1 A couple of Functions

Let's take another example of a simple lisp function – one that does insertion sort. Let us assume that this sort function takes as input a list of numbers and sorts them in ascending order. Remember how insertion sort works – it first orders 2 elements with respect to each other, then inserts a third into that in the correct place and so on. At step n, n numbers are sorted with respect to each other.

```
> (insert-sort '(5 1 2))
1 Enter INSERT-SORT (5 1 2)
| 2 Enter INSERT-SORT (1 2)
|   3 Enter INSERT-SORT (2)
|   | 4 Enter INSERT-SORT NIL
|   | 4 Exit INSERT-SORT NIL
|   | 4 Enter INSERT-INTO-SORTED 2 NIL
|   | 4 Exit INSERT-INTO-SORTED (2)
|   3 Exit INSERT-SORT (2)
|   3 Enter INSERT-INTO-SORTED 1 (2)
|   3 Exit INSERT-INTO-SORTED (1 2)
| 2 Exit INSERT-SORT (1 2)
| 2 Enter INSERT-INTO-SORTED 5 (1 2)
|   3 Enter INSERT-INTO-SORTED 5 (2)
|   | 4 Enter INSERT-INTO-SORTED 5 NIL
|   | 4 Exit INSERT-INTO-SORTED (5)
|   3 Exit INSERT-INTO-SORTED (2 5)
| 2 Exit INSERT-INTO-SORTED (1 2 5)
1 Exit INSERT-SORT (1 2 5)
(1 2 5)

(defun insert-sort (nums)
  ``takes a list of numbers and sorts them in ascending order
   using insertion sort''
```

```
> (insert-sort '(3 5 1 9 2 4))
(1 2 3 4 5 9)
```

2 More Complicated Recursion

So far everything we have defined requires only simple recursion – calling the function again on the cdr of a list. We may encounter problems which require more complicated recursion.

Let's do a more complicated example of a member function. Call it `(emb-member ele emb-list)`. This function returns t if ele occurs ANYWHERE in emb-list. Note here that emb-list is an arbitrary complicated list. So, for example:

```
> (emb-member 'a '(a b c))
T
> (emb-member 'a '((c d (a) b) e))
T
```

base: – emb-list is empty – ele is the car of emb-list

simpler-problems: – if car is emb-list is a list and ele is an emb-member of it – if ele is an emb-member of the cdr of l

```
(defun emb-member (ele emb-list)
  ``takes as input an s-expression, ele, and an arbitrarily
   complex list, emb-list.  Returns t if ele occurs somewhere
   in emb-list''
```

```
> (emb-member 'a '((c d (a) b) e))
T
> (emb-member 'a ())
NIL
> (emb-member 'a '(((a))))
T
```

```
> (emb-member 'a '(b (c d (e))))
NIL
```

Define a function `(subst new old emb-list)` which substitutes the value of new for old wherever old occurs in emb-list. Examples:

```
> (subst 'a 'b '((a b) (c (b) d) (e)))
((A A) (C (A) D) (E))
> (subst 'a 'b '(d b e f b))
(D A E F A)
> (subst '(a b) 'c '(a (b c) d ((e c))))
(A (B (A B)) D ((E (A B))))
> (subst 'a '(b c) '(a (b c) d e))
(A A D E)
```

base conditions: – when emb-list is empty, when the car of emb-list is old

easier problem: – look to see if new occurs in the car (provided it is a list) and cons what you get to doing the subst to the cdr – else cons the car to what you get from doing the subst to the cdr.

```
(defun subst (new old emb-list)
  ``substitutes new for old wherever it occurs in emb-list and
   returns the new list''
```

```
> (subst 'a 'b '((c b) b d))
((C A) A D)
```

3 The Symbol Table

Each entry in the symbol table is a fairly complicated thing. For instance, we can think of the symbol table as consisting of 5 fields (only 4 of which I know).

Print-Name	the name of the symbol that gets printed on the terminal.
Value	the symbols value as is set by setf or by being bound within a function call (when it is a formal parameter for the function for example).
Function Definition	the definition of the function named by the symbol. This definition is "set" via defun.
Property-list	a list of name-value pairs. These may be set using the `(setf (get name property) value)` function, and retrieved using the `(get name property)` function.

Property lists are useful as ways of holding information about objects. For instance, we might have some information about an object named AI-Class. For instance, this object may have 4 assignments, 5 homeworks, a midterm and a final exam, and an instructor named McCoy. This could be kept in a property list:

```
(setf (get 'AI-Class 'numb-assignments) 4)
(setf (get 'AI-Class 'numb-homeworks) 5)
(setf (get 'AI-Class 'exams) '(midterm final))
(setf (get 'AI-Class 'instructor) 'McCoy)
```

In our program we could then query these individual items:

```
> (get 'AI-Class 'numb-homeworks)
5
> (get 'AI-Class 'instructor)
McCoy
```

4 Lisp Style

Just a word on what the functions you hand in should look like.

