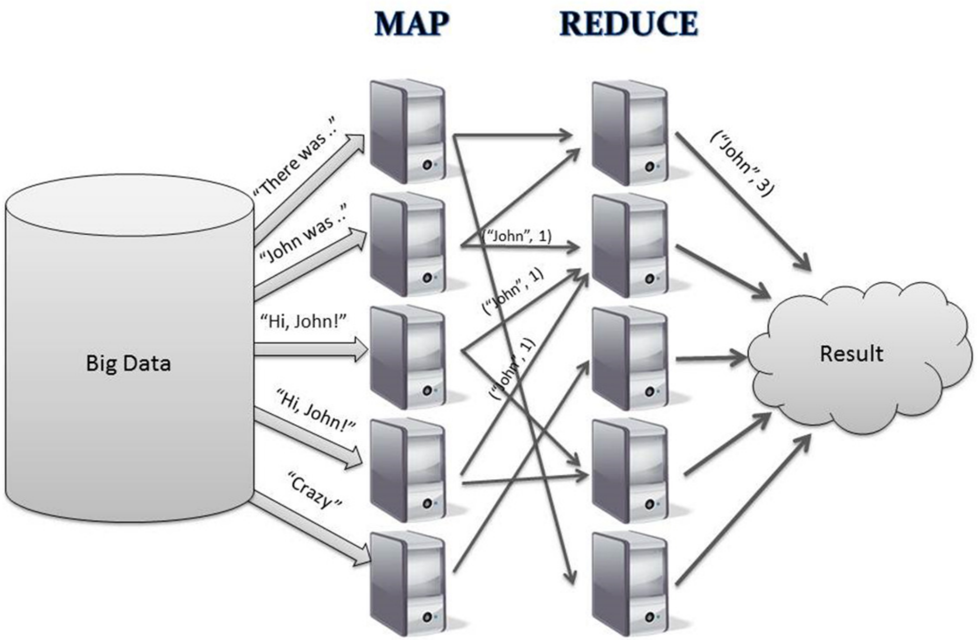
→ Should be easy to use.

E.g. [MapReduce2], Sorting 1 petabyte (10 trillion 100-byte records) with map reduce:

2008: 6 hours on 4000 machines

2011: 33 minutes on 8000 machines

- Many frameworks that implement MapReduce. E.g. Hadoop (Java)
- Resources: E.g. Amazon Elastic MapReduce web service



The user code consists of only two functions:
map
reduce

"Our abstraction is inspired by the map and reduce primitives present in **Lisp** and many other functional languages."

[MapReduce1]

- Functional programming is great for distribution.
  - The idea of map/reduce exists in many functional languages

- MapReduce will probably be a topic in later semesters (e.g. on distributed systems)

Topics today

- Lists
- Higher-order Procedures on Lists
- Conventional Interfaces

```
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))
```

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

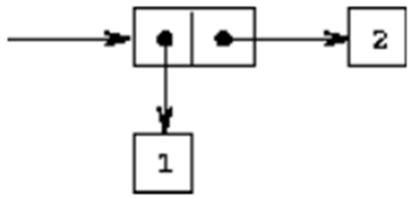
sum cube a inc b

Today: a short one-liner with built-in functions only

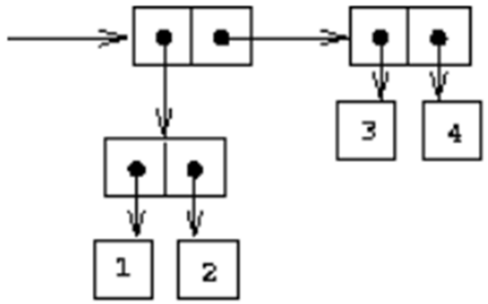
Hierarchical Data (box-pointer-representation)

from last lecture

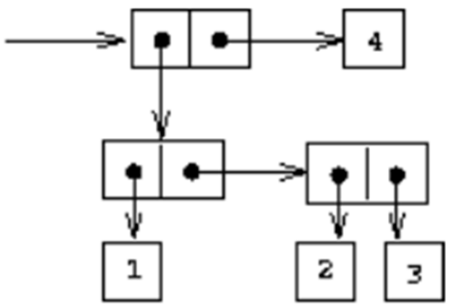
- (cons 1 2)



