A crash course on LISP

Functional programming

- Definition From the "comp.lang.functional FAQ"

  Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands.

  The expressions in these language are formed by using functions to combine basic values.

  A functional language is a language that supports and encourages programming in a functional style.

- Why functional programming matters?

  http://www.cs.chalmers.se/~rjmh/Papers/whyfp.pdf

Lisp resources

- CMU AI repository

  http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/0.html
- Reference book:

  Common Lisp the Language, 2nd edition by **Guy L. Steele**, Thinking Machines, Inc. Digital Press 1990 paperbound 1029 pages ISBN 1-55558-041-6

  http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/mirrors.html
- Lisp FAQ

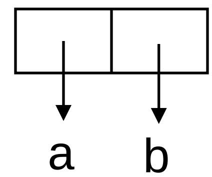
  http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/faq/0.html
- Lisp tutorials

  http://mypage.iu.edu/~colallen/lp/
- Code taken from

  ANSI Common Lisp, by Paul Graham.

Definitions

- **<u>Cons:</u>** A **cons** is a pair of pointers, the first of which is the **car** and the second one is the **cdr.**



- **<u>Atom:</u>**
  - » Basic lisp entity: the empty list, the constant atom t, a symbol, a number (real (rational (ratio integer) float) complex), a vector, an array, a character, a string
  - » Everything that is not a **cons**

    (defun our-atomp (x) (not (consp x)))
- **<u>List</u>**:
  - » An ordered collection of atoms or lists (the elements of the list)
  - » A list is either **nil** or a **cons**

    (defun our-listp (x) (or (null x) (consp x)))
- **<u>Expression:</u>** An atom or a list.
- **<u>Form:</u>** An expression to be evaluated by the Lisp interpreter.
- **<u>Evaluation:</u>**
  - » If the form is an atom: The value of the atom.
  - » If the form is a list: The value of a function evaluation
    - – The first element of a list is interpreted as the name of the function to be evaluated.
    - – The remaining elements are evaluated and given as the input to the function (prefix notation).

Definitions

- **<u>Proper list:</u>**
  - » A lisp entity susceptible of being constructed with the **list** command.
  - » A proper list is a list that is either **nil** or a **cons** whose **cdr** is a proper list

```
(defun our-proper-listp (x)
  (or (null x)
      (and (consp x)
           (our-proper-listp (cdr x)))))
```

- **<u>Assoc-list (aka alist):</u>**
  - » A list of conses.
  - » Each of these conses represents an association of a given key with a given value
    - – the **car** of each cons is the key
    - – the **cdr** is the value associated with that key

Warning: assoc-lists are slow (linear-time access)

Exercise: Write a function to determine whether an object is an assoc-list

```
(defun our-assoc-listp (x) ...)
```

Conses

cons, car, cdr, consp

» (setf x (cons 'a 'b))

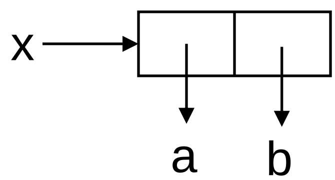
(a.b)

» (car x)

a

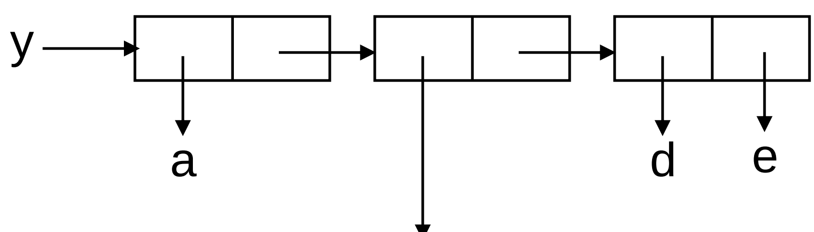
» (cdr x)

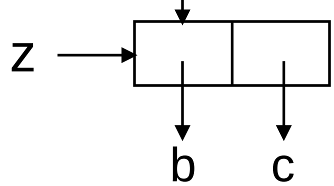
b



» (setf y (cons 'a (cons (cons 'b ' c) (cons 'd 'e))))

(A (B . C) D . E)





» (setf z (car (cdr y)))

(b.c)

» (consp (cdr y))

T

» (consp (cdr z))