- Indent code to reflect level of nesting of parenthesis. Otherwise it is impossible to read – if you put that setenv thing in your .login file vi will do a pretty good job of getting the indentation right. Conventions:
- Make code reflect the way you think about the problem – recursively! Each function usually consists of a conditional checking base conditions and then a recursive call – usually cons the car onto result of doing function to rest of list.
- Variable and function names should be well-chosen (descriptive)

- Use special names for global variables (e.g., *data*).
- Use global variables only when they are the clearest way to do things (e.g., as pointers to a global data base).
- Stress functional embedding. Avoid temp or local variables. "The judicious use of LET and functional embedding can remove the need for most instances of SETF. Doing this is considered the mark of an expert lisp programmer."
- Keep functions short. Break up large functions into several logically self-contained programs. This will make testing and debugging much easier!
- COMMENT CODE! Each function should have a block comment in the beginning explaining what kind of input arguments are expected, and what is returned. In the code, anything tricky should have a comment. Another mark of a good lisp programmer is to be able to look at a system written years before and figure out how it works in 5 minutes – find functions to borrow etc...

5 Logical Operators and Equal

First, a couple more predicates need to be introduced:

`(and` $x_1x_2...x_n$`)` – returns `NIL` is any argument evaluates to nil. This function is smart in that it only evaluates until it finds a null argument.

`(or` $x_1x_2...x_n$`)` – returns non-null if any argument evaluates to non-null. The value returned is the value of the first non-null argument. This function is also smart in its evaluation – it only evaluates until it finds an argument that evaluates to non-null.

`(list` $x_1x_2...x_n$`)` – makes a list out of the arguments.

In lisp, there are really two different functions for deciding whether or not two things are equal.

`(eql` x_1x_2`)` – returns t if x_1 and x_2 are pointers to the same memory location. That is, if they are the same atom, or they are pointers to the same list cell.

`(equal` x_1x_2`)` – returns t if x_1 and x_2 look the same. That is, if they are the same atom (as in eql), or if they are lists with the same structure and elements that look to be the same.

We can write the `equal` function in terms of `eql` as below:

```
(defun our-equal (x y)
  ``takes two s-expressions and returns t if they appear equal''
```

```
> (our-equal 'a 'a)
T
> (our-equal '(a b) '(a b))
T
> (our-equal '(a (b)) '(a b c))
NIL
> (our-equal '(a (b) c) '(a (b)))
NIL
> (our-equal '(a (b (c d)) e) '(a (b (c d)) e))
T
```

6 Printing Functions

If you want your functions to print something out on the screen, there are a number of lisp functions available for this purpose. You may find these particularly useful in debugging (if the trace function does not give you all of the information that you need).

`(print x)` – prints the value of x and returns the value of x.

`(princ x)` – like print but omits delimiters

`(terpri)` – prints a carriage return

Some simple examples:

```
> (print 'a)

A
A
> (print "this and that")

"this and that"
"this and that"
> (princ "this and that")
this and that
"this and that"
> (terpri)
```

```
NIL
```

The backquote ‘ is a rather useful device for formatting output. It allows you to insert a value for a variable (using ,) or splice the value in (using ,@). There are particularly useful in conjunction with the print statement.

```
> (setf x '(a b c))
(A B C)
> (print `(the value of x is ,x))

(THE VALUE OF X IS (A B C))
(THE VALUE OF X IS (A B C))
> (print `(the value of the three arguments is ,@x))

(THE VALUE OF THE THREE ARGUMENTS IS A B C)
(THE VALUE OF THE THREE ARGUMENTS IS A B C)
```

7 Iterative Programming Constructs

The most general form of an iterative construct can be found in the prog statement:

```
(prog (v1 v2 ... vn)
    e1
    e2
    .
    .
    .
   ek)
```

Allows local variables v1...vn to be set (using setf) within the prog statement. These values are initialized to nil upon entering the prog, and are “lost” upon exiting the prog.

Other things you might find in a prog: the (return value) statement – causes the prog to be exited and allows a value to be returned form the prog. Also useful is the (go label) – which causes control to be passed to the label “label”.

8 Functions as Arguments

In lisp, functions are first class objects. That is, you can treat a function just like any other object. In particular, you can pass them as arguments to functions.

Of course, when you do such a thing, you probably do it so you can <u>use</u> the function inside the function – so you need a method of calling a function which has been passed in as a value to another function.