- (cons (cons 1 2)
  (cons 3 4))

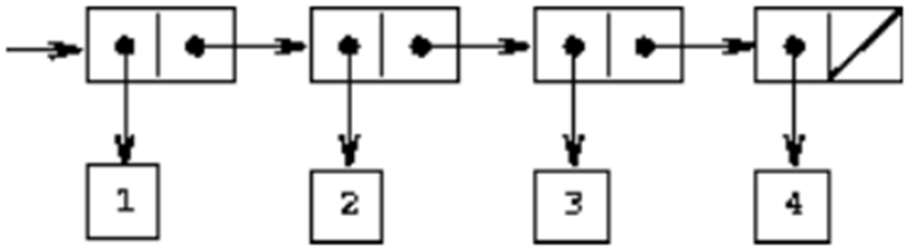


- (cons (cons 1
  (cons 2 3))
  4)



Sequences (by chaining pairs)

from last lecture




- ```
  (cons 1
        (cons 2
              (cons 3
                    (cons 4 null))))
  ```

(cons <a_1> (cons <a_2> (cons ...(cons <a_n> null) ...)))

is the same as

(list <a_1> <a_2> ... <a_n>)

- `(list 1 2 3 4)`

null terminates the chain of pairs.
→ empty list

```
(define mylist (list 1 2 3 4))
(car mylist)
(cdr mylist)
(car (cdr mylist))
(car (cdr (cdr mylist)))
```
```

Lists

from last lecture

- Creation
  - (list 1 2 3)
- empty list
  - `'()`, `null`, `empty` , `(list)`
- check for emptiness
  - `null?`, `empty?`
- selection
  - `car`, `cdr` (for pairs and lists)
  - `first`, `rest` (for lists)

List operation

from last lecture

- Get the n-th element of the list
- `(define (list-ref items n)`

```
conditional (if (= n 0)
                (first items) base case
                (list-ref (rest items) (- n 1))))
                                        self-referential case
```

- Get the length of the list
- `(define (length items)`

```
    (if (null? items)
        0
        (+ 1 (length (rest items)))))
```

Demo3g

Your turn

We know:
null?, first, rest,
car, cdr,
length, list-ref

- Implement ***last***
  - Returns the last element of a list

List operations (cont'd)

- Append
  - (define squares (list 1 4 9 16 25))
  - (define odds (list 1 3 5 7))

  - (append squares odds)
    (1 4 9 16 25 1 3 5 7)

```
(define (myappend xs ys)
  (if (null? xs)
      ys
      (cons (first xs) (myappend (rest xs) ys))))
```

Demo3g

List operations (cont'd)

We know:
null?, first, rest,
car, cdr,
length, list-ref,
append

- Last
-
```
(define (last xs)
  (if (null? (cdr xs))
    (car xs)
    (last (cdr xs))))
```

Demo4a

- Reverse
-
```
(define (rev xs)
   (if (null? xs)
      null
      (append (rev(cdr xs)) (list (car xs)))))
```

Mapping over lists

- Map
  - apply some transformation to each element in a list and generate the list of results

- Motivation

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

Mapping over lists (2)

- Map

```
(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)
        null
        (cons (proc (car items))
              (map proc (cdr items)))))
  ```

- Examples:
  - `(map abs (list -10 2.5 -11.6 17))`
    `(10 2.5 11.6 17)`
  - `(map (lambda (x) (* x x)) (list 1 2 3 4))`
    `(1 4 9 16)`

Demo4b
  ```

[Motivation] Example

- We want to get only **even** fibonacci numbers:
  - Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89…
  - →0, 2, 8, 34, …
  - We want a list of all even fibonacci numbers Fib(k), where k is less than or equal a given integer n:
    - E.g. (even-fibs 10) → (0 2 8 34)

- We already know how to get **the kth fibonacci** number (using the function **fib**):

