Procedural abstraction example: sqrt

To find an approximation of the square root of x:

- Make a guess G
- Improve the guess by averaging G and x/G
- Keep improving the guess until it is good enough

```scheme
(define try (lambda (guess x)
  (if (good-enuf? guess x)
      guess
      (try (improve guess x) x))))

(define improve (lambda (guess x)
  (average guess (/ x guess))))

(define average (lambda (a b) (/ (+ a b) 2)))

(define good-enuf? (lambda (guess x)
  (< (abs (- (square guess) x)) 0.001)))

(define sqrt (lambda (x) (try 1 x)))
```

Pairs (cons cells)

- `(cons <x-exp> <y-exp>) ==> <P> ;type: x, y Pair
  - Where <x-exp> evaluates to a value <x-val>,
    and <y-exp> evaluates to a value <y-val>
  - Returns a pair <P> whose
    car-part is <x-val> and whose
cdr-part is <y-val>

- `(car <P>) ==> <x-val> ;type: Pair type of car part
  - Returns the car-part of the pair <P>

- `(cdr <P>) ==> <y-val> ;type: Pair type of cdr part
  - Returns the cdr-part of the pair <P>
Lists - Box and pointer diagram

- `(cons <el1> (cons <el2> nil))`

Printed representation

`(define x (list 1 2))`
`(define y (list (list 1 2) (list 1 2)))`
`(define z (list x x))`

\[ X == (1 2) \]
\[ Y == ((1 2) (1 2)) \]
\[ Z == ((1 2) (1 2)) \]

Cons, car, cdr

`(define thing (cons (cons 1 nil) (cons 2 (cons 3 nil))))`

\[ \text{thing} \Rightarrow ((1) 2 3) \]
\[ (\text{cons } 1 \text{ nil}) \Rightarrow (1) \]
\[ (\text{cons } 1 \ (\text{cons } 2 \text{ nil})) \Rightarrow (1 2) \]
\[ (\text{car thing}) \Rightarrow (1) \]
\[ (\text{cdr thing}) \Rightarrow (2 3) \]
\[ (\text{car (car thing)}) \Rightarrow 1 \]
\[ (\text{car (cdr (cdr thing)}) \Rightarrow 3 \]

List drill

`(car (cons (+ 1 2) (- 3 4))) \Rightarrow 3`
`(cdr 6) \Rightarrow \text{error}`
`(cdr (car (cons (cons 1 2) (cons 3 4)))) \Rightarrow 2`
`(pair? #t) \Rightarrow #f`
`(pair? (car (cons 1 2))) \Rightarrow #f`
`(pair? (cons (+ 1 2) (car (cons 3 4)))) \Rightarrow #t`

Length of a list

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

cdr'ing down a list

`(define (list-ref lst n)
  (if (= n 0)
    (car lst)
    (list-ref (cdr lst) (- n 1)))`
More list drill

x => (())
y => (1 2 3)
z => (1 2 3 4 5)
w => (1 2 3 4 5)

(length x)
(length y)
(length z)
(list-ref z 2)
(append x y)
(cons x y)

Orders of Growth

What is the order of growth of last-k?

(define (last-k k lst)
  (if (= (length lst) k)
      lst
      (last-k k (cdr lst)))))

Writing some procedures

The procedure copy-some that copies the first n elements of a list
(copy-some 3 (list 1 2 3 4 5)) ==> (1 2 3)

(define (copy-some n list)
  ;
  )

More list drill

x => (())
y => (1 2 3)
z => (1 2 3 4 5)
w => (1 2 3 4 5)

(length x)
1
(length y)
3
(length z)
3
(list-ref z 2)
((4))
(append x y)
(() 1 2 3)
(cons x y)
(((1) 1 2 3))

Orders of Growth

What is the order of growth of last-k?

(define (last-k k lst)
  (if (= (length lst) k)
      lst
      (last-k k (cdr lst)))))

Θ( n^2 )

Writing some procedures

The procedure copy-some that copies the first n elements of a list
(copy-some 3 (list 1 2 3 4 5)) ==> (1 2 3)

(define (copy-some n list)
  (if (= n 0)
      nil
      (cons (car list)
            (copy-some (- n 1) (cdr list)))))
)
Writing some procedures

(repeat x m) returns a list containing the value x repeated m times. For example:
(repeat 5 3) => (5 5 5)
(repeat (list 1 2) 2) => ((1 2) (1 2))

Recursive solution:
(define (repeat x m)
  (if (= m 0)
      nil
      (cons x (repeat x (- m 1)))))

time O(m), space O(m)

Iterative solution:
(define (repeat x m)
  (define (helper i answer)
    (if (> i m)
        answer
        (helper (+ 1 i) (cons x answer))))
  (helper 0 nil))

time O(m), space O(1)

Writing some procedures

(append lst1 lst2) appends lst2 to the end of lst1. E.g.:
(append (list 0 1 2) (list 3 4)) => (0 1 2 3 4)
(append nil (list 5 6)) => (5 6)

Recursive solution:
(define (append lst1 lst2)
  (if (null? lst)
      lst2
      (cons (car lst1)
            (append (cdr lst1) lst2))))

time O(n^2), since append runs in linear time O(m) where m=(length answer), each successive call to append takes more and more time: 1 + 2 + 3 + ... + n = O(n^2)
space O(n), where n=(length lst)

Iterative solution:
(define (append lst1 lst2)
  (define (helper rest answer)
    (if (null? rest)
        answer
        (helper (cdr rest) (cons (car rest) answer))))
  (helper lst nil))

time O(n^2), space O(1), where n=(length lst)

Writing some procedures

(reverse lst) reverses lst. E.g.:
(reverse (list 0 1 2)) => (2 1 0)
(reverse (list (list 3 5) (list 2 4))) => ((2 4) (3 5))

Recursive solution:
(define (reverse lst)
  (if (null? lst)
      nil
      (append (reverse (cdr lst))
              (list (car lst)))))

time O(n^2), since append runs in linear time O(m) where m=(length answer), each successive call to append takes more and more time: 1 + 2 + 3 + ... + n = O(n^2)
space O(n), where n=(length lst)

Iterative solution:
(define (reverse lst)
  (define (helper rest answer)
    (if (null? rest)
        answer
        (helper (cdr rest) (cons (car rest) answer))))
  (helper lst nil))

time O(n^2), space O(1), where n=(length lst)
Writing some procedures

(num-leaves tree) returns the number of leaves in a tree, where a leaf is anything that isn’t a list (number, string, boolean, procedure). E.g.:

(num-leaves (list 0 1 (list 3 5) (list 2 (list 4))))) => 6

(num-leaves (list (list (list (list nil))))) => 0

Writing some procedures

(define (num-leaves tree)
  (cond ((null? tree) 0)
    ((pair? tree) (+ (num-leaves (car tree))
      (num-leaves (cdr tree))))
    (else 1)))

time O(n), space O(1), where n is the number of pairs in the tree, and d is the depth of the tree.

Remember...

• calendar...