Recursion & Iteration

York University CSE 3401

Vida Movahedi

Overview

- Recursion
  - Examples
- Iteration
  - Examples
- Iteration vs. Recursion
  - Example

[ref.: Chap 5,6- Wilensky]

Recursion

- A natural programming style in LISP
- A function is recursive if it calls itself
  - Boundary condition: not recursive
  - Recursive condition: must be a smaller problem to converge

Example- factorial

```
(defun factorial (n)
     (cond   ((zerop n) 1)
             (t  (* n (factorial (1-  n))))  ))
```

- The above code works if n is a positive integer.
- Other numbers (not positive integers) will not reach the boundary condition and a stack overflow error will be encountered.
- A better implementation:

```
(defun factorial (n)
        (cond    ((not  (and (integerp n)  (>= n 0)))   nil)
                 ((zerop n) 1)
                 (t  (* n (factorial (1-  n))))  ))
```

Example- length of list

- Using **cond**:

```
 (defun llist (lst)
    (cond      ((null   lst)        0)
               (t   (1+  (llist (cdr lst))))  ))
```

- Using **if**:

```
 (defun llist (lst)
    (if        (null   lst)        0
               (1+  (llist (cdr lst)))  ))
```

- This one will return 0 if an empty list or an atom (note **nil** is an atom)

```
 (defun llist (lst)
    (if  (atom   lst)   0    (1+  (llist (cdr lst)))  ))
```

Example- member of list

- Test if an element is a member of a list

```
(defun  lmember (e  lst)
     (cond   ((null  lst)                 nil)
             ((equal   e (car lst))       )
             (t                           (lmember  e  (cdr lst)))  ))
```

- Another way of writing above is:

```
(defun  lmember (e  lst)
     (and     lst
              (or      (equal   e (car lst))
                       (lmember  e  (cdr lst))  )))
```

The second argument of **<u>and</u>** will be evaluated, only if the first is evaluated to **true** (i.e. non-nil)

The second argument of **<u>or</u>** will be evaluated, only if the first is evaluated to **false**

Example- member of list

- Test if an element is a member of a list, return the portion of list from the point of first match

```
(defun  lmember (e  lst)
     (cond   ((null  lst)                  nil)
             ((equal   e (car lst))        lst)
             (t                            (lmember  e  (cdr lst)))  ))
```

- Exercise: Write a function that looks for members inside nested lists.

Example- substitution in nested lists

- Function ***lsubst(in out lst)*** substitutes every occurrence of *out* with *in* in *lst*, which can be a list or an atom.
  - e.g. (lsubst ‘a ‘x ‘(b (x x) x)) will evaluate to (B (A A) A)

```
(defun lsubst (in  out  lst)
   (cond
         ((equal  out  lst)        in)        ; if lst is out, return in
         ((atom  lst)              lst)       ; otherwise if atom, no change
                                          ; otherwise two recursions cons’ed
         (t      (cons     (lsubst  in  out  (car  lst))
                           (lsubst  in  out  (cdr  lst))   ))  ))
```

- Exercise: Change the function definition to only substitute if *lst* is a list (no change if it is an atom)

Iteration

- Iteration:
  - A loop, to be executed repeatedly
  - Boundary condition (or terminating condition)
  - A return value upon termination
  - Index variables, their initial value, and the modification rule upon each iteration
- Unlike recursion, we need special functions, such as **do** to implement iteration

Iteration- Do

- General form

```
(do
    ( (var1   val1   rep1)
      (var2   val2   rep2)...)
    exit-clause
    form1   form2   ...)
```

  - In which exit-clause can be nil or in the form of
    (test test-form1 test-form2 ...)

1. Assign all *vari* with corresponding (evaluated) *vali* in parallel.
2. Examine exit-clause. If nil, return nil as value of do (and stop). Otherwise, if *test* evaluates to true, evaluate *test-formi* in order. Return the value of the last form as the value of do (and stop).
3. If test evaluates to false, evaluate *formi* in order.
4. Assign all *vari* with corresponding (evaluated) *repi* in parallel.
5. Go to step 2.

Example

- Find length of list

  For example :

  ```
  > (dolength  ‘(x y z))
  3

  (defun dolength(lst)
     (do
         (  (tlst  lst   (cdr tlst))
            (sum  0  (1+  sum)))
         ( (atom  tlst)  sum )  ))
  ```

Two index variables:
tlst and sum
They are just like formal parameters: local to <u>do</u>.

Terminating condition:
when the list is an atom
(including nil)
Value of sum is returned upon termination.

Examples

- Use **do** to return a list which is the same as a list lst1 without the first n elements
  - assuming its length is greater than n

```
(do  ((x n (1- x)) (lst2  lst1 (cdr lst2)))
     ( (zerop x) lst2))
```

- Use **do** to return a list of numbers from 1 to n

```
(do  ( (m n (1-  m))  (x  nil  (cons m x)))
     ( (zerop m) x))
```

Do vs. Do*

- **do**: evaluates all *vali* first and assigns index variables in parallel

```
> (setq n 3)
3

> (do ( (m n (1-  m))  (x  nil  (cons m x)))  ( (zerop m) x))
(1  2  3)
```

- **do***: Evaluation of *vali* and assignment to *vari* are done in sequential order

```
> (do*  ( (m n (1-  m))  (x  nil  (cons m x)))  ( (zerop m) x))
(0  1  2)
```

Iteration- other functions

- **dolist**: iterates over elements of a list

  (dolist (var list-val return-val) form1 form2 ...)

  – In each iteration, *val* is assigned with a value from list of values *list-val,*

  – Loops over *formi*, until all done.

  – Then *return-val* is returned.

- **dotimes**: iteration over integer values up to a limit

  (dotimes (var stop-val return-val) form1 form2 ...)

  – Initializes *var* to **0**,

  – In each iteration *formi* are evaluated

  – *var* is increased by 1, until it reaches *stop-val*, at which point *return-val* is returned.

Example

- Searching for a certain element in a given list

```
(setq  mylist  ‘(1 2 3 4 5))
(setq  srch  2)

(dolist  (i  mylist  nil)
     (cond  ((equal  i  srch)  (return  t))))
```

  - What will be returned in above case?

  Answer. T will be returned, since 2 exists in mylist.

  - **Note**: dolist goes through elements of the list without the need for us to explicitly use **car** and **cdr**

Example

- Delete the first n items from a list

```
(setq   mylist   '(1 2 3 4 5))
(setq   n   2)

(dotimes  (i   n  mylist)
          (setq  mylist  (cdr mylist)))
```

  - What will be returned in above case?

Answer. (3 4 5) will be returned.

Iteration vs. Recursion

Example: Reversing a list

- Using iteration:

```
(defun do-rev(lst)
    (do ((x lst (cdr x)) (result nil (cons (car x) result)))
    (null x) result)))
```

- Using recursion:
  - Cannot add to the end of a list
  - We therefore use an extra variable (accumulator)
  - More overhead due to recursive calls

```
(defun rev2 (lst acc)
    (cond    ((null lst) acc)
             ( t (rev2  (cdr lst) (cons (car lst) acc))) ))
(defun reverse(lst) (rev2  lst  nil))
```