# Sample Lisp Programs

;  
;  
; Note: By commenting out explanations, you can easily convert this file  
; into a Lisp program file so that you can load it and run it.  
;  
;  
;Example 1.  
;  
;Let's convert the following definition to a Lisp function  
;  
; reverse(L)  
; = if null(L) then L  
; else append(reverse(r(L)), cons(f(L), ()))  
  
  
(defun reverse (L)  
 (if (null L)  
 L  
 (append (reverse (cdr L))  
 (cons (car L) nil))  
 )  
)  
  
  
;Example 2.  
;  
;Let's convert the following definitions to Lisp functions  
  
; append(L1,L2)  
; = if null(L1) then L2  
; else append(removeLast(L1), cons(last(L1), L2))  
;  
;  
: removeLast(L)  
; = if null(r(L)) then ()  
; else cons(f(L), removeLast(r(L)))  
;  
; last(L)  
; = if null(r(L)) then f(L)  
; else last(r(L))  
  
  
(defun xappend (L1 L2)  
 (if (null L1)  
 L2  
 (xappend (removeLast L1) (cons (last L1) L2))  
 )  
)  
  
  
(defun removeLast (L)  
 (if (null (cdr L))  
 nil  
 (cons (car L) (removeLast (cdr L)))  
 )  
)  
  
(defun last (L)  
 (if (null (cdr L))  
 (car L)  
 (last (cdr L))  
 )  
)  
  
  
;Example 3. Multiple Recursion   
;  
; By this example we want to show that one should not try to program recursion  
; over a number of lists in one shot. Instead, decompose the problem to   
; smaller ones and use recursion over a single list at a time.  
;  
; Given two lists, we want to define a function called cartesian,  
; that gives all the pairs corresponding to cartesian product; e.g.  
;  
;  
; (cartesian '(a b) '(d e)) ==> ((a d) (a e) (b d) (b e))  
;  
; We will define cartesian using the following function  
;  
; (pair m N):  
; it pairs m with each element in N  
;  
; E.g.  
; (pair 'a '(1 2 3)) => ((a 1) (a 2) (a 3))  
  
  
(defun pair (m N)  
 (if (null N)  
 NIL  
 (cons (cons m (cons (car N) nil))  
 (pair m (cdr N)))  
 )  
)  
  
  
; Now we are ready to define cartesian:  
  
(defun cartesian (M N)  
 (if (null M)  
 NIL  
 (append (pair (car M) N)  
 (cartesian (cdr M) N))  
 )  
)  
  
; Test it.  
  
>(cartesian '(a b) '(d c e))  
((A D) (A C) (A E) (B D) (B C) (B E))

# Lisp Built-in Functions and Special Forms

Load a file:

>(load "file-name")

### 1. The role of NIL, and Truth Values in Lisp

NIL is a very special symbol in Lisp. It represents an empty list and is considered an atom. So,

>(atom nil)  
T  
  
>(null nil)  
T

NIL also represents False in Lisp. E.g.

>(> 3 4)  
NIL

In fact, NIL represents False and anything else represents True. That is, in Lisp anything except NIL is considered True. E.g.

>(if 3 4 5)  
4  
  
; 3 is considered True  
  
>(if 'a 4 5)  
4  
  
; the symbol a is considered True.

## 2. Let

"Let" is useful if we want to have the same expression in a number of places, e.g.,

>(let ((x 3) (y 4)) (\* (+ x y) x))  
21

Of course, this is the same as

>(\* (+ 3 4) 3)  
21

Then why do we need "let"?  
  
Just imagine that x is bound to a long expression and there are many occurrences of x .... If you don't use "let" you would have to write the same long expression many times.  
  
More importantly, if an expression is needed in a number of places, without using a let you would have the same expression evaluated many times. This is very inefficient and the program is difficult to read--the same expression must be identified by the person reading your program in a number of different places.

This identification is made easy by using a let.

E.g.

