LISP Part 2

|  |  |
| --- | --- |
| **Debugging LISP Functions**  Common LISP provides a (TRACE *funcName*) macro which turns on tracing of that function. TRACE must be executed after the function is defined. When the specified function is executed, TRACE causes it to print   * a line indicating what is passed * a line indicating the result of the function call. | > (trace memset)  ;; tracing function MEMSET.  (MEMSET)  > (memset 'x '(w x y))  1. Trace: (MEMSET 'X '(W X Y))  2. Trace: (MEMSET 'X '(X Y))  2. Trace: MEMSET ==> T  1. Trace: MEMSET ==> T  T  > (memall 'X '( (Y X) Z)) // have to define function first  1. Trace: (MEMALL 'X '((Y X) Z))  2. Trace: (MEMALL 'X '(Y X))  3. Trace: (MEMALL 'X 'Y)  3. Trace: MEMALL ==> NIL  3. Trace: (MEMALL 'X '(X))  4. Trace: (MEMALL 'X 'X)  4. Trace: MEMALL ==> T  3. Trace: MEMALL ==> T  2. Trace: MEMALL ==> T  1. Trace: MEMALL ==> T  T |
| **Assignment Functions**  (**setf** *variable1 expr1 variable2 expr2 …* )  assigns the value of *exprk*  to *variablek*.  (set *variableExpr*) - assigns the value of *expr* to the variable resulting from the *variableExpr.* | > (setf A '(x y))  (X Y)  > (setf B A)  (X Y)  > (setf x 'bark y 'meow)  MEOW  > (setf D 'B)  B  > (print (list a b d))  ((X Y) (X Y) B)  ((X Y) (X Y) B)  > (set D '(h e l l o)) ;;; notice this is SET not SETF  (H E L L O)  > (print B)  (H E L L O)  (H E L L O)  > (set (car A) 'UTSA)  UTSA  > X  UTSA |
|  |  |
| **LET, LET\*, and Local Variables**  The LET and LET\* functions allow creation of variables which have scope only during that function.  (**let** ( (*var1 expr1*) (*var2 expr2*) … (*varN exprN*))  *bodyExpr1 … bodyExprN* )  (**let\*** ( (*var1 expr1*) (*var2 expr2*) … (*varN exprN*)  *bodyExpr1 … bodyExprN*)  **let** and **let\*** create local variables *var1* thru *varN.* If the corresponding *exprk* is present it assigns its value to the variable. Those local variables can be used in each *bodyExpr*. The result of let is the value of the last *bodyExpr*.  What is the difference between **let** and **let\***? The variables of **let** are not in scope until *bodyExpr1.*  **let** allows the compiler to evaluate each *exprk* independently and in any order. This means that *expr2* doesn't know about *var1*.  With **let\***, the scope of each *vark* is known to all expressions after *exprk*. | > (let ((A 10) (B 4))  (print (list 'diff= (- a b)))  (print (list 'sum= (+ a b)))  )  diff= 6  sum= 14  sum= 14  > (setf A 10) ;;; sets the global variable A to 10  > (let ((A 5) (B (\* A 3))) ;;; local A isn't known for B assign  (print (list 'let A B))) ;;; locals A and B are known  (LET 5 30)  (LET 5 30)  > (let\* ((A 5) (B (\* A 3))) //as soon as var is encountered = known(fixes problem above ex has)  (print (list 'let\* A B)))  (LET\* 5 15)  (LET\* 5 15)  > A  10 |
| **PROG, and PROG\***  PROG allows sequential execution of multiple statements. It also allows the use of the RETURN function.  (**PROG** ( (*var1 expr1*) (*var2 expr2*) … (*varN exprN*))  *bodyExpr1 … bodyExprN* )  (**PROG\*** ( (*var1 expr1*) (*var2 expr2*) … (*varN exprN*)  *bodyExpr1 … bodyExprN*) | (defun CheckData (data)  (PROG ()  (if (NULL data)  (RETURN NIL) )  (if (NULL (car data))  (RETURN NIL) )  ...  ) )  //prog\* ->it has within its scope var k’s |
| **Iterative Loops - do**  (**do** ( (*var1 expr1*) (*var2 expr2*) … (*varN exprN*))  ( *terminateCondExpr resAction* )  *bodyExpr1*  *…*  *bodyExprN* )  //if TCE-> fall out ret resAction  The **do** function provides an iterative loop that terminates when the *terminateCondExpr* is T, returning *resAction.* Similar to LET, variables can be defined. While the *terminateCondExpr* is NIL, it executes each of the *bodyExprK*.  Additional Notes:   * Just like LET\*, there is also a DO\*. * To reduce the explanation, I didn't show that the *vark* *exprK* lists can also have update expressions for advancing variables. * When the terminateCondExpr is T, it evaluates the *resAction.* Actually, the syntax allows many *resAction* expressions and returns the value of the last one. | > (setf fruits '(apple orange banana))  (APPLE ORANGE BANANA)  > (do ((count 0)(fruitVar fruits) )  ((null fruitVar) count) ;;; when it is NIL the do loop exits  ;;; and returns count  (setf count (1+ count))  (print (list count (car fruitVar)))  (setf fruitVar (cdr fruitVar))  )  (1 APPLE)  (2 ORANGE)  (3 BANANA)  3 |
