DDG University

Informal Goals

1. Learn new things related to technology.
2. Learn from each other.
3. Foster inter-team building.
4. To become better engineers.

Search for DDG University in Asana.

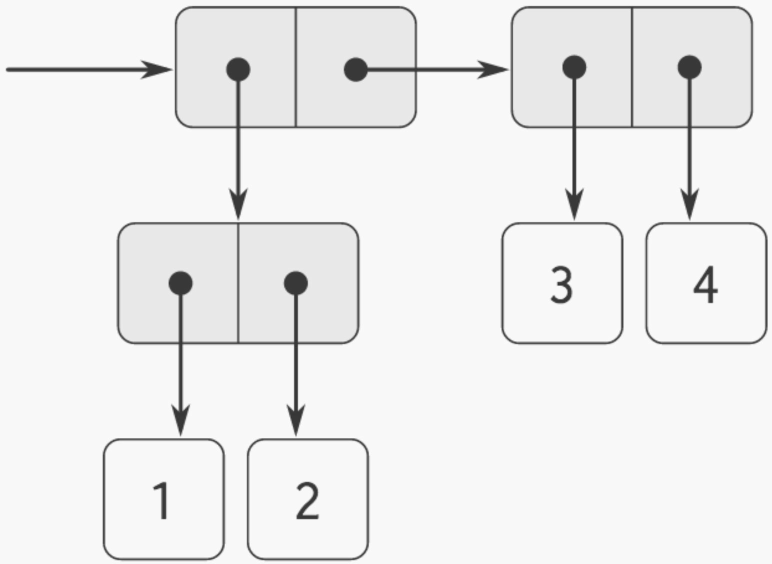
Structure and Interpretation of Computer Programs *(SICP)*

by Harold Abelson and Gerald Jay Sussman

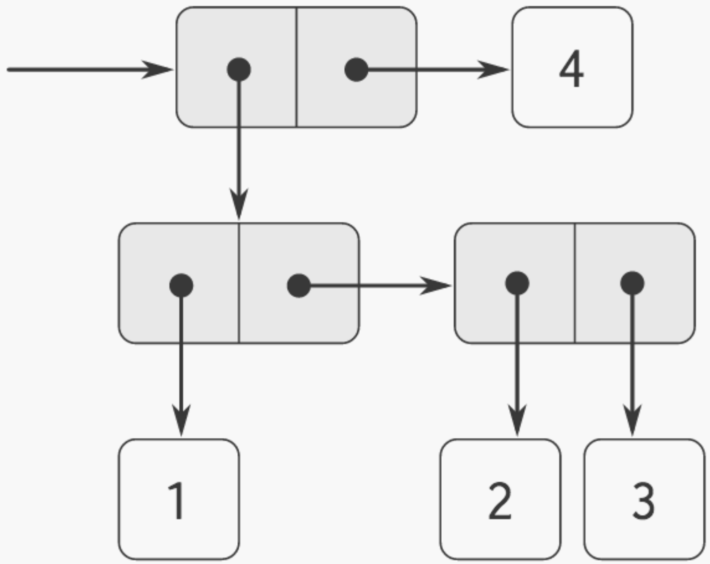
2.2 Hierarchical Data and the Closure Property

1. ***Structured data***
2. ***Closure***
3. ***Sequences***

Internal Structures



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



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

Closure Properity

- In the mathematical sense: when an operation on members of a set results in a member of the same set.

-or-

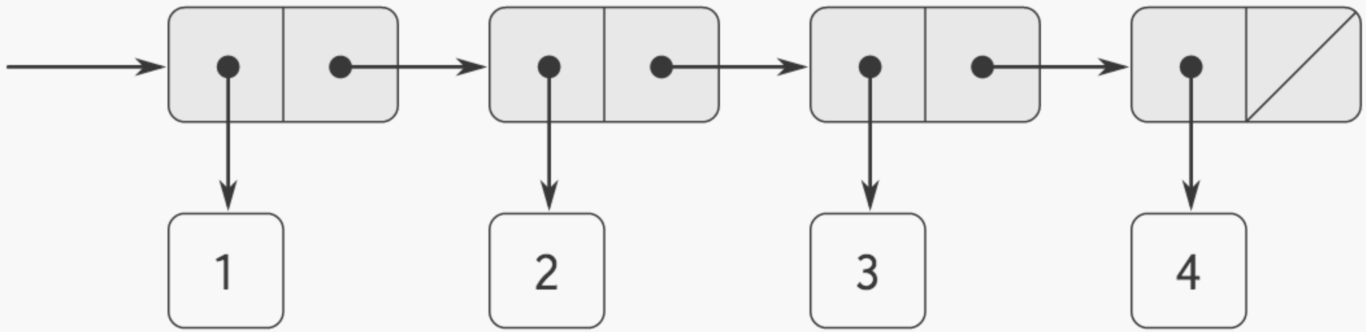
- An operation for combining data objects satisfies the closure property if the results of combining things with that operation can themselves be combined using the same operation. For example: `cons` creates pairs who's elements are pairs.

