Knowledge-based systems in Bioinformatics, 1MB602

Lecture 3:
Lists and sequences

Lecture Overview

- Data abstraction
- Lists
- Trees
- Hierarchical structures

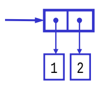
Review: Pair abstraction

- Constructor
  `(cons <x> <y>)` → `<Pair>`
- Accessors
  `(car <Pair>)` → `<x>`
  `(cdr <Pair>)` → `<y>`
- Predicate
  `(pair? <z>)`
  → `#t` if `<z>` evaluates to a pair, else `#f`

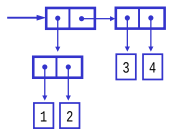
Box-and-pointer notation

- Standard way of visualizing a pair
  - Each object is shown as a pointer to a box

```
(cons 1 2)
```



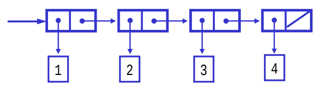
```
(cons (cons 1 2)
      (cons 3 4))
```



- Closure property of cons

Representing sequences

- One of the useful structures we can build with pairs is sequences – an ordered collection of data objects
- Straightforward representation:



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

$\Big\}$ `(list 1 2 3 4)`

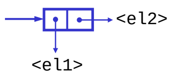
Lists

- A list is a data object that can hold an arbitrary number of ordered items.
- More formally, a list is a sequence of pairs with the following properties:
  - Car-part of a pair in the sequence – holds an item
  - Cdr-part of a pair in the sequence – holds a pointer to the rest of the list
  - Empty-list `()` – signals no more pairs, or end of list
- Note that lists are closed under operations of `cons` and `cdr`.*

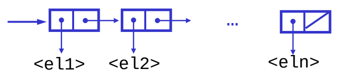
*a set of elements is said to be closed under an operation if applying the operation to elements in the set produces an element that is again an element of the set

Lists cont.

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



```
(list <el1> <el2> … <eln>)
```



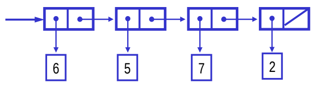
Predicate:

```
(null? <z>)
```

→ `#t` if `<z>` evaluates to empty list

Operations on lists

```
(define my-list (list 6 5 7 2))
```



- **Retrieving elements from my-list**

```
(car my-list) → 6
(cdr my-list) → (5 7 2)
(car (cdr my-list)) → 5
(cdr (cdr (cdr (cdr my-list)))) → ()
```

- **Possible to abstract to general procedure?**

```
(nth my-list 0) → 6
(nth my-list 2) → 7
```

Operations on lists cont.

```
(define (nth lst n)
  (if (= n 0)        ;base case
      (car lst)
      (nth (cdr lst) ;recursive case
           (- n 1))))
```

How to compute the length of a list?

```
(define (length lst)
   (if (null? lst) ;base case
       0
       (+ 1        ;recursive case
          (length (cdr lst)))))
```

Cons'ing up lists

Representing intervals with lists

```
(define (enumerate-interval from to)
  (if (> from to)
      ()
      (cons from
            (enumerate-interval
              (+ 1 from)
              to))))
```

Trace:

```
(enumerate-interval 2 4)
(if (> 2 4) () (cons 2 (enumerate-interval (+ 1 2) 4)))
(if #f () (cons 2 (enumerate-interval 3 4)))
(cons 2 (enumerate-interval 3 4))
(cons 2 (cons 3 (enumerate-interval 4 4)))
(cons 2 (cons 3 (cons 4 (enumerate-interval 5 4))))
(cons 2 (cons 3 (cons 4 ())))

            → (2 3 4)
```

Mapping over lists

- Square each element in a list

```
(define (square-list lst)
   (if (null? lst)
       ()
       (cons (square (car lst))
             (square-list (cdr lst)))))
```

- Scale each element by a factor

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

Mapping over lists cont.

More generally:

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

(define (square-list lst)
   (map square lst))

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

Filtering lists

- Keep all odd integers in a list

```
(define (keep-odds lst)
   (cond ((null? lst) ())
         ((odd? (car lst))
          (cons (car lst)
                (keep-odds (cdr lst))))
         (else (keep-odds (cdr lst)))))
```

- Keep all integers greater than 10

```
(define (keep-ten-greaters lst)
   (cond ((null? lst) ())
         ((> (car lst) 10)
          (cons (car lst)
                (keep-ten-greaters (cdr lst))))
         (else
           (keep-ten-greaters (cdr lst)))))
```

Filtering lists cont.

