Today’s Menu

- Continue with Advanced LISP Features
  - Optional Parameters in LISP
  - Other Types of Parameters in LISP
  - Functional Arguments, *funargs*
• **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:**
```lisp
(defun fname (arg1 arg2 &optional arg3-argn) (<body>) )
```

• Example: Write a function to count the top-level elements of a list
```lisp
(defun count-top (lis &optional (answer 0))
  (if (endp lis) answer
   (count-top (cdr lis) (+ 1 answer)) ))
```

```lisp
> (defun union4 (S1 S2 &optional (Ans S2)) (cond
    ( (null S1)      Ans)
    ( (member (car s1) S2) (union4 (cdr S1) S2 Ans))
    ( 'else          (union4 (cdr s1) S2 (cons (car S1) Ans)))
  ))
UNION4

> (union4 '(a b) '(c d)) > (union4 '(a b) nil)
(B A C D) (B A)
> (union4 '(a b c) '(c d)) > (union4 '(a b) '(a b))
(B A C D) (A B)
> (union4 nil '(a b)) > (union4 '(a b) '(b a c))
(A B) (B A C)
```
EEL5840: Elements of Machine Intelligence
LISP

/* Function union4 using append instead of cons */
> (defun union4 (S1 S2 &optional (Ans)) (cond
  ((null S1) (append Ans S2))
  ((member (car S1) S2) (union4 (cdr S1) S2 Ans))
  ('else (union4 (cdr S1) S2 (append Ans (list (car S1)))))
))
union4
> (union4 '(a b) '(c d))
(A B C D)
> (union4 '(a) '(a b c d))
(A B C D)
> (union4 '(a b) '(b c a))
(B C A)

> (defun memset (sex lis &optional (f 'equal))(cond
((null lis) nil)
((funcall f sex (car lis)) t)
('else (memset sex (cdr lis) f))
))
MEMSET
> (memset 'a '(a b))
T
> (memset 'a '(a b) 'eq)
NIL
> (memset 'a '(a b) 'equal)
T
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:

```lisp
> (defun manyappend (&rest lists) (app-h lists))
MANYAPPEND
> (defun app-h (lists)
  (if (endp lists) nil (append (car lists)
    (app-h (cdr lists))))))
APP-H
```

Why do we need a helper fcn?

The arguments to manyappend are bound in a list on each call.

```lisp
> (manyappend '(a b) '(c d) (d e))
creates lists = ((a b) (c d) (d e))
```

Recursing on app-h does not do this!

But recursing on manyappend adds another level of parentheses per call?

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.

```lisp
(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))))
```

Error: The function TESTFCN is not defined. Happened in: #<FSubr-COND: #735a14>
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. 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))
> (put-quotes '(this is a test))
((QUOTE THIS) (QUOTE IS) (QUOTE A) (QUOTE TEST))
```

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))
```

Lambda expressions are the actual execution mechanism for user-defined functions in XLISP. For example, (add1 2) or typing 
```
((lambda (x) (+ 1 x)) 2)
```
yield the same result, a 3 is returned.
See Class 07 & 08

LISP

The End!