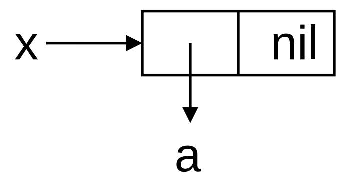
NIL

Lists

cons, car (first) , cdr (rest), list

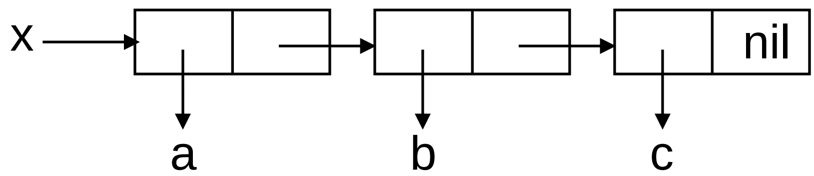
» (setf x (cons 'a nil))

(a)



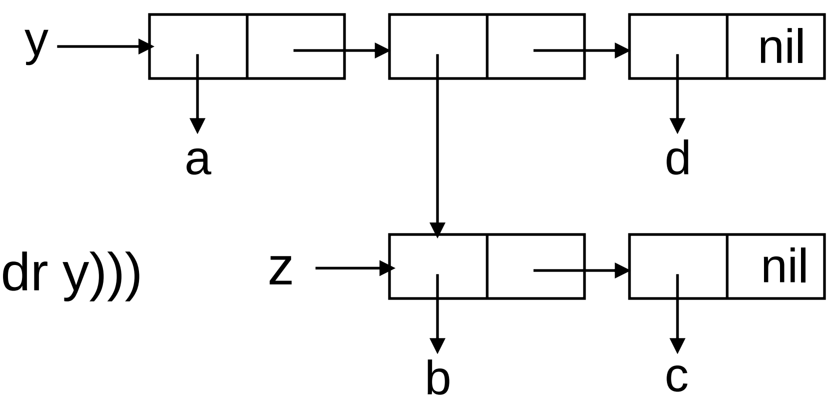
» (setf x (cons (car x) '(b c)))

(a b c)



» (setf y (list 'a (list 'b 'c) 'd))

(a (b c) d)



» (setf z (car (cdr y)))

(b c)

» (eql z (cdr x))

NIL

» (equal z (cdr x))

T

» (eql z (car (cdr y)))

T

Commands for lists (Learn me!, use me!)

- Constructing lists:
  cons, list, append, copy-tree
  copy-list [! only copies cdr's of the elements]
  nconc [! Destructive; macro]
- List properties:
  null, listp [Boolean]
- Lists as conses:
  car (first), cdr (rest), cadr, caddr,caaaar,…,cddddr,nthcdr
  first, second, third,..., tenth, nth, last, rest
- Lists as sets:
  member, member-if, subsetp
  adjoin, union, intersection, set-difference
- Lists as stacks
  push, pop [! destructive; macros]
- Lists as sequences (sequences = vectors + lists)
  length, count
  max, min, find, find-if, position, position-if
  merge
  remove
  delete [! destructive]
  subseq, reverse
  nreverse, sort [! destructive]
  every, some [Boolean]
- Association lists
  assoc

Example: Our-lisp-functions (1)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Home-made lisp functions.
;;;;

(defun our-length (x)
  (if (null x)
      0
    (+ 1 (our-length (cdr x)))))

(defun our-copy-list (lst)
  (if (atom lst)
      lst
      ;;  Watch out: you are only copying the cdr’s
      (cons (car lst) (our-copy-list (cdr lst)))))

(defun our-assoc (key alist)
  (and (consp alist)
       (let ((pair (car alist)))
         (if (eql key (car pair))
             pair
           (our-assoc key (cdr alist))))))
;;;;
;;;; Note the use of recursion
;;;; Do not forget the default case!
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

