CS339: Abstractions and Paradigms for Programming

Programming with Lists

Manas Thakur
CSE, IIT Bombay

Autumn 2025

Quiz 1

- When: **Friday**, August **22**, 8:30-9:15 AM
- Syllabus: Everything until Thursday, August 21st
- Last year's QP would be uploaded, but no guarantee that the pattern would be similar (except that it would be a short exam to test progress)
- Closed book, closed smart devices (bring ID cards)
- Seating arrangement to be informed a day before
- Quiz 2 date also available on the course webpage

shutterstock.com · 413192068

Index a list like an array

Getting the n^{th} element from `l` is same as getting the $(n\text{-}1)^{th}$ element from `(cdr l)`.

```
(define (get n lst)
  (if (= n 0)
      (car lst)
      (get (- n 1) (cdr lst))))
```

But this throws an error with `(get 3 (list 1 2))`

```
(define (get n lst)
  (cond ((null? lst) nil)
        ((= n 0) (car lst))
        (else (get (- n 1) (cdr lst)))))
```

This returns an empty list.

Which one is better? A design decision.

Determine length of a list

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

Length of a list `l` is one plus the length of `(cdr l)`.

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

Iterative version.

It doesn't need to do extra computation after function call. Hence kind of iterative

Now it's getting easier to fathom

- Sum the elements of a list:

Summing a list (`l`) is same as adding the first element of the list (`car l`) to the sum of the remaining list (`cdr l`).

```
(define (sum lst)
  (if (null? lst)
      0
      (+ (car lst) (sum (cdr list)))))
```

Recursive definitions are often easier to spell out and understand!

Form sublists

- Take the first n elements of a list:

```
(define (take n lst)
  (if (= n 0)
      nil
      (cons (car lst) (take (- n 1) (cdr lst)))))
```

Get used to “spelling out” the recursive definitions.

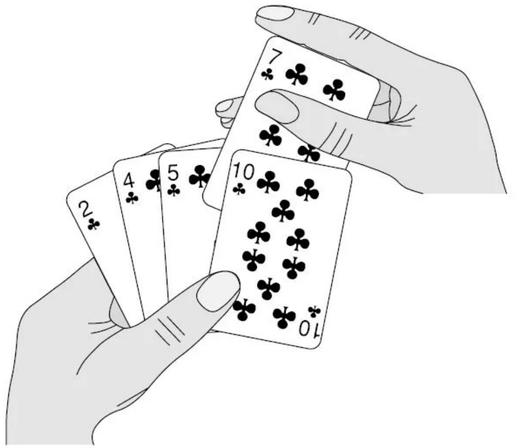
- Drop the first n elements of a list:

```
(define (drop n lst)
  (if (= n 0)
      lst
      (drop (- n 1) (cdr lst))))
```

Let's get complicated! How about Insertion Sort?

- Let's insert a number at its right place in a sorted list:

```
(define (insert num lst)
  (cond ((null? lst) (cons num nil))
        ((<= num (car lst)) (cons num lst))
        (else (cons (car lst) (insert num (cdr lst))))))
```



- Job's almost done:

```
(define (isort lst)
  (if (null? lst)
    nil
    (insert (car lst) (isort (cdr lst)))))
```

- Test:

```
> (define l (list 56 47 89 23 100 27 38))
> (isort l)
```

Interested in comparing this with other languages?

But I like trees more!

- Let's make a binary search tree (BST) node:

```
(define (make-tree datum left right)
  (list datum left right))
```

- Once we got a "constructor", we need some "selectors":

```
(define (datum t) (car t))
(define (left-tree t) (cadr t))
(define (right-tree t) (caddr t))
```

What if this was `cddr`?

Our tree is currently empty!

- Inserting an element in a BST:

```
(define (insert e t)
  (cond ((null? t) (make-tree e nil nil))
        ((= e (datum t)) t)
        ((< e (datum t)) (make-tree (datum t)
                                    (insert e (left-tree t))
                                    (right-tree t)))
        (else (make-tree (datum t)
                         (left-tree t)
                         (insert e (right-tree t))))))
```

- Test:

```
> (define t (make-tree 4 nil nil))

> (define t1 (insert 6 (insert 3 (insert 5 t))))
```

Practice makes a *tree* perfect

- Checking for presence of an element

```
(define (elem? e t)
  (cond ((null? t) #f)
        ((= e (datum t)) #t)
        ((< e (datum t)) (elem? e (left-tree t)))
        (else (elem? e (right-tree t)))))
```

- Inorder traversal:

```
(define (inorder-try t)
  (if (null? t)
      nil
      (cons (inorder-try (left-tree t))
            (cons (datum t)
                  (inorder-try (right-tree t))))))
```

What's the problem with this?

Properly formatted inorder traversal of a tree

- Concatenate two lists:

```
(define (concat l1 l2)
  (if (null? l1)
      l2
      (cons (car l1) (concat (cdr l1) l2))))
```

- Inordo perfectum:

```
(define (inorder t)
  (if (null? t)
      nil
      (concat (inorder (left-tree t))
              (concat (list (datum t))
                      (inorder (right-tree t))))))
```

Why this?

Can we now have a nice tree-sort procedure?

- Form a BST out of a list, and then print its inorder traversal!

```
(define (list2tree l)
  (define (l2t-iter t l)
    (if (null? l)
        t
        (l2t-iter (insert (car l) t) (cdr l))))
  (l2t-iter (make-tree (car l) nil nil) (cdr l)))

(define (tree-sort l)
  (inorder (list2tree l)))

> (define l (list 56 47 89 23 100 27 38))

> (tree-sort l)
```

Even more fun in the next class :-)