>(let ((x (+ 4 2)) (y (+ 3 2))) (\* x x y y))  
900

Without using let, you would have written

>(\* (+ 4 2) (+ 4 2) (+ 3 2) (+ 3 2))  
900

(Yes, + and \* in Lisp are generic functions for one or more arguments!)

In programming, the result of a function application can be "stored" by "let" locally and used in different places.

E.g.

With the definition

(defun f (x) (\* 5 x))  
  
> (let ((y (f 3))) (cons y (cons (+ y 1) (cons (+ y 2) nil))))  
(15 16 17)

However, you should not think of let as something similar to an assignment statement in C or Pascal. Let only allows a name to be bound to an expression. You can only use it within an appropriate scope. Consider

(let ((x 1)) (let ((x (+ x 1))) x))  
 | |  
 -------------------

This is a nested let expression. The occurrence of x in (+ x 1) is bound to the outside x. This expression is equivalent to

(let ((x 1)) (let ((y (+ x 1))) y))

That is, you should avoid using the same name in different scopes just to confuse yourself or anyone who reads your program. The above expression should be indented too.

(let ((x 1))   
 (let ((y (+ x 1))) y))

Should you find a nested let expression is needed in your code, you may find let\* useful.

The let\* special form is just like let except that it allows values to reference variables defined earlier in the let\*. For example,

> (setq x 7)  
7  
> (let\* ((x 1)  
 (y (+ x 1)))  
 y  
 )  
2

The form

(let\* ((x a)  
 (y b))  
 ...  
)

is equivalent to

(let ((x a))  
 (let ((y b))  
 ...  
) )

## 3. The difference between Lisp functions eq and equal

In Lisp:

>(eq '(a b) '(a b))  
NIL

That is, eq doesn't test the equality of s-expressions or lists. We can easily define a function to do that:

(defun equal (S1 S2)  
 (if (atom S1)  
 (if (atom S2)  
 (eq S1 S2)  
 NIL)  
 (if (atom S2)   
 NIL  
 (if (equal (car S1) (car S2))  
 (equal (cdr S1) (cdr S2))  
 NIL  
 )  
 )  
 )  
)  
  
  
>(equal '(a b) '(a b))  
T  
  
>(equal  
'(a (b) (((c)))) '(a (b) ((c))))  
NIL  
  
>(equal '(a (b) (((c)))) '(a (b) (((c)))))  
T

A shorter way to define equal uses the fact that (eq S1 S2) will be nil whenever S2 is not an atom. So we can also write the equal function as:

(defun equal (S1 S2)  
 (if (atom S1)   
 (eq S1 S2)  
 (if (atom S2)  
 nil  
 (and (equal (car S1) (car S2))  
 (equal (cdr S1) (cdr S2))  
 )   
 )   
 )   
)

## 4. Logic Connectives and some other predicates

(and E1 ... En)  
(or E1 ... En)  
(not E)  
  
(numberp x) ; true if x is a number (integer or real)  
  
(atom x) ; true if x is an atom  
  
(null x) ; true if x is an empty list

## 5. Cond (for Conditional)

The Lisp function cond is one of the most useful. Its general form is:

(cond (P1 S11 S12 ... S1m)  
 (P2 S21 S22 ... S2m)  
 ......  
 (Pn Sn1 Sn2 ... Snm)  
)

If P1 is true then evaluate S11, S12, ... and S1m and return the result of evaluating S1m; otherwise if P2 is true, evaluate S21, S22, ... and S2m and so on.

Often we only need the form

(cond (P1 S1)  
 (P2 S2)  
 ...  
 (T Sn)  
)

This is sufficient to allow us to define case statement: if P1 is true then S1; otherwise if P2 is true then S2, ....

For example, for n = 4, an equivalent expression by if-then-else is:

(if P1  
 S2  
 (if P2  
 S2  
 (if P3  
 S3  
 S4  
 )  
 )  
)

As you can see, this is awful--difficult to read and understand.