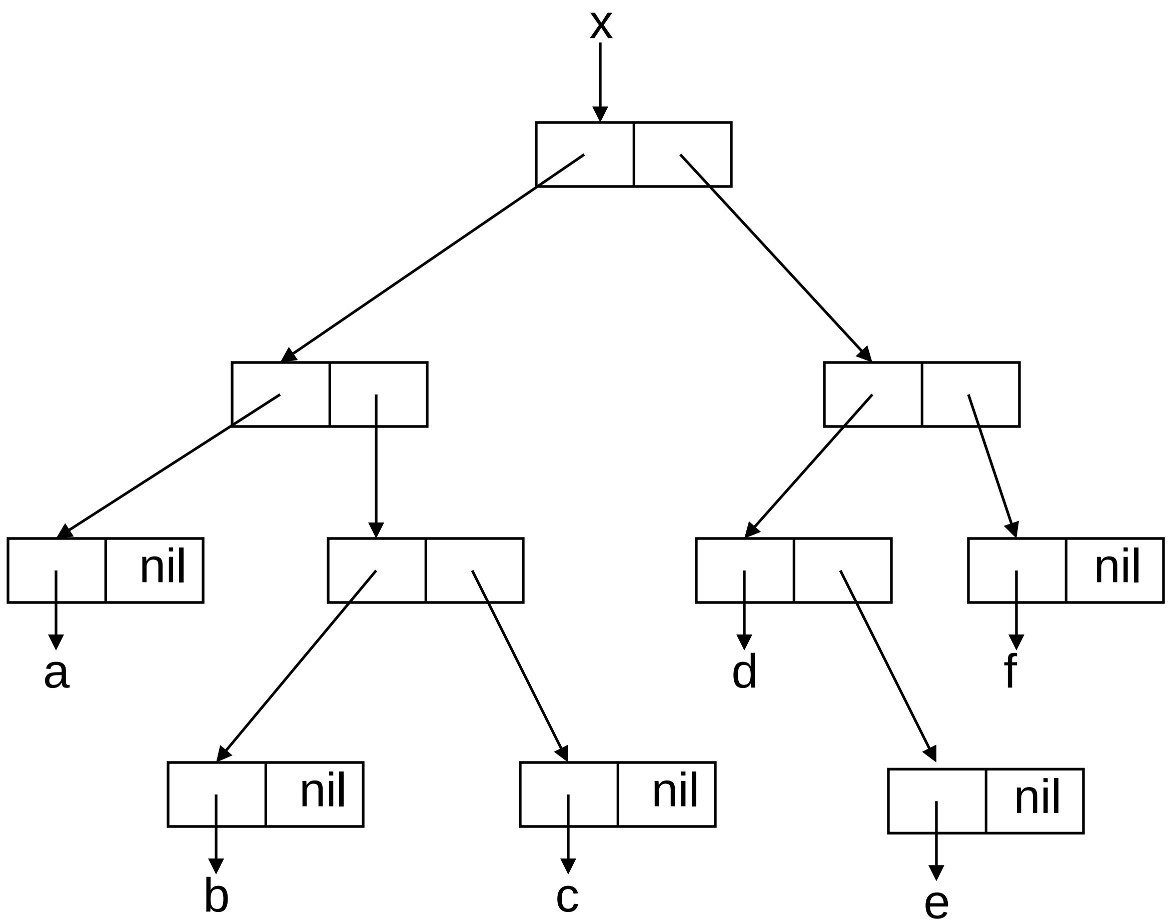
Examples: Our-lisp-functions (2)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;  More home-made lisp functions.
;;;;

(defun our-member (obj lst)
  (cond ((atom lst) nil)
        ((eql (car lst) obj) lst)
        (t (our-member obj (cdr lst)))))
;;;;
;;;; Note the use of recursion
;;;; Do not forget the default case!
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun lst-and (lst)
  (cond
   ((null lst) t)
   ((null (first lst)) nil)
   (t (lst-and (rest lst)))))


(defun lst-or (lst)
  (cond
   ((null lst) nil)
   ((null (first lst)) (lst-or (rest lst)))
   (t t)))
```

Conses and lists as trees

Conses can be thought of as binary trees with the **car** as the left subtree and the **cons** as the right subtree.

» (setf x '(((a) (b) c) (d e) f)



- Functions on trees

  copy-tree

  tree-equal

  subst

Example: Our-tree-functions (1)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Home-made tree lisp functions.

(defun our-copy-tree (tr)
  (if (atom tr)
      tr
      (cons (our-copy-tree (car tr))
            (our-copy-tree (cdr tr)))))

;;;; Compare with our-copy-list:
;;;;  copy-tree copies the car and the cons.
;;;;  copy-list only copies the cons.
;;;;  If some car of the list elements is not an atom,
;;;;  changing (e.g. with setf) some value inside
;;;;  that car in the copy modifies original!

(defun our-substitute (new old tr)
  (if (eql tr old)
      new
      (if (atom tr)
          tr
          (cons (our-substitute new old (car tr))
                (our-substitute new old (cdr tr))))))
```

Example: Our-tree-functions (2)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; More home-made tree lisp functions.