- More generally:

```
(define (filter pred lst)
   (cond ((null? lst) ())
         ((pred (car lst))
          (cons (car lst) (filter pred (cdr lst))))
         (else (filter pred (cdr lst)))))

(define (keep-odds lst)
   (filter odd? lst))

(define (keep-ten-greaters)
   (filter (lambda (x) (> x 10)) lst))
```

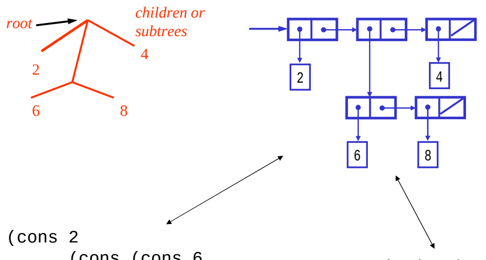
Accumulating results

```
(define (add-up lst)
  (if (null? lst)
      0
      (+ (car lst)
         (add-up (cdr lst)))))

(define (mult-all lst)
  (if (null? lst)
      1
      (* (car lst)
         (mult-all (cdr lst)))))

(define (accumulate op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (accumulate op init
                         (cdr lst)))))
(define (add-up lst)
  (accumulate + 0 lst))
```

Trees

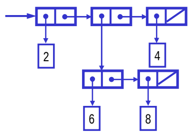


```
(cons 2
      (cons (cons 6
                  (cons 8 ()))       (2 (6 8) 4)
            (cons 4 ())))
```

Tree procedures

- A tree is a list → we can use list procedures on them

```
(define my-tree
    (list 2 (list 6 8) 4))
```



```
(length my-tree) → 3

(countleaves my-tree) → 4
```

Countleaves procedure

- Strategy
  - base case: count of the empty tree is 0
  - base case: count of a leaf is 1
  - recursive strategy: the count of a tree is the sum of the countleaves of each child in the tree
- Implementation:

```
(define (countleaves tree)
  (cond ((null? tree) 0)       ;base case
        ((leaf? tree) 1)       ;base case
        (else                  ;recursive case
          (+ (countleaves (car tree))
             (countleaves (cdr tree))))))

(define (leaf? x)
  (not (pair? x)))
```

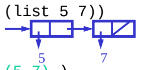
Countleaves example

```
(define (countleaves tree)
  (cond ((null? tree) 0) ;base case
        ((leaf? tree) 1)      ;base case
        (else                 ;recursive case
          (+ (countleaves (car tree))
             (countleaves (cdr tree))))))

(define (leaf? x)
  (not (pair? x)))
```

Example #1 with substitution model

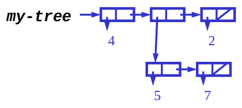
```
(countleaves (list 5 7))
(countleaves                    )
```



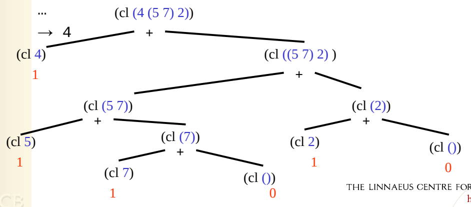
```
(countleaves (5 7) )
(+ (countleaves 5) (countleaves (7) ))
(+ 1 (+ (countleaves 7) (countleaves ())))
(+ 1 (+ 1 0))
→ 2
```

Countleaves – bigger example

```
(define my-tree (list 4 (list 5 7) 2))
```



```
(countleaves my-tree)
(countleaves (4 (5 7) 2) )
(+ (countleaves 4) (countleaves ((5 7) 2) ))
…
→ 4
```



Finding all the primes

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

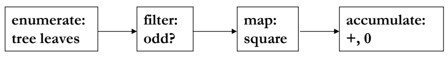
… And here is how to do it!

```
(define (sieve lst)
  (if (null? lst)
      ()
      (cons (car lst)
            (sieve
             (filter (lambda (x)
                       (not (divisible? x (car lst))))
                     (cdr lst))))))


(sieve (enumerate-interval 2 420))
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
 73 79 83 89 97 101 103 107 109 113 127 131 149 151 157
 163 167 173 179 181 191 193 197 199 211 223 227 229 233
 239 241 251 257 263 269 271 277 281 293 307 311 313 317
 331 337 347 349 353 359 367 373 379 383 389 397 401 409
 419)
```

Sum-odd-squares

Sum the squares of all odd elements in a tree



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

Calculate GC content

- Calculate the GC content of a DNA segment
  - GC content: the proportion of GC base pairs
  - AT content = 1 – GC content
- Why?
  - Correlation (weak) between GC-content and gene rich regions
  - Classification of organisms in taxonomies
- Representational issues
  - One strand is enough
  - List representation
    `('A 'C 'G 'C 'A 'T 'G 'C ... 'A)`

Some calc-GC-cont variants

```
(define (calc-GC-cont1 segm)
   (let (frac (/ 1 (length segm)))
       (accumulate + 0
                  (map
                    (lambda (x) frac)
                    (filter GC? segm)))))

(define (calc-GC-cont2 segm)
   (/ (length (filter GC? segm)) (length segm)))

(define (calc-GC-cont3 segm)
  (define (calc-GC count gccount segm)
    (if (null? segm)
        (/ gccount count)
        (calc-GC (+ 1 count)
                 (if (GC? (car segm))(+ 1 gccount) gccount)
                 (cdr segm))))
  (calc-GC 0 0 segm))
```

Which would you choose with respect to time complexity?

References

- **H. Abelson, G.J. Sussman, Structure and Interpretation of Computer Programs 2nd ed, The MIT Press, Cambridge, Massachusetts, 2000, Chp: 2.2-2.2.3, pp: 97-124**
- **6.001 Spring 2000: Lecture Notes, lecture 5,6, http://sicp.ai.mit.edu/Spring-2000/lectures/**