Optional Parameters in LISP

We can elegantly define procedures that include optional parameters in LISP by using the &OPTIONAL keyword in the argument list of a function definition. By enclosing the parameter(s) that follow the &optional keyword in parentheses one can supply a default value (otherwise they are automatically defaulted to nil). More than one parameter can follow the &optional keyword.

template: (defun fname (arg1 arg2 &optional arg3-argn) (<body>))

Example: Write a function to count the top-level elements of a list

```
(defun count-top (lis &optional (answer 0))
     (if (endp lis) answer (count-top (cdr lis) (+ 1 answer))
))
```

Note this version eliminates the need for a helper function

```
(defun count-top (lis) (count-help lis 0))
(defun count-help (lis answer)
     (if (endp lis) answer (count-help (cdr lis) (+ 1 answer))
))
```

Other Types of Parameters in LISP

The &REST parameter in LISP is bound to a list of all otherwise accounted for argument values. Using this we could define a multiple list append as follows:

```
(defun manyappend (&rest lists) (append-help lists))
(defun append-help (lists)
     (if (endp lists) nil (append (car lists)
                                  (append-help (cdr lists))) ))
```

Keyword parameters are designated by &KEY to allow for the easy identification of many parameters and to bypass positional binding. Using this we can write two versions of member, one using eq the other using equal.

```
(defun MYMEMBER (SEX LIS &KEY TESTFCN) (COND
     ((NULL LIS)                          NIL       )
     ((IF (NULL TESTFCN)
          (EQ SEX (CAR LIS))
          (TESTFCN SEX (CAR LIS)))        LIS       )
     (    T     (MYMEMBER SEX (CDR LIS) :TESTFCN TESTFCN))
     ))
```

Functional Arguments

Unfortunately this version looks good but it will not run! The LISP reader interprets TESTFCN as the actual name of a function (which it does not have) rather than a variable that represents the actual name of the function. Functional arguments (or parameters) are called *funargs* and demand special processing. The system supplies two functions to deal with funargs: FUNCALL and APPLY.

template: (funcall #'<fname> <arg1> <arg2> ... <argN>)

(car '(a b c)) == (funcall #'car '(a b c))
(append '(a b) '(c d)) == (funcall #'append '(a b) '(c d))

Apply works similarly, by using its first argument which must be a *funarg* on the elements of its second argument value, a list.

template: (apply #'<fname> <list of arguments>)

(car '(a b c)) == (apply #'car '((a b c)))
(append '(a b) '(c d)) == (apply #'append '((a b) (c d)))

Using funcall we can fix MYMEMBER as follows:

```
(defun MYMEMBER (SEX LIS &KEY TESTFCN) (COND
      ((NULL LIS)                                   NIL   )
      ((IF (NULL TESTFCN)
           (EQ SEX (CAR LIS))
           (FUNCALL TESTFCN SEX (CAR LIS)))  LIS  )
       (    T          (MYMEMBER SEX (CDR LIS) :TESTFCN TESTFCN))
))
```

MAPping Functions

The MAPCAR and related primitive functions allow us to easily transform lists. The function takes two arguments, the first is a *funarg* and the second a list. The answer is a list of the result of applying the functional argument to each successive car of the second argument. For example (mapcar #'f '(x_1 x_2 x_3)) ==> (f(x_1) f(x_2) f(x_3))
We can define mapcar as follows:

```
(defun mapcar (f lis) (cond
      ( (null lis)              nil )
      (     t     (cons (funcall f (car lis)) (mapcar f (cdr lis)))
      ))
```

(mapcar #'oddp '(1 2 3)) ==> (T NIL T)

Let us write substop, a function that substitutes sex1 for sex2 at the top-level of lis, a list of elements.

(defun replacesex (sex3) (if (equal sex1 sex3) sex2 sex3))
(defun substop (sex1 sex2 lis) (mapcar #'replacesex lis)) /* This code will not run */

If one tries this code, it fails because sex1 is unbound. The problem is resolved using lambda expressions below.

Related mapping functions include, REMOVE-IF and REMOVE-IF-NOT, where like MAPCAR the 1st argument is a *funarg* that is evaluated on each successive car of the second argument. The car is deleted/kept in the final answer, accordingly.

COUNT-IF and FIND-IF allow you to count/find elements in a list that satisfy a test predicate function (the *funarg*).

LAMBDA Expressions allow us to define anonymous procedures

The lambda expression is the actual mechanism that the LISP interpreter uses to "evaluate" dummy arguments in a procedure. This is what XLISP returns when you use FUNCTION-LAMBDA-EXPRESSION or what VAX LISP returns when you use PPRINT #'fname. Suppose we want to put quotes around all the elements of an input list.

```
(defun put-quotes(lis) (mapcar #'qhelp lis))
(defun qhelp(sex) (list (quote quote) sex))
```

But if you begin to run out of names for your "helper" functions (especially if they are only used once in a program) a more elegant solution is given by:

```
(defun put-quotes(lis)
        (mapcar #'(lambda(sex) (list (quote quote) sex)) lis ))
```

User-Defined MAPping Functions

Using our definition of MAPCAR we can define MAPPENDCAR, which applies f to successive cdrs as follows:

```
(defun mappendcar (f lis) (cond
     ( (null lis)               nil )
     (  t  (append (funcall f (car lis)) (mappendcar f (cdr lis)))
     ))

(defun inter (s1 s2) (mappendcar #'ihelp s1))
(defun ihelp (s1el) (cond
     ( (member s1el s2) (list s1el))
     ( t nil) ))
```

This function will not work unless we use a LAMBDA expression for ihelp in the definition of inter:

```
> (defun inter (s1 s2) (mappendcar #'(lambda (s1el) (cond
                              ( (member s1el s2) (list s1el))
           (    t                   nil)))         s1))
```

Here is the correct version of substop using lambda expressions:

```
> (defun substop (sex1 sex2 lis) (mapcar #'(lambda (lisel) (if (equal sex1 lisel) sex2 lisel))
lis))

> (substop 'coke 'pepsi '(coke is it))
(PEPSI IS IT)
```