(defun same-shape-tree (tr1 tr2)
  (tree-equal tr1 tr2 :test #'our-true))

(defun our-true (&rest ignore) t)

;;;; (same-shape-tree '(((A) ((B) (C))) ((D (E)) F))
;;;;                  '((((A)) ((B) (C))) ((D (E)) F)))
;;;;
;;;; (same-shape-tree '(((A) ((B) (C))) ((D (E)) F))
;;;;                  '(((1) ((2) (3))) ((4 (5)) 6)))
;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Example: Quicksort on vectors

```
;;;; Quick sort on vectors (vector = one-dimensional array)
(defun quicksort (vec l r)
  (let ((i l)
        (j r)
        (p (svref vec (round (+ l r) 2)))) ; middle element as pivot
    ;;
    ;; Partition vector by swapping elements until all
    ;; elements of the vector lower than the pivot are to the
    ;; to the left of those greater than the pivot
    (while (<= i j)
      (while (< (svref vec i) p) (incf i))
      (while (> (svref vec j) p) (decf j))
      (when (<= i j)
        (rotatef (svref vec i) (svref vec j))
        (incf i)
        (decf j)))
    ;;
    ;; If either of the partitions has two or more elements,
    ;; apply quicksort recursively to the partitions.
    (when (> (- j l) 1) (quicksort vec l j))
    (when (> (- r i) 1) (quicksort vec i r)))
  vec)
;;  Example: (quicksort (vector   1 -2  3 -4) 0 3)
```

Exercise: Implement quicksort with lists

Some LISP functions (1)

Consider the following LISP functions:

- » COUNT / COUNT-IF / COUNT-IF-NOT
- » FIND / FIND-IF / FIND-IF-NOT
- » REMOVE / REMOVE-IF / REMOVE-IF-NOT

- <u>REMOVE / REMOVE-IF</u>

  - » (remove <elmnt> <list> [:test <equality-test>] [:key <key>])
    - – (remove 2 '(1 2 3 4 2 4 5 2))
      (1 3 4 4 5)
    - – (remove 2 '((1 a) (2 b) (3 c) (4 d) (2 f)))
      ((1 A) (2 B) (3 C) (4 D) (2 F))
    - – (remove 2 '((1 a) (2 b) (3 c) (4 d) (2 f)) :key #'car)
      ((1 A) (3 C) (4 D))
    - – (remove '(2 b) '((1 a) (2 b) (3 c) (4 d) (2 f)))
      ((1 A) (2 B) (3 C) (4 D) (2 F))
    - – (remove '(2 b) '((1 a) (2 b) (3 c) (4 d) (2 f)) :test #'eql)
      ((1 A) (2 B) (3 C) (4 D) (2 F))
    - – (remove '(2 b) '((1 a) (2 b) (3 c) (4 d) (2 f)) :test #'equal)
      ((1 A) (3 C) (4 D) (2 F))

  - » (remove-if[-not] <predicate><list>)
    - – (remove-if #'oddp '(1 2 3 4 2 4 5 2))
      (2 4 2 4 2)

Some LISP functions (2)

- FIND / FIND-IF
  - » (find <element> <list> [:test <equality-test>] [:key <key>])
    - (find 2 '(1 2 3 4 2 4 5 2))
      2
    - (find 2 '((1 a) (2 b) (3 c) (4 d) (2 f)))
      NIL
    - (find 2 '((1 a) (2 b) (3 c) (4 d) (2 f)) :key #'car)
      (2 b)
    - (find '(2 b) '((1 a) (2 b) (3 c) (4 d) (2 f)))
      NIL
    - (find '(2 b) '((1 a) (2 b) (3 c) (4 d) (2 f)) :test #'equal)
      (2 b)
  - » (find-if[-not] <predicate><list>)
    - (find-if #'oddp '(1 2 3 4 2 4 5 2))
      1
- COUNT / COUNT-IF
  - » (count <elmnt> <list> [:test <equality-test>] [:key <key>])
    - (count 4 '(1 2 3 4 2 4 5 2))
      2
  - » (count-if[-not] <predicate><list>
    - (count-if #'oddp '(1 2 3 4 5 6))
      3

Higer order functions (1)

- # ' <fn>: Reference to a function
- **apply**

  Arguments: a function + a collection of arguments the last of which is a list

  Evaluates to : Value of the function applied to the arguments

  » (apply #'+ '(1 2 3))

  » (apply #'+ 1 '(2 3))

  » (apply #'+ 1 2 3 '())
- **funcall**

  Arguments: a function + a collection of arguments

  Evaluates to : Value of the function applied to the arguments

  » (funcall #'+ 1 2 3)