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

[Example]....implemented in **Java** and **Racket**

Demo4c

Java

```
private static List<Integer> evenFibs(int n) {
   List<Integer> evens = new ArrayList<Integer>();
   for (int i = 0; i < n; i++) {
     int f = fib(i);
     if (f % 2 == 0) {
       evens.add(f);
     }
   }
   return evens;
 }
```

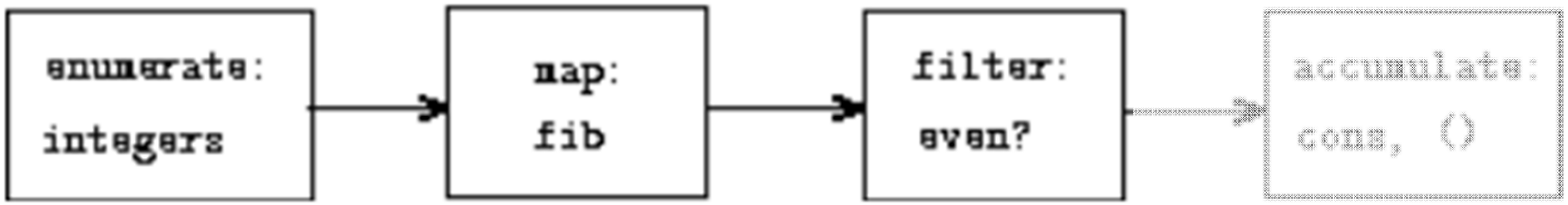
Racket

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

You told us that func. prog. is more concise and easier to read?

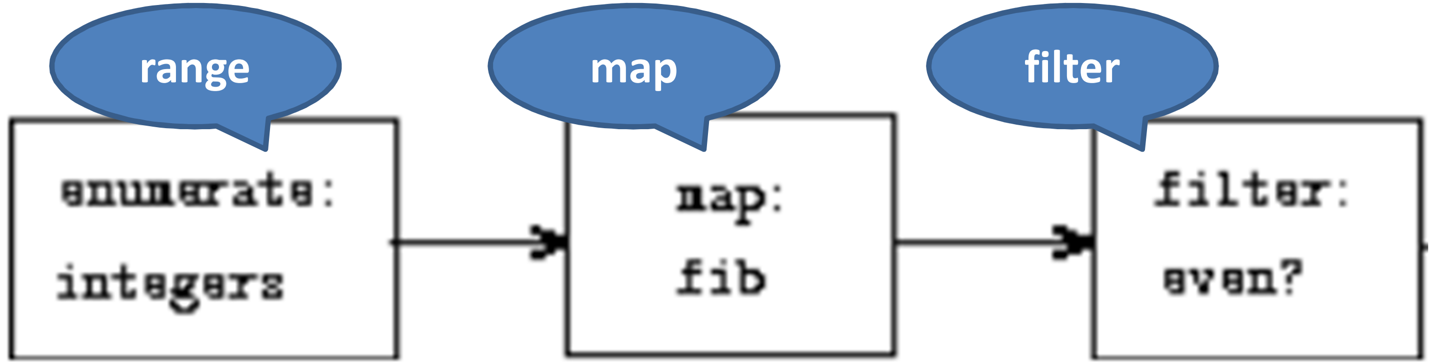
[Example]....the idea

- enumerate the integers from 0 to *n*;
- compute the Fibonacci number for each integer;
- filter them, selecting the even ones; and
- accumulate the results using cons, starting with the empty list



(signal flow representation)

[Example]...the real **FP** implementation




```
(range 0 10)
→ '(0 1 2 3 4 5 6 7 8 9)

(map fib <result>)
→ '(0 1 1 2 3 5 8 13 21 34)

(filter even? <result>)
→ '(0 2 8 34)
```

```
(filter even? (map fib (range 0 10)))
```

→ (0 2 8 34)

Is this concise enough for you? ;-)

Sequences as **Conventional Interfaces**

- Concentrate on the “signals” that flow from one stage in the process to the next.
  - Lists
  - List operations