| **Iterative Loops with Control Variables**  (**dolist**  (*controlVar list resExpr*)  *bodyExpr1*  *…*  *bodyExprN* )  **dolist** iteratively assigns each of the values from list to the specified *controlVar.* On exit, it evaluates and returns *resExpr.* | > (setf numbers '(30 40 20))  (30 40 20)  > (let ((hi 0) (count 0))  (dolist (num numbers hi) ;;; returns the highest number  (setf count (1+ count))  (print (list count num))  (if (> num hi) (setf hi num))  )  )  (1 30)  (2 40)  (3 20)  40 |
| **Defining functions with a variable number of arguments**  When declaring a function, specify &rest before a parameter which will contain the rest of the parameters.  //&rest= keyword  //prog returns nil | This function prints any values not equal to the first value and also prints the number of occurrences of the first value.  > (defun PRINT\_NOT\_EQ (first &rest others)  (PROG ((COUNT 0))  (dolist (item others)  (if (equal item first)  (setf count (1+ count))  (print item) ) )  (print (list 'occ 'of first '= count))  ) )  > (PRINT\_NOT\_EQ 'DOG 'CAT 'DOG 'MONKEY 'DOG 'LION)//evaluates  CAT MONKEY  LION  (OCC OF DOG = 2)  NIL |
| **Defining Macros**  Macros are used to create control constructs and can extend LISP. Effectively, macros translate calls into expanded expressions.  //takes expression and executes it  (**defmacro** *macroName* (*parmList*) *expansionExpr*)  Define a macro named *macroName.* *parmList* is a list of parameters and/or parameter directives. Arguments are passed unevaluated to the macro. The macro decides which arguments to evaluate and how to evaluate them. | ;;; Macro Example #1: setNIL  > (defmacro setNIL (x)//definition not implementation  (list 'setf x nil))//literally “setf” in list  SETNIL  > (setNIL king)//not val king  NIL  > king  NIL  >(defmacro setNIL (x)  `(setf x nil))  SETNIL  >(setf x ‘bat king ‘man)  >(setNIL king) //(setf X NIL)  NIL  >KING  MAN  >X  NIL  >(defmacro setNIL(x)  `(setf ,x NIL) // (setf king nil)  SETNIL  > (macroexpand-1 '(setNIL king))//show macroexpansion  (SETF KING NIL)  T |
| **Macros are initially passed unevaluated arguments** | > (defmacro PRINT\_NOT\_EQ (first &rest others)  (PROG ((COUNT 0))  (dolist (item others)  (if (equal item first)  (setf count (1+ count))  (print item) ) )  (print (list 'occ 'of first '= count))  ) )  > (PRINT\_NOT\_EQ DOG CAT DOG MONKEY DOG LION)//unevaluated?  CAT MONKEY  LION  (OCC OF DOG = 2)  NIL  What would happen if the function is passed those unquoted arguments?  It would attempt to access the variable DOG, CAT,… |
| **Macro Aids**  Many macros are a lot easier to write when backquote, comma, and comma-at are used.  //,a evaluates the var  Initially looking at backquote (it is on the key with tilde on most keyboards), it seems a lot like quote; however, it also understands comma and comma-at.  Within a backquote expression, placing a comma in front of something will cause it to be evaluated.  Also within a backquote expression, placing a comma-at in front of a value will splice its argument into a list. | > '(x y z)  (X Y Z)  > `(x y z)  (X Y Z)  > (setf a 1 b 2)  2  > `(since a is ,a and b is ,b the sum is , (+ a b))  (SINCE A IS 1 AND B IS 2 THE SUM IS 3)//this is a list  //series of tokens in a list->macros  > (setf L '(X Y Z))  (X Y Z)  > `(L is ,L)  (L IS (X Y Z))  > `(L is ,@ L)//got rid of parenthesis(1 level)  (L IS X Y Z) |