Higer order functions (2)

- **mapcar**

  Arguments: a function + one or more lists

  Evaluates to: List of values resulting from applying the function to each of the elements of the list(s), until some list is exhausted

```
» (mapcar #'> '(1 2 3) '(4 1 2 5))
(NIL T T)
» (mapcar #'sqrt '(1 4 9 16))
(1 2 3 4)
```

- **mapcan:**   (mapcar …) = (apply #'nconc (mapcar …))

```
(setf lst '(NIL 2 NIL 4 NIL 6))

(apply #'nconc (mapcar #'(lambda (x)
                            (if (null x)
                                NIL
                              (list x)))
                  lst))

(mapcan  #'(lambda (x)
                            (if (null x)
                                NIL
                              (list x)))
                  lst)
```

- **maplist**

  Arguments: a function + one or more lists

  Evaluates to : List of values resulting from applying the function to the list(s) and to each of the **cdr**s of the list(s), until some list is exhausted

```
»  (maplist #'(lambda (x y) (append x y)) '(a b c) '(e f))
((A B C E F) (B C F))
```

The lambda function (anonymous functions)

- Notation used by Whitehead y Russell in *Principia Mathematica.*

- Evolution of notation
  - » ^x(x + x) (Alonzo Church, 1941: definition of *lambda calculus)*
  - » Λx(x + x)
  - » λx(x + x)
  - » (lambda (x) (+ x x)) (McCarthy 1958)

- Usage in LISP
  - »
```
(funcall #'(lambda (x) (+ x x)) 3)
6
```
  - »
```
(mapcar #'(lambda (x) (* x x))
    '(1 2 3))
(1 4 9)
```
  - »
```
(mapcar #'(lambda (x y) (list x y))
    '(a b c d) '(1 2 3))
((a 1) (b 2) (c 3))
```

Variable visibility: Lexical scope

- Scope determined by the structure of the code

```
>(let ((x 10)) (defun foo () x))
>(let ((x 20)) foo)
  10
```

- **Lexical closure**: Section of the code where a variable with lexical scope is *visible*

```
> (defun main (z)
   (example-of-scope z 3))

> (defun example-of-scope (x y)
      (append (let ((x (car x)))
                (list x y))
              x
              z))
```

z is lexically invisible

Variable visibility: Dynamic scope

- The LISP keyword `special` is used to denote that a variable has dynamic scope.
- In this example `x` in function *foo* does not refer to the lexically defined variable. It makes reference to any variable x declared as special when the function is evaluated.

```
>(let ((x 10)) (defun foo ()
    (declare (special x)) x))
>(let ((x 20))
    (declare (special x))
    (foo))
     20
```

- Usage: give a global variable global a new value only temporarily.
- very few programming languages have dynamic scope (Lisp, Tcl,...)
- It is difficult to use and debug

Use of let

- Use let to avoid repeating evaluations

WITH REPEATED CODE:

```
(defun foo(x)
  (if ( >  (* x x) 4)
      x
    (* x x)))
```

WITHOUT REPEATED CODE **(preferable)**:

```
(defun foo(x)
  (let ((aux (* x x)))
    (if (> aux 4)
      x
      aux)))
```

Structures

A structure is a composite object that groups related data.

Example: Binary search tree
A binary search tree (BST) is either **nil** or a node whose left and right children are BST's

```
» (defstruct node
          elt
          (l nil)
          (r nil))
```

The following functions are immediately defined

- **make-node** (constructor)
- **node-p** (is … a node?)
- **copy-node** (copy structure)
- **node-elt** (value of **elt** field)
- **node-l** (value of **l** field)
- **node-r** (value of **r** field)

```
» (setf nd1 (make-node :elt 0 ))
» (setf root (make-node :elt 1 :l nd1))
```

Example: Binary-search-trees (1)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Binary search trees.

