CS107 Handout 52
Autumn 2000 November 29, 2000

LISP Functions as Data

Handout written by Julie Zelenski and Nick Parlante.

One of the points we earlier remarked on is that programs and data take the same form in LISP— both are expressed as lists. Capitalizing on this, it's very easy to write LISP programs that operate on (e.g. generate, modify, evaluate, etc.) other programs and functions. Although you've seen some weaker form of this with C function pointers, in LISP, functions can be manipulated much more completely and flexibly than a CT-language like C can allow.

Eval

As a start, consider the function `eval` which evaluates its single list argument according to the usual rules of evaluation, just as if that S-expression has been fed into the top-level read-eval-print loop of the interpreter. (In fact, the read-eval-print loop calls `eval` itself!)

```
? (eval '(* 3 4))
12
? (eval (cons '+ '(1 2 3)))
6
```

Note that in the first example, even without quoting the expression we are passing to `eval`, it would evaluate to the same thing, because the arguments to a function are recursively evaluated before calling the function. `eval` is more useful when the expression we are evaluating is constant, but one constructed dynamically.

Passing Functions as Parameters

In C, we passed a function pointer to the polymorphic `qsort` function in order to specify how to compare the elements, whatever type they might be. Similarly, LISP's built-in `sort` function is intended to sort lists of any type, it makes no assumption about what comparison function to apply to the list elements, instead the comparator is also passed as an argument. The comparator should be a predicate function that returns `T` if the first argument is strictly less than the second argument and `nil` otherwise. If we were sorting a list of numbers, we might want to pass the arithmetic comparator < to the sort function. However, our first attempt isn't quite what is needed:

```
? (sort '(4 7 56 2 3) <)
> Error: Unbound variable: <
> While executing: SYMBOL-VALUE
> Type Command-/ to continue, Command-. to abort.
> If continued: Retry getting the value of <.
See the Restarts… menu item for further choices.
```

Remember that all arguments to a function are first evaluated before the function is called, and the error here indicates the interpreter choked when looking up the value bound to the symbol <. This makes sense because the symbol < doesn't have a value bound to it, however it does have a **function** bound to it. A symbol has different bindings associated with it, one for its value, another for its function (remember in lecture how we talked about how naming a parameter `list` doesn't interfere with use of `list` as a function symbol?) The interpreter can keep them straight because there is no ambiguity about how to interpret a symbol. If a symbol is the first element in an S-expression, the interpreter uses the function binding for that symbol to apply to the rest of the list. Any symbol elsewhere in an S-expression evaluates to its value binding. Due to the position of < in our above sort expression, the interpreter is trying access its symbol value. However, we would don't want the value binding, we want the function binding, thus we need to manually extract the function binding ourselves.

Function Objects

`#'` takes a symbol like '+' and extracts the function binding associated with that symbol. The "function object" can then be passed as a parameter.

```
?#'cons
#<a compiled functional object for the symbol CONS>
?#'+
#<a compiled functional object for the symbol +>
```

To get back to our sorting example, we can pass the function binding for < like this:

```
?(sort '(45 6 34 2 34 12 9) #'<)
(2 6 9 12 34 34 45)
```

There are many nifty things we can do with a function object once we have one.

mapcar

The `mapcar` built-in function is an "iterator" for lists. If you have a list and want to do something to each element, `mapcar` is the function for you. In its simplest form, it takes a list and a function of one argument. It runs the function down the list, evaluating the function once for each element. If there are N elements in the list, then the function will be called N times, and the N results will collected and returned in their own list.

```
? (mapcar #'atom '(1 "hi" (4 5) 6.7))
(T T NIL T)

?(mapcar #'sqrt '(1 4 10 16))
(1 2 3.1622776601683795 4)

?(mapcar #'list '(1 2 3 4 5 6 7))
((1) (2) (3) (4) (5) (6) (7))
```

`mapcar` is profoundly useful— many computations include steps where you want to iterate over a collection of data. In a language that has a well-defined iterator, you just specify what to do for each element (the function of one argument) and the iterator deals with the details of iterating over the collection and collecting the results. This is sort of like `ArrayMap`, but it turns out to be even better, just wait.