Example:

Let's redefine the equal function

(defun xequal (S1 S2)  
 (cond  
 ((and (atom S1) (atom S2)) (eq (S1 S2)))  
 ((and (atom S1) (not (atom S2))) NIL)  
 ((and (atom S2) (not (atom S1))) NIL)  
 (t (and (xequal (car S1) (car S2))  
 (xequal (cdr S1) (cdr S2))))  
 )   
)

Again, we could also write a shorter version by using the properties of eq:

(defun xequal (S1 S2)  
 (cond  
 ((atom S1) (eq (S1 S2)))  
 ((atom S2) NIL)  
 (t (and (xequal (car S1) (car S2))  
 (xequal (cdr S1) (cdr S2))))  
 )  
)

Last modified: Thursday, 9 June 2011, 10:24 AM

# Binary Tree: An Example of Representation

; Representation in Lisp  
;  
; A central issue in Lisp programming concerns the representation of  
; the objects to be manipulated. Here is an example.  
;   
;  
; We want to build a package that consists of some functions   
; operating on binary trees. The first question is how binary trees  
; are represented by lists. Here is one possibility.  
;   
;  
; A binary (search) tree is represented as  
; nil tree by nil  
; a tree with one node by (nil value nil)   
; where value is the node value  
; and in general by (left\_subtree value right\_subtree)  
;  
; For example, the following tree  
;   
;  
 4  
; / \  
; 2 6  
; / \  
; 5 8  
;  
; is represented by   
;  
;  
; ((nil 2 nil) 4 ((nil 5 nil) 6 (nil 8 nil)))  
;  
;  
; We define a function that, given a binary tree and an integer,  
; returns a binary tree with integer inserted   
;  
; The following example illustrates the idea  
; of Abstract Data Types (ADT). In an ADT, we have a underlying  
; data structure, in this example, binary trees, and a number of   
; "access functions", which depend on the underlying data structure.  
; But user programs should all be independent of the underlying data   
; structure. This is achieved by calling only access functions.   
;   
  
(defun insert (Tr Int)  
 (if (isEmptyTree Tr)  
 (create\_tree (create\_empty\_tree) Int (create\_empty\_tree))  
 (if (eq (node\_value Tr) Int)  
 Tr  
 (if (< (node\_value Tr) Int)  
 (create\_tree (left\_subtree Tr)  
 (node\_value Tr)  
 (insert (right\_subtree Tr) Int)  
 )  
 (create\_tree (insert (left\_subtree Tr) Int)  
 (node\_value Tr)  
 (right\_subtree Tr)  
 )  
 )  
 )  
 )  
)  
  
;  
; The following are some of the access functions for binary trees  
;  
  
(defun isEmptyTree (Tr)  
 (null Tr)  
)  
  
(defun create\_empty\_tree ()  
 nil  
)   
  
(defun create\_tree (L N R)  
 (cons L (cons N (cons R nil)))  
)   
  
(defun node\_value (Tr)  
 (car (cdr Tr))  
)  
  
(defun left\_subtree (Tr)  
 (car Tr)  
)  
  
(defun right\_subtree (Tr)  
 (car (cdr (cdr Tr)))  
)  
  
;  
;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
;  
; Nested if-expressions can be difficult to understand.   
; Essentially, we have a number of cases to deal with; the conditional cond  
; in this case is more natural and easier to understand; it gives   
; a clearer structure and intention of the program.  
;   
; In general, one should not have more than two ifs nested.  
  
(defun xinsert (Tr Int)  
 (cond ((null Tr) (create\_tree (create\_empty\_tree)   
 Int   
 (create\_empty\_tree)))  
 ((eq (node\_value Tr) Int) Tr)  
 ((< (node\_value Tr) Int) (create\_tree (left\_subtree Tr)  
 (node\_value Tr)  
 (xinsert (right\_subtree Tr) Int)))  
 (T (create\_tree (xinsert (left\_subtree Tr) Int)   
 (node\_value Tr)  
 (right\_subtree Tr)))  
 )  
)