(defstruct (node (:print-function
                  (lambda(n stream depth)
                    (format stream "#<~A>" (node-elt n)))))
  elt (l nil) (r nil))

(defun bst-insert (obj bst <)
  (if (null bst)
      (make-node :elt obj)
    (let ((elt (node-elt bst)))
      (if (eql obj elt)
          bst
        (if (funcall #'< obj elt)
            (make-node
             :elt elt
             :l (bst-insert obj (node-l bst) <)
             :r (node-r bst))
          (make-node
           :elt elt
           :l (node-l bst)
           :r (bst-insert obj (node-r bst) <)))))))
```

Example: Binary-search-trees (2)

```
(defun print-tree (n)
  (if (null n)
      ()
      (progn
        (format t "~A" (node-elt n))
        (print-tree (node-l n))
        (format t "r")
        (print-tree (node-r n)))))


(defun bst-find (obj bst <)
  (if (null bst)
      nil
      (let ((elt (node-elt bst)))
        (if (eql obj elt)
            bst
            (if (funcall #'< obj elt)
                (bst-find obj (node-l bst) <)
              (bst-find obj (node-r bst) <))))))
```

Example: Binary-search-trees (3)

```
(defun bst-min (bst)
  (and bst
      (or (bst-min (node-l bst)) bst)))

(defun bst-max (bst)
  (and bst
      (or (bst-max (node-r bst)) bst)))

(defun bst-remove (obj bst <)
  (if (null bst)
     nil
     (let ((elt (node-elt bst)))
    (if (eql obj elt)
       (percolate bst)
       (if (funcall #'< obj elt)
      (make-node
       :elt elt
       :l (bst-remove obj (node-l bst) <)
       :r (node-r bst))
          (make-node
       :elt elt
       :l (node-r bst)
       :r (bst-remove obj (node-r bst) <)))))))
```

Example: Binary-search-trees (4)

```
(defun percolate (bst)
  (cond  ((null (node-l bst))
       (if (null (node-r bst))
           nil
           (rperc bst)))
      ((null (node-r bst)) (lperc bst))
      (t (if (zerop (random 2))
        (lperc bst)
            (rperc bst)))))


(defun rperc (bst)
  (make-node :elt (node-elt (node-l bst))
         :l (node-l bst)
         :r (percolate (node-r bst))))

(defun lperc (bst)
  (make-node :elt (node-elt (node-l bst))
         :l (percolate (node-l bst))
         :r (node-r bst)))

;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Recursive programming (1)

- Lists are recursive data structures.
- Recursion is preferred to iteration in LISP
  - » Intuitive and elegant implementation.

- Example: Power of a number

```
(defun our-power (x n)
   (if (= n 0)
        1
        (* x (our-power x (- n 1)))))
```

- Example: Count atoms

```
(defun count-atoms (expr)
   (cond ((null expr) 0)
         ((atom expr) 1)
         (t (+ (count-atoms (first expr))
               (count-atoms (rest expr))))))
```

» `(count-atoms '(a (b c) ((d (e)) f)))`
   `6`

Recursive programming (2)

- Example: Flatten list

```
(defun flatten (lst)
  (cond
   ((null lst) NIL)
   ((atom (first lst))
    (cons
     (first lst)
     (flatten (rest lst))))
   (t (append
       (flatten (first lst))
       (flatten (rest lst))))))

>> (flatten '(a (b c) ((d (e)) f)))
   (a b c d e f)
```

Recursive programming (3)

- Example: Number of sublists in a list (number of times a parenthesis is opened minus 1)

```
(defun number-of-sublists (expression)
  (if (or (null expression) (atom expression))
      0
     (+ (if (atom (first expression)) 0 1)
        (number-of-sublists (first expression))
        (number-of-sublists (rest expression)))))

>> (number-of-sublists '(a (b c) ((d (e)) f)))
    4
```

Recursion vs. Iteration (1)

- Recursive version

```
(defun scalar-product (v1 v2)
  (if (or (null v1) (null v2))
      0
    (+ (* (first v1) (first v2))
       (scalar-product (rest v1) (rest v2)))))
```

- Iterarive version with mapcar

```
    defun scalar-product (v1 v2)
        (apply #'+ (mapcar #'* v1 v2)))
```

ERRONEOUS USE: Iterative version with dolist

```
    (defun scalar-product (v1 v2)
      (let ((sum 0))
        (dotimes (i (length v1))
          (setf sum
            (+ sum (* (nth i v1)
                      (nth i v2)))))
        sum))

    (scalar-product '(2 3) '(4 5))
```

Recursion vs. Iteration (2)

Our version of remove-if

- Mapcar version:

```
(defun our-remove-if (predicate lst)
  (remove NIL
          (mapcar #'(lambda (element)
                      (unless
                         (funcall predicate element)
                       element))
    lst)))
```

- Mapcan version:

```
(defun our-remove-if (predicate lst)
  (mapcan #'(lambda (element)
              (unless
                 (funcall predicate element)
                (list element)))
    lst))
```

```
>> (our-remove-if #'evenp '(1 2 3 4))
   (1 3)
```