`funcall` and `apply`

The `funcall` and `apply` functions allow you to "call" a function object. `apply` takes a function and the arguments to that function wrapped up in a list. `funcall` is like `apply` except the arguments to the function are not wrapped up in a list.

```
?(funcall #'+ 1 2 3)       ;; like (+ 1 2 3)
6
?(apply #'+ '(1 2 3))      ;; note difference b/w apply & funcall
6

?(funcall #'append  '((1) (2) (3) (4 5) (6)))
((1) (2) (3) (4 5) (6))
?(apply #'append  '((1) (2) (3) (4 5) (6)))
(1 2 3 4 5 6)
```

The main appeal of funcall and apply is that they allow you to write functions which take in function objects as arguments and invoke them. `mapcar` is an example of such a function. Now with `funcall`, we are capable of writing our own implementation of `mapcar`:

```
;; Takes a functional object of one argument and a list.
;; Calls the function once for each element in the list and returns
;; the list of results.

? (defun my-mapcar (fn list)
    (if (null list) '()
      (cons (funcall fn (car list))    ;; call fn with (car list) as the argument
         (my-mapcar fn (cdr list)))))  ;; cons onto result of recursive call

? (defun square (x) (* x x))  ;; a function of a single argument

? (my-mapcar #'square '(1 2 3 4))
(1 4 9 16)
```

Lambda

The final element necessary for exceptionally flexible use of functions is a shorthand notation for specifying functions on the fly. A lambda expression allows you to create an unnamed (sometimes called **anonymous**) function object anywhere you need one. The function is defined within a lexical scope— that is, the function is aware of the bindings active within its environment and the function only exists as long as the lexical scope does.

We can execute an anonymous function, bind it to a variable, or pass it as a parameter, just like any named function object. We create anonymous functions using the `lambda` built-in. A `lambda` expression looks like a `defun` missing the name of the function. The lambda directly returns the functional object instead of binding it to some symbol. The syntax for building a function object using lambda is:

```
   #'(lambda (arg-list) expression-to-evaluate)
```

In the square example above, we used defun to create a functional object which squares a number and bound that functional object to the symbol square. We then used #' to get the functional object back and passed it to mapcar. That sequence can be simplified using lambda to just the following:

```
   ? (mapcar #'(lambda (x) (* x x)) '(1 2 3 4))
   (1 4 9 16)
```

In its simplest form, a lambda is just a shorthand which saves you having to do a defun. However, there are some more clever things you can do with a lambda expression because of the lexical closure associated with such a function.

Lexical Closures

A lambda expression creates what is called a **lexical closure**. In a lexical closure, the environment in which the function was created is "closed over" so that all of the variables that are visible at this creation time will be available at any future execution time (i.e. each time the function is called). This can come in handy if you use lambda to define a function within another function. Essentially, there are times when you want to define function A inside function B because you want A to be able to see some of B's local variables. Given lexical scoping, the only way to get this effect is to define A inside of B. Suppose you wanted to define a function which would take a list of numbers and a scale factor, and multiply all the numbers by the scale factor.

```
   ? (scale '(0 1 -1 5 10 11) 10)   ;; scale numbers up by 10
   (0 10 -10 50 100 100)
```

You can use mapcar for this, but you need a functional object which knows about the current scale factor, and since mapcar only passes one argument (the list element) to the iterator function, it will have to obtain access to the scale factor by some other means (potentially with some vulgarity like a global variable.) With lexical scoping, we can create a lambda that can see the variable called factor. When the lambda is passed, to mapcar in this case, the lambda contains the environment where it was defined, so it can see the values of any previously bound variables such as factor. It would not be possible to write scale using mapcar without this feature of lambdas.

```
   (defun scale (list factor)
     (mapcar #'(lambda (num) (* num factor)) list))
```

The lexical scoping property of a lambda expression increases the usefulness of function objects in LISP tremendously. In particular, C function pointers are not capable of getting the lexical scoping behavior right, which makes them considerably less useful than a full lambda expression.

Exercise: Suppose you had a linked list package in C with a mapcar function that takes a function pointer and evaluates the function once for each element in the list. Try to write the scale function above without any client data or global variable hack.

