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 (search) 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  
; under a particular representation of 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)))  
 )  
)