`(apply fn-spec args-list)` is the function you want! The first argument to `apply` evaluates to a function (name), the second evaluates to the list containing the function arguments.

```
> (apply '+ (list 1 2 3 4))
10
> (apply '+ '(1 2 3 4))
10
> (apply 'cons '(a (b c)))
(A B C)
> (apply 'atom (list 'a))
T
```

Apply seems a little funny since the arguments are in a list. `funcall` takes a function (name) and a number of arguments to that function and applies the function to the arguments.

```
> (funcall 'cons 'a '(b c))
(A B C)
```

Why would you even want to use such function?

Recall our insertion sort function and remember how I told you that lisp does no type checking.... What if I wanted to sort into descending order instead of ascending order? Or, I wanted to sort characters instead of numbers?

We could do all of this if we not only passed the function a list to be sorted, but also passed it the comparison function you wanted it to sort by.

```
(defun insert-your-fun-sort (nums &optional (order-fn (function <)))
  ``takes a list and an ordering function which works on
   the elements in the list and sorts the list according
   to the sorting function using insertion sort''
  (cond ((null nums) nil)
        ((insert-your-fun-into-sorted (car nums)
                                      (insert-your-fun-sort
                                        (cdr nums)
                                        order-fn)
                                      order-fn))))

(defun insert-your-fun-into-sorted (ele l ordering-fun)
  ``inserts ele into the sorted list l according to the
   ordering function''
```

```
  (cond ((null l) (list ele))
        ((funcall ordering-fun ele (car l))
         (cons ele l))
        ((cons (car l)
               (insert-your-fun-into-sorted ele
                                            (cdr l)
                                            ordering-fun)))))

> (insert-your-fun-sort '(1 3 2 9 3 7) (function <))
(1 2 3 3 7 9)
> (insert-your-fun-sort '(1 3 2 9 3 7) (function >))
(9 7 3 3 2 1)
```

Lisp also gives you finer control over the evaluation of functions – so much so that it allows you to call the evaluation!

`eval` – is a function that takes one argument, it evaluates that argument and then evaluates it again.

```
> (setf x '(cons 'a '(b c)))
(CONS (QUOTE A) (QUOTE (B C)))
> (eval x)
(A B C)
```

9 Mapping Functions

Suppose you want to do something (the same thing) to several sets of arguments (not just one). For instance, you want to `add 1` to each member of an entire list of numbers. Lisp allows you to do this with some special mapping functions. Input to these functions are a function specification and a list of arguments. The mapping function will apply the function to each argument on the list and do something with the results (that something is what makes one mapping function different from another).

The most used mapping function is `mapcar`. E.g.,

```
> (mapcar '1+ '(100 200 300))
(101 201 301)
```

Note that the list may be a variable which you get out of a computation – you need not know its length.

```
(defun add-to-all (n-list)
  (mapcar '1+ n-list))
ADD-TO-ALL
> (add-to-all '(1 2 3 4 5))
(2 3 4 5 6)
```

`mapcar` can be used with functions requiring more than one argument. In this case you give one list for each argument.

```
> (mapcar '+ '(1 2 3 4) '(10 20 30 40) '(100 200 300 400))
(111 222 333 444)
```

`(mapcar <fn-specification> l1 ... ln)` where `l1 ... ln` are all lists of the same length `fn-specification` specifies a function of n arguments.

The value: `mapcar` evaluates its arguments and then applies the first argument to the cars of each of the latter arguments, then "cdr's" down each list. The value returned is a list containing the results of the function applications.

What if we want to apply some function of, say, two arguments to a list of items, but want the second argument to be the same in each case. For instance, in my dissertation at one point I have a special object `obj`. Part of the processing I have to do is to collect all objects in my knowledge base that have a special property `p`. I then want to test their similarity with my special object `obj`. I have a similarity function `(similarity obj1 obj2)`. How can I do this with mapcar short of generating a list of n objects where n is the number of items returned from my gathering step?

`mapcar` allows this kind of processing by essentially allowing you to specify a new function (which doesn't have a name). But first, we have to know a little bit more about functions.

10 Mapcar and Lambda Expressions

What does it mean to be a function? We use `defun` to define functions. Defun takes a function name, formal-parameter-list, and function body and stores these things under the "function" property in the symbol table under the function-name. Really the name doesn't do anything but associate things – the formal parameters and the body – it gives us a shorthand for referring to things.

Just in case you don't need that shorthand, it would be nice to be able to create a function without a name (for instance, to be used in mapcar). LAMBDA notation, gives us a way of doing that.