| **WHILE Macro for a While Loop**  Suppose we want a while loop added to LISP which is the following syntactically:  (**while** *test bodyExpr1 bodyExpr2 …* )//condition nil -stay  **while** will execute the body expressions while the condition is non-NIL.  Example usage:  Print the numbers 0 to 9 across a line:  (let ((i 0))  (while (< i 10)  (princ i)  (princ " ")  (setf i (1+ i))  )  )  //princ ->print across | ;;; Macro Example #2: WHILE using macro aids and &rest  ;;; using the &rest directive to assign all but the first  ;;; argument to bodies  > (defmacro while (test &rest bodies)  `(do ()  ((**not** , test))//test=stay in//not= opposite  ,@ bodies)  )  WHILE  > (let ((i 0))  (while (< i 10)  (princ i)  (princ " ")  (setf i (1+ i))  )  )  0 1 2 3 4 5 6 7 8 9  NIL |
| **Exercise: LOOPN**  Define a macro loopN which is the following syntactically:  (**loopN** *n bodyExpr1 bodyExpr2 …* )  **loopN** will execute the body expressions *n* times.  Example usage:  Print ten periods:  (loopN 10 (princ ".")) | ;;; Macro Example #3  ;;; this isn't exactly right  > (defmacro loopN (n &rest bodies)  `(do ( (i 0 (+ i 1)) )  ((>= i ,n))  ,@ bodies)  )  LOOPN  > (loopN 10 (princ "."))  ..........  NIL |
| **Noise**  Unfortunately, our solution can cause noise to the user. | ;;; User expects this to print the values 10 thru 14  > (let ((i 10))  (loopN 5  (print i)  (setf i (+ i 1))  )  )  0  2  4  NIL  ;;; we can't expand that using macroexpand-1 since it  ;;; wants the CAR as a macro, but it is a let.  ;;; That let above expands to:  (LET ( (I 10) )  (do ( (i 0 (+ i 1)) )//DO doesn’t SET UP NEW SCOPE  ( (>= i 5) )  (PRINT I)  (SETF I (+ I 1))  )  )  **What happened?**  ?? |
| **Avoid Noise by Using Generated Symbols**  The function (gensym) generates a symbol.  > (defmacro loopN (n &rest bodies)  `(do ( (i 0 (+ i 1)) )  ((>= i ,n))  ,@ bodies)  ) | ;;; Macro Example #4: corrected LOOPN using GENSYM  ;;; Using gensym to generate the variable name,  ;;; we don't have a conflict.  > (defmacro loopN (n &rest bodies)  (let ( (g (gensym)) ) //every time it sees g It replaces is with gensym// “let” gives g its value here  `(do ( (,g 0 (+ ,g 1)) )  ( (>= ,g ,n))  ,@ bodies  )  )  )  LOOPN  > (let ((i 10))  (loopN 5  (print i)  (setf i (+ i 1))  )  )  10  11  12  13  14  NIL |
| **Thought we would be clever and use do\* and remove the let.** In theory, so that g is assigned gensym and then used. | ;;; Macro Example #5: incorrect LOOPN without LET  > (defmacro loopN (n &rest bodies)  `(do\* ( (g (gensym)) (,g 0 (+ ,g 1)) )  ( (>= ,g ,n))  ,@ bodies  )  )  LOOPN  > (let ((i 10))  (loopN 5  (print i)  (setf i (+ i 1))  )  )  LET\*: variable G has no value  Why that error?  In ex5, we gave g a value before the backquote expansion of therefore, g had a value when the, g evaluates.  In ex6, g doesn’t have a value when the macro is expanding.  The assigning of (gensym) o g doesn’t happen until the macro expansion is completed and then the expression is executed. During the expansion, backquote see the ,g and attempts to get the value of g, **which hasn’t received a value//extra credit do let outside backquote** |
| **Scope**  Originally, LISP used dynamic scope. Today, most LISPs use a type of static scope known as **lexical scope**.  The **let** function provides a lexical scope for the variables defined in its variable definition list. Any references to those variables within the body expressions for that let will reference those lexically scoped variables.  Any called functions which have a non-local reference to a variable will reference a **global** and not those lexically scoped values. | ;;; Scope Example #1: Lexical Scope  > (setf ALPHA 10)  10  > (setf BETA 20)  20  > (defun sub2 (Y)  (print (list 'sub2 'alpha= Alpha 'beta= Beta 'y= y))  )  SUB2  > (let ((Alpha 30) (beta 40))  (print (list 'let 'alpha= Alpha 'beta= Beta ))  (sub2 5)) //sub2’s values here are from global alpha beta  (LET ALPHA= 30 BETA= 40)  (SUB2 ALPHA= 10 BETA= 20 Y= 5) |