Returning a Function

All of our examples so far have involved passing a function into a function. But it is also possible to pass a function **out** of another function. Here is a simple example that creates and returns a new function which computes the absolute difference between two numbers:

```
(defun delta-fn()
   #'(lambda (x y) (abs (- x y))))

? (delta-fn)
#<Anonymous Function #x3217F6>
```

Note that evaluating delta-fn returns a new function object, one that could be passed to mapcar or used with apply or funcall to evaluate on a pair of arguments.

```
? (funcall (delta-fn) 5 9)
4
```

Every time we evaluate delta-fn, we get the same function back. Not too interesting and note that its result is somewhat more awkward to use than symbol returned by defun. But, by capitalizing on the fact that we have lexical closures, we can write a function that creates a function which depends upon the bindings in the environment at creation time. By calling it with different bindings, we can create different versions of the function. For example, let's build a function that will return a multiply-by function for use in the scale operation above.

```
(defun multiply-by (factor)
   #'(lambda (num) (* num factor)))

? (multiply-by 10)
#<COMPILED-LEXICAL-CLOSURE #x318856>
```

Evaluating multiply-by returns a new function and keeps with it its lexical closure, which includes the binding for the parameter factor. If we funcall our new function, it will have the effect of multiplying its argument by the saved binding for factor, which is this case, was 10:

```
? (funcall (multiply-by 10) 5)
50
```

By calling `multiply-by` with an argument of `17`, we create a different new function that has a saved binding of `17` for the factor. And so on for any value of the factor to multiply by that we would like to create a function for.

Using the `multiply-by` function, we can re-write our scale example to generate a new function that contains the binding for the correct scale factor and then use `mapcar` to apply it.

```
(defun scale(list factor)
   (mapcar (multiply-by factor) list))

? (scale '(3 4 5 6 7 5 6 4) 10)
(30 40 50 60 70 50 60 40)
```

Functions in and out!

Consider this example that takes two functions as parameters and returns a function object that represents the composition of the functions:

```
(defun compose (f g)
  #'(lambda (x) (funcall f (funcall g x))))
```

The parameter bindings for `f` and `g` will be contained within the lexical closure returned and thus will be valid when used as part of evaluating the returned function.

```
? (funcall (compose #'sqrt #'abs) -9)
3
```

An example where we might want a compose operation would be if we had a two-step process we wanted to apply when iterating down a list. We could first compose the two steps into one combined function and then iterate using mapcar just once:

```
? (mapcar (compose #'sqrt #'abs) '(-9 16 1024 -12 -1))
(3 4 32 3.4641016151377544 1)
```

Rather than using `mapcar` twice:

```
? (mapcar #'sqrt (mapcar #'abs '(-9 16 1024 -12 -1)))
(3 4 32 3.4641016151377544 1)
```

Some examples

The following examples use geometric points— a point is represented as a list of numbers. So the (x,y) point (1,3) would be represented as the two element list (1 3). This code shows the use of appy, mapping, and lambdas to sort points by their distance from the middle point.

```
;;; Return the average of a non-empty list of numbers
(defun average (nums)
  (/ (apply #'+ nums) (length nums)))


(defun square(num) (* num num))


;;; Return the distance between two two-dimensional points
(defun distance(pt1 pt2)
  (sqrt (+ (square (- (first pt1) (first pt2)))
           (square (- (second pt1) (second pt2))))))


;;; Given a list of 2-d points, compute the center point
;;; which is the average of all the points.
(defun center-point(points)
  (list
    (average (mapcar #'car points))
    (average (mapcar #'second points))))


;;; A variant of the above which works for points of any dimension.
(defun center-point2(points)
  (if (null (car points)) '()
    (cons (average (mapcar #'car points))
          (center-point2 (mapcar #'cdr points)))))


;;; Given two points of any dimension, compute their sum point.
(defun add-points(pt1 pt2)
  (if (null pt1) '()
    (cons (+ (car pt1) (car pt2))
          (add-points (cdr pt1) (cdr pt2)))))


;;; Given a list of 2-d points, sort them into increasing order
;;; according to their distance from the middle.
(defun radial-sort(points)
  (let ((middle (center-point points)))
    (sort points #'(lambda (pt1 pt2) (< (distance pt1 middle)
                                        (distance pt2 middle))))))
```