For instance, we might want a function which checks to see if the input is a list of length 2.

```
(lambda (a) (and (listp a) (equal (length a) 2)))
```

This is a lambda form. It is not a function CALL, but a FUNCTION SPECIFICATION – it is a function itself. It can be put wherever you put function names.

```
> ((lambda (a) (and (listp a) (equal (length a) 2))) '(a b))
T
```

We could use lambda notation in our mapcar before.

```
(defun foo (prop kb spec-obj)
           .
           .
           .
  (apply 'foo2 (mapcar (function (lambda (x)
                                   (similarity x spec-obj)))
                        (gather-objs-with-prop-prop kb))
```

Here I will do a comparable numeric example. We really want a function like:

```
(defun add-num (num list)
  (mapcar (function (lambda (x) (+ num x))) list))

> (tester 2 '(1 2 3 4))
(3 4 5 6)
```

(Note we use the function function here – this function is similar to the quote function but it is used to quote functions. It causes "lexical closure" to take place. On this instance, lexical closure makes available the local variables in the calling function.)

Lambda happens to be the basic internal mechanism with which functions are implemented. Defun actually builds a lambda form. The lambda is fetched for function application. Using defun hides this.

Another example. Suppose that I wanted to take a list of numbers, take the square root of each one and then add one to each (and return the results in a list).

```
(defun list-sqrt-add1 (lis)
  (mapcar 'sqrt-add1 lis))

(defun sqrt-add1 (x)
  (1+ (sqrt x)))
```

Is the above a valid helping function? Probably not. Lambda allows us to define a function within another function – this function is not even given a name since we don't expect it to be used again (it is rather specific to our purposes).

```
(defun new-list-sqrt-add1 (lis)
  (mapcar (function (lambda (x)
                      (1+ (sqrt x))))
          lis))
> (new-list-sqrt-add1 '(1 4 9))
(2.0 3.0 4.0)
```

Now write a function like the above, except it sums the results:

```
(defun add-list-sqrt-add1 (lis)
  (apply '+  (mapcar (function (lambda (x)
                                  (1+ (sqrt x))))
                     lis)))

> (add-list-sqrt-add1 '(1 4 9))
9.0
```

11 Let Statements

Suppose we want to write a function `longer-list` which takes two lists. It returns the list that contains the most elements or the symbol `equal` if both lists are the same length.

```
(defun longer-list (l1 l2)
  ``does as above says!''
  (cond ((> (length l1) (length l2)) l1)
        ((> (length l2) (length l1)) l2)
        (t 'equal)))
```

Problem – this function causes length to be evaluated a number of times. The let statement allows us to "set a local variable" and thus avoid that extra computation.

`(let ``list'' ``exps'')`

`` ``list'' `` is a possibly empty list of objects each having the form `(symbol expression)` – in parallel all expressions are evaluated and simultaneously bound to their corresponding symbols. Next, `` ``exps'' `` are evaluated (the last one evaluated is returned as the value of the let) and the symbols are returned to their previous values.

```
(defun let-longer-list (l1 l2)
  ``like above but uses let''
  (let ((len-l1 (length l1)) (len-l2 (length l2)))
    (cond ((> len-l1 len-l2) l1)
          ((> len-l2 len-l1) l2)
          (t 'equal))))
```

12 Some Additional Lisp Functions

Lisp lists are a convenient way to represent *sets* – a collection of 0 or more elements in which there are no duplicates.

```
(defun setunion (s1 s2)
  (cond ((null s1) s2)
        ((our-member (car s1) s2)
```

```
         (setunion (cdr s1) s2))
        ((cons (car s1)
              (setunion (cdr s1) s2)))))

(defun setintersection (s1 s2)
  (cond ((or (null s1) (null s2)) nil)
        ((our-member (car s1) s2)
         (cons (car s1)
              (setintersection (cdr s1) s2)))
        ((setintersection (cdr s1) s2))))


(defun remove-all (x l)
  (cond ((null l) nil)
        ((equal (car l) x)
         (remove-all x (cdr l)))
        ((cons (car l)
              (remove-all x (cdr l))))))

(defun count-up (n)
  ``creates a list of numbers from 1 to n if n > 0, nil otherwise''
  (count-up-rec 1 n))

(defun count-up-rec (start end)
  ``does the work for count-up and checks boundary conditions''
  (cond ((> start end) ())
        ((cons start (count-up-rec
                        (1+ start)
                        end)))))
> (count-up 5)
(1 2 3 4 5)
> (count-up 0)
NIL


(defun classify (l)
  ``takes a list of elements and returns a list made
   up of the original elements of l but all numbers have
   been put in the car and all identifiers in the cadr
   and all else is ignored''
  (cond ((null l) (list nil nil))
        ((numberp (car l))
         (add-to-car (car l)
                     (classify (cdr l))))
        ((atom (car l))
```

```
         (add-to-cadr (car l)
                      (classify (cdr l))))
        ((classify (cdr l)))))

(defun add-to-car (ele l)
  (cons (union (list ele) (car l))
        (cdr l)))

(defun add-to-cadr (ele l)
  (cons (car l)
        (list (union (list ele)
                     (cadr l)))))
```