Recursion vs. Iteration (3)

- Recursive version:

```
(defun our-remove-if (predicate lst)
  (cond
   ((null lst) nil)
   ((funcall predicate (first lst))
    (our-remove-if predicate (rest lst)))
   (t (cons (first lst)
            (our-remove-if predicate (rest lst))))))

>> (our-remove-if #'evenp '(1 2 3 4))
```

- Iterative version (DO NOT USE):

```
(defun our-remove-if (predicate lst)
 (let (aux)
  (dolist (i lst aux)
    (unless (funcall predicate i)
      (setf aux (append aux (list i)))))))

(our-remove-if #'evenp '(1 2 3 4))
```

Mapcar + recursion

- Maximum depth of a list

```
(defun max-depth (expression)
  (cond
   ((null expression) 0)
   ((atom expression) 1)
   (t (+ 1
         (apply #'max (mapcar #'max-depth
                              expression))))))

>> (max-depth '1)
    1
>> (max-depth '(1))
    2
>> (max-depth '((1 (2 (3))) 4))
    5
```

Tail recursion (1)

Recursion is sometimes not efficient

Example: Program to generate Fibonacci numbers

fibonacci(0) = 0

fibonacci(1) = 1

fibonacci(n) = fibonacci(n-1)+fibonacci(n-2); $n \geq 2$

```
;;;  Recursive: clear but inefficient
(defun fibonacci-recursive (n)
  (cond
   ((<= n 0) 0)
   ((= n 1) 1)
   (t (+ (fibonacci-recursive (- n 1))
         (fibonacci-recursive (- n 2))))))

;;;  Iterative: efficient but unclear
(defun fibonacci-iterative (n)
  (if (<= n 0)
      0
    (do ((i n (- i 1))
         (f1 1 (+ f1 f2))
         (f2 1 f1))
        ((<= i 2) f1))))
```

Tail recursion (2)

- **Tail recursion** is a special case of **recursion** that can be transformed into an **iteration**.
- **Tail call optimization:**
  - » If the function is tail recursive, the result of the last call can be returned directly to the original caller.
  - » This reduces the amount of stack space used and improves efficiency.

Example: Factorial

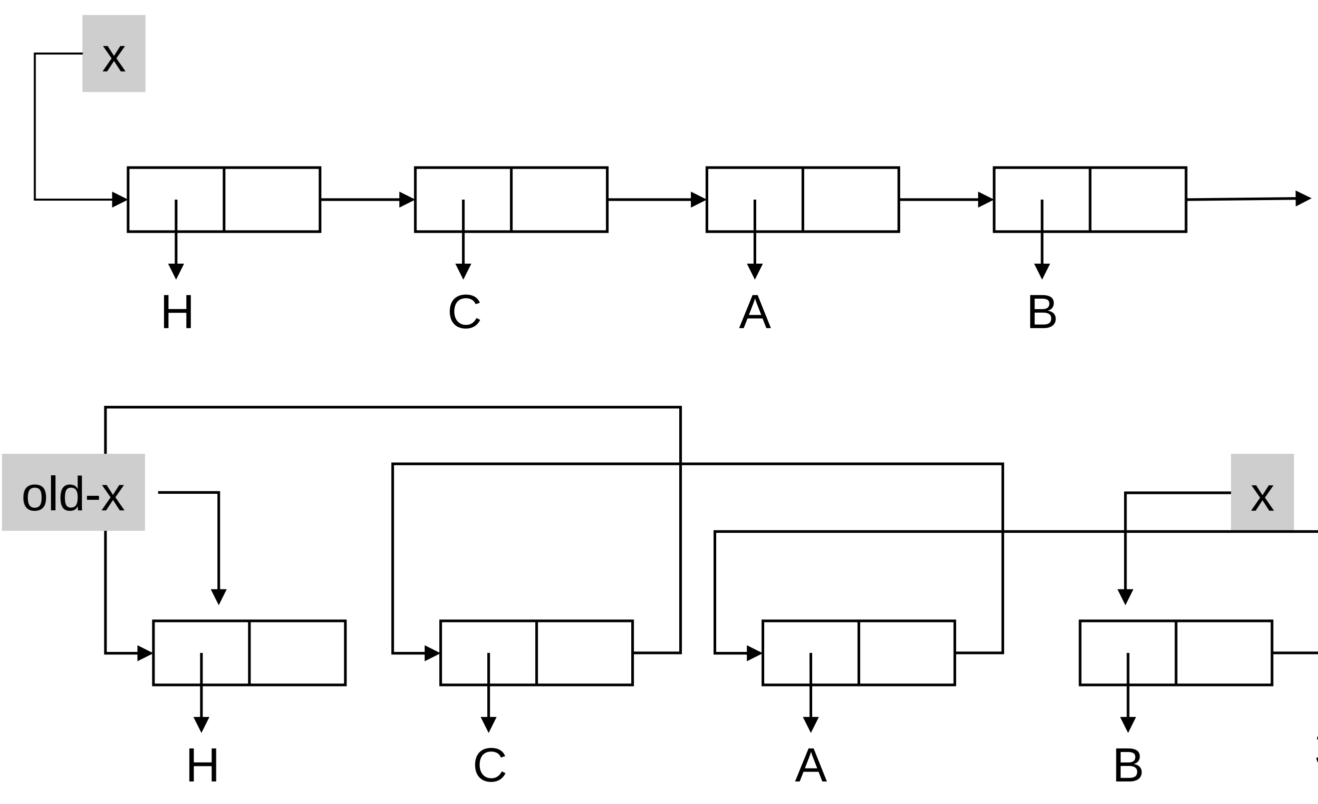
```
;;;  Recursive
(defun factorial-recursive (n)
  (if (<= n 1)
      1
    (* n (factorial-recursive (- n 1)))))

;;;  Tail recursive: More efficient
(defun factorial-tail-recursive (n)
  (if (<= n 1)
      1
    (factorial-aux n 1))
(defun factorial-aux (n accumulator)
  (if (<= n 1)
      accumulator
    (factorial-aux (- n 1) (* n accumulator))))
```