-NOT-

- In the programming sense: accessing lexially scoped variables bound in a function.

Sequences

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



-or-

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

Sequences (cont.)

Some examples:

```
(define one-through-four (list 1 2 3 4))

one-through-four
;(1 2 3 4)
```

```
(car one-through-four)
;1
```

```
(cdr one-through-four)
;(2 3 4)
```

```
(car (cdr one-through-four))
;2
```

```
(cons 10 one-through-four)
;(10 1 2 3 4)
```

```
(cons one-through-four 10)
;((1 2 3 4) . 10)          ; WAT?
```

A Little More Lisp

- list
- null?
- pair?
- cadr, cdar, etc...

list

```
(list ⟨a1⟩ ⟨a2⟩ … ⟨an⟩)
```

is equivalent to

```
(cons ⟨a1⟩
  (cons ⟨a2⟩
    (cons …
      (cons ⟨an⟩
        nil)…)))
```

```
;(⟨a1⟩ ⟨a2⟩ … ⟨an⟩)
```

null? and pair?

`null?` - predicate to test for an empty list:

```
(null? ())
;#t

(null? (list 1))
;#f
```

`pair?` - predicate to test for a list:

```
(pair? (cons 1 2))
;#t

(pair? (list 1 2 3))
;#t

(pair? 1)
;#f
```

cadr, cdar, etc...

You can combine list accessors into one operations:

```
(car (cdr (list 1 2 3 4))
;2

(cadr (list 1 2 3 4))
;2

(car (car (list (list 4 3 2 1))))
;4

(caar (list (list 4 3 2 1)))
;4
```

List Operations

cdr-ing down a list:

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

(length (list 1 3 5 7))
;4
```

cons-ing up a list:

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

(append (list 1 2 3 4) (list 4 3 2 1))
;(1 2 3 4 4 3 2 1)
```

Mapping Over Lists

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

(scale-list (list 1 2 3 4 5) 10)
;(10 20 30 40 50)
```

We can abstact this to `map`:

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

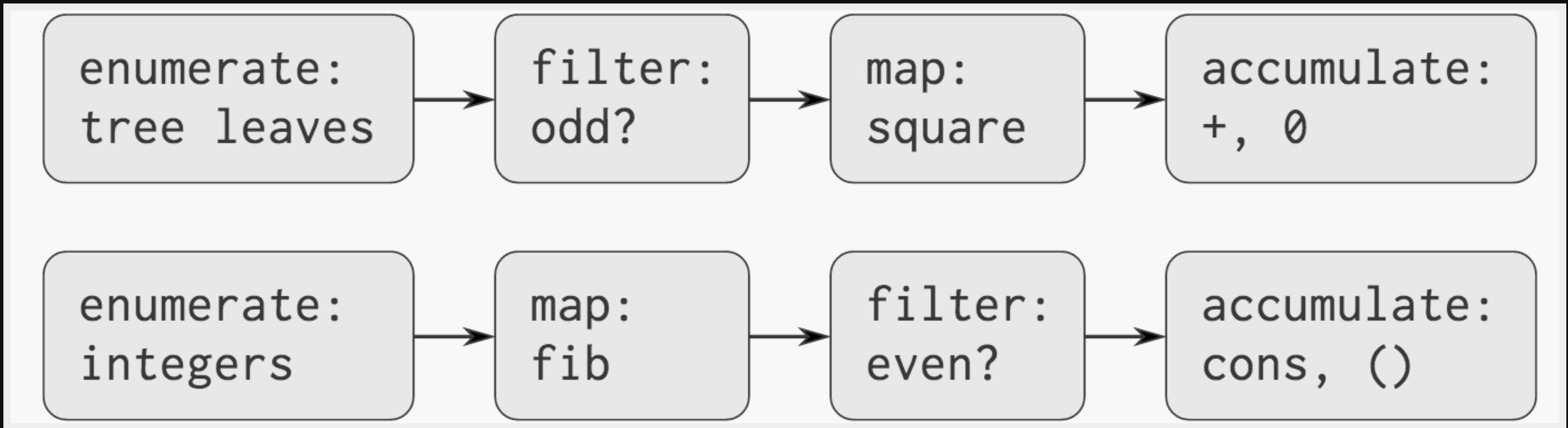
and redefine `scale-list`:

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

Sequences Operations

```scheme
(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))))))
```

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



Sequences Operations (cont.)

Map

```
(map square (list 1 2 3 4 5))
;(1 4 9 16 25)
```

Sequences Operations (cont.)

Filter

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

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

Sequences Operations (cont.)

Accumulate

```
(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))
;15
(accumulate cons () (list 1 2 3 4 5))
;(1 2 3 4 5)
```

Sequences Operations (cont.)

Enumerate

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

(enumerate-interval 2 7)
;(2 3 4 5 6 7)
```

Sequences Operations (cont.)

```scheme
(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))))))
```

becomes:

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

Sequences Operations (cont.)

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

becomes:

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

Wrapping-up

- Structured data
- Closure
- Sequences

That's all for section 2.2.

Thanks!