- Lists
- List operations (exact names are language depending; small diffs)
  - enumerate, range
  - filter
  - map
  - accumulate, reduce, fold
  - …

range [aka enumerate]

demo4d

- `(range 10)`
  - `'(0 1 2 3 4 5 6 7 8 9)`
- `(range 10 20)`
  - `'(10 11 12 13 14 15 16 17 18 19)`
- `(range 20 40 2)`
  - `'(20 22 24 26 28 30 32 34 36 38)`
- `(range 20 10 -1)`
  - `'(20 19 18 17 16 15 14 13 12 11)`

map

```
;; map : (X  ->  Y) (listof X)  ->  (listof Y)
;; to construct a list by applying f to each item on alox
;; that is, (map f (list x-1 ... x-n)) =
;; (list (f x-1) ... (f x-n))
(define (map f alox) ...)
```

- `(map abs (list -10 2.5 -11.6 17))`
  - `'(10 2.5 11.6 17)`
- `(map (lambda (x) (* x x)) (list 1 2 3 4))`
  - `'(1 4 9 16)`
- `(map sin (list 0.0 (/ pi 2)))`
  - `'(0.0 1.0)`

filter

```
;; filter : (X  ->  boolean) (listof X)  ->  (listof X)
;; to construct a list from all those items on alox for which p holds
(define (filter p alox) ...)
```

- `(define lst (range 10))`
  - `'(0 1 2 3 4 5 6 7 8 9)`
- `(filter odd? lst)`
  - `'(1 3 5 7 9)`
- `(filter (λ(x) (= 4 (square x))) lst)`
  - `'(2)`

```
(define (myfilter predicate sequence)
  (cond ((null? sequence) null)
        ((predicate (car sequence))
         (cons (car sequence)
               (myfilter predicate (cdr sequence))))
        (else (myfilter predicate (cdr sequence)))))
```

accumulate [**foldr** in Racket][reduce]

```
;; foldr : (X Y  ->  Y) Y (listof X)  ->  Y
;; (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))
(define (foldr f base alox) ...)
```

- `(define lst (range 10))`
  - `'(0 1 2 3 4 5 6 7 8 9)`
- `(foldr + 0 lst)`
  - `45`

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

foldr vs. foldl [in Racket]

```
;; foldr : (X Y  ->  Y) Y (listof X)  ->  Y
;; (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))
(define (foldr f base alox) ...)

;; foldl : (X Y  ->  Y) Y (listof X)  ->  Y
;; (foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))
(define (foldl f base alox) ...)
```

- `(foldr - 0 '(1 2))` `(- 1 (- 2 0))`
  - `-1`
- `(foldl - 0 '(1 2))` `(- 2 (- 1 0))`
  - `1`

- `(foldr - 0 '(1 2 3 4))` `(- 1 (- 2 (- 3 (- 4 0))))`
  - `-2`
- `(foldl - 0 '(1 2 3 4))` `(- 4 (- 3 (- 2 (- 1 0))))`
  - `2`

foldl is implemented differently in other languages! E.g., Haskell:

foldr (-) 0 [1..10]	(1-(2-(...10-0**)...)**	-5
foldl (-) 0 [1..10]	**(...(**0-1)-2)...)-10)	-55

in Racket: → 5

we will only use **foldr**

```
(filter even? (map fib (range 0 10)))
```

→ (0 2 8 34)

```
(foldr + 5000 (filter even? (map fib (range 0 10))))
```

→ 5044

… → sum-cubes

Demo4f

[Conclusion] Conventional Interfaces

- Benefits
  - The value of expressing programs as sequence operations is that this helps us make program designs that are **modular**
    - i.e., designs that are constructed by **combining relatively independent pieces**.
    - We can encourage modular design by providing a **library of standard components** together with a **conventional interface** for connecting the components in flexible ways.
  - Modular construction is a powerful strategy for **controlling complexity** in engineering design.
  - Functionality available in many many many languages:
    - Lisp-family, Haskell, Clojure, Erlang, F#, OCaml, Scala, Ruby, Python, Java8, …
- Drawbacks
  - to come …

That’s it for today

References

- SICP
- HTDP
- [MapReduce] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters
- [MapReduce2] Grzegorz Czajkowski et al. Sorting Petabytes with MapReduce - The Next Episode

Thank you!