| **Using Dynamic Scope in Common LISP**  Common LISP provides a mechanism to change a non-local reference to use dynamic scope rather than a reference to a global by using (**defvar** *variable*).  In some literature, this is described as making the **defvar**  variable SPECIAL. | ;;; Scope Example #2: Dynamic Scope using DEFVAR  ;;; Define the variable ALPHA to use dynamic scope.  > (DEFVAR ALPHA)  ;;; using the same sub2 from above and the same LET  > (let ((Alpha 30) (beta 40))  (print (list 'let 'alpha= Alpha 'beta= Beta ))  (sub2 5))  (LET ALPHA= 30 BETA= 40)  (SUB2 ALPHA= 30 BETA= 20 Y= 5) |
| **Evaluating Expressions**  The function **eval** is used to evaluate expressions. It is coded into the read-eval loop provided by the prompts in LISP. It can also be used directly.  It is sometimes necessary to dynamically construct an expression that is passed to **eval.** | > (eval '(max2 10 20))// evaluating expression  20  > (setf fn 'max2)  MAX2  > (setf args '(10 20))  (10 20)  > (eval (cons fn args))//evokes max2  20  > (eval `(,fn ,@ args))//val of fn (max 10 20)  20  > (eval `(,fn args)) //(max2 args)  Too few arguments (1 instead of at least 2)  > (setf perExpr (list '+  (list '\* 'width 2)  (list '\* 'length 2)))  (+ (\* width 2)  (\* length 2))  > (setf width 10 length 6)  6  > (eval perExpr)  32 |
| **Exercise: switch macro**  Suppose we would like a switch macro:  (switch *value*  (*val1 resExpr1*)  ...  (*valN resExprN*)  (default *resExprDefault*)  )  Example:  (switch x  ( 5 (print "it is 5") )  ( 10 (print "it is 10") )  ( default (print "default") )  )  Approach: loop through the pairs:   * if the (caar *pairs*) is 'DEFAULT or   the (caar *pairs*) is *value*,  return the eval of (cadar *pairs*)  Why return (eval (cadar *pairs*))? | ;;; Macro Example #6: SWITCH with return EVAL  > (defmacro switch (value &rest pairs)  (let ((g (gensym)) )  `( do ((,g ' ,pairs (cdr ,g)) )  ((null ,g) NIL)  (if (or  (eql 'DEFAULT (caar ,g))  (eql ,value (caar ,g))  )  (return (eval (cadar ,g)))  )  )  )  )  SWITCH  > (setf x 10)  > (switch x  ( 5 (print "it is 5") )  ( 10 (print "it is 10") )  ( default (print "default") )  )  "it is 10"  "it is 10"  //need quote on line 2 so It would see ,pairs as a list  //i.e, not evaluate it |
| Suppose we returned (cadar *pairs*) instead of (eval (cadar *pairs*))? | ;;; Macro Example #7: Incorrect SWITCH without EVAL  > (defmacro switch (value &rest pairs)  (let ((g (gensym)) )  `( do ((,g ' ,pairs (cdr ,g)) )  ((null ,g) NIL)  (if (or  (eql 'DEFAULT (caar ,g))  (eql ,value (caar ,g))  )  (return (cadar ,g))  )  )  )  )  SWITCH  > (switch x  ( 5 (print "it is 5") )  ( 10 (print "it is 10") )  ( default (print "default") ) )  (PRINT "it is 10")  // cadar is the literal list (print “It is 10”) need eval |
| For the "EQL ,value", should we use an eval?  With the example switch call above it wouldn't matter.  What about the one at the right? | ;;; suppose we used an expression for one of the values  ;;; when we use the SWITCH macro.  > (setf x 10)  10  > (switch x  ( 5 (print "it is 5") )  ( (+ 4 6) (print "it is 10") )  ( default (print "default") ) )  "default"  "default"  //problem: need to evaluate list (+ 4 6) to = 10  //currently only sees as a literal list |
|  | ;;; Macro Example #8: SWITCH  ;;; how can we fix that? what do we need to do?  > > (defmacro switch (value &rest pairs)  (let ((g (gensym)) )  `( do ((,g ' ,pairs (cdr ,g)) )  ((null ,g) NIL)  (if (or  (eql 'DEFAULT (caar ,g))  (eql ,value (eval(caar ,g)))  )  (return (eval (cadar ,g)))  )  )  )  )  SWITCH  ;;; Now it works  > (switch x  ( 5 (print "it is 5") )  ( (+ 4 6) (print "it is 10") )  ( default (print "default") ) )  "it is 10" "it is 10"  //bad code example  > > (defmacro switch (value &rest pairs)  (let ( )  `( do ((cv ' ,pairs (cdr cv)) )  ((null cv) NIL)  (if (or  (eql 'DEFAULT (caar cv))  (eql ,value (eval(caar cv)))  )  (return (eval (cadar cv)))  )  )  )  ) |