Destructive operations

- Consider the example

```
> (setf x '(H C A B))
    (H C A B)
> (setf old-x x)
    (H C A B)
> (setf x (nreverse old-x))
    (B A C H)
> x
    (B A C H)
>old-x
    (H)
```

- `NREVERSE` destroys the list pointed at by `old-x`. Pointers are reasigned in order not to generate garbage.



`NREVERSE` redefined

```
(defun our-nreverse (n)
   (our-nreverse-1 nil n))

(defun our-nreverse-1 (head tail)
   (let ((residual (cdr tail)))
        (our-nreverse-2 (setf (cdr tail) head)
                             residual
                             tail)))

(defun our-nreverse-2 (drop residual tail)
   (if (null residual)
         tail
         (our-nreverse-1 tail residual)
```

Example: `delete` (destructive) / `remove` (non-destructive)

```
   >>  (setf lst '(a b c d))
       (a b c d)
   >>  (delete 'a lst)
       (b c d)
   >>  lst
       (a b c d)
   >>  (delete 'b lst)
       (a c d)
   >>  lst
       (a c d)
```

Repeat operations using `remove` instead of `delete`

Example: Compression (1)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;;;; Run-length encoding compression
;;;;

(defun compress(x)
  (if (consp x)
      (compr (car x) 1 (cdr x))
      x))

(defun compr (elt n lst)
  (if (null lst)
      (list (n-elts elt n))
    (let ((next (car lst)))
      (if (eql next elt)
          (compr elt (+ n 1) (cdr lst))
        (cons (n-elts elt n)
              (compr next 1 (cdr lst)))))))

(defun n-elts (elt n)
  (if (> n 1)
      (list n elt)
     elt))
```

Example: Compression (2)

```
(defun uncompress (lst)
  (if (null lst)
      nil
    (let ((elt (car lst))
      (rest (uncompress (cdr lst))))
       (if (consp elt)
           (append (apply #'list-of elt) rest)
         (cons elt rest)))))

(defun list-of (n elt)
  (if (zerop n)
      nil
    (cons elt (list-of (- n 1) elt))))
```

;;;
;;; Note the use of recursion + top-down design
;;;
;;

Example: Search in graphs

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Breadth-first-search in graphs

(defun shortest-path (start end net)
   (bfs end (list (list start)) net))

(defun bfs (end queue net)
  (if (null queue)
      nil
      (let ((path (car queue)))
         (let ((node (car path)))
           (if (eql node end)
               (reverse path)
               (bfs end
                     (append (cdr queue)
                               (new-paths path node net))
                      net))))))

(defun new-paths (path node net)
  (mapcar #'(lambda(n)
                       (cons n path))
              (cdr (assoc node net))))
;;;
;;;  recursion +   top-down design (new-paths) + use of queue
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```