Higher order functions

- As we saw earlier with apply and maphash, some functions can accept other functions as parameters

```
(apply ‘somefunction ListOfArgs)
(maphash ‘somefunction HashTable)
```

- Sort is another example of a higher order function since we pass the comparison function as one of the parameters, e.g.

```
(sort ‘(10 3 5 22) ‘<)
```

When to use higher order functions

- As we’ve seen with sort, maphash, and apply, sometimes we have an algorithm that is fairly uniform, but different specific functions could be inserted in one spot, e.g. sorting using <, or >, or string<, etc
- Sometimes we will have our lisp code build new functions as it runs. We need a way to run these, but can’t put calls into the source code because the functions don’t exist yet, so we pass a variable containing the newly built function to our higher order function

apply

- As we saw earlier this term, (apply f L) runs function f taking its parameters from L, e.g.

```
(apply ‘+ ‘(10 20))  ; acts like (+ 10 20)
```

- Make sure you have valid operands before calling apply:

apply will crash if passed invalid parameters, and the function being run may crash if the contents of L aren’t suitable for that function

When would we use apply?

- If the arguments we want to pass to a function are already in a potentially long list, then apply can be very effective
- Even if we know the exact length of the list, something like `(apply ‘f L)` is cleaner than something like

```
(f (nth 0 L) (nth 1 L) ... (nth i L))
```

- When f can accept any number of arguments (using &rest) then apply can be the most effective way to go (as with our final version of “smallest” in the &rest examples)

funcall

- funcall is similar to apply, except that we actually list all the arguments individually, rather than as a list, e.g.

```
(funcall f x y z)  ; acts like (f x y z)
```

- This is typically used in situations where f is passed to us as a parameter or stored in a variable

eval

- Eval is similar to apply and funcall, but this time the entire expression has been built as a list and we now want to run that list like a command, e.g.

```
(eval ‘(+ x y z))  ; acts like (+ x y z)
```

- This foreshadows the idea that we’ll use lisp code to build lists of lisp code, then execute them later, e.g.

```
(defvar a 10)
(defvar e (list ‘sqrt a)) ;*actually builds ‘(sqrt 10)
(eval e)  ; runs (sqrt 10)
```

Eval cont.

- Same snippet, but using symbol ‘a

```
(defvar e (list ‘sqrt ‘a)) ; builds ‘(sqrt a)
(defvar a 16)
(eval e)   ; returns 4
(setf a 49)
(eval e) ; returns 7
```

map

- Map allows us to run a function a number of times, and build a list, string, vector etc out of the results
- We specify what form the result should be in (e.g. ‘list), the function to run (e.g. ‘foo), and then provide a separate list of values for each of the arguments the function expects
- e.g. suppose our foo expects 3 parameters, and we want to run (foo 1 2 3) and (foo 10 17 4) and build a list of the results. The call to map would look like:

```
(map ‘list ‘foo ‘(1 10) ‘(2 17) ‘(3 4))
```

maplist

- For functions that expect just a list, maplist runs on the list, then the tail of the list, then the tail of that, etc, and build a list of the results

```
(maplist ‘length ‘(10 20 30))
```

- Runs length on (10 20 30) and gets 3
- Runs length on (20 30) and gets 2
- Runs length on (30) and gets 1
- Sees the empty list and stops
- Finally returns (3 2 1)

mapcar

- Also for functions that expect just a list, mapcar runs the function once on each element and builds a list of results

```
(mapcar ‘sqrt ‘(16 4 169))
```

- Returns (4 2 13)

reduce

- Suppose we have a situation where we want to use a value computed so far together with the next data value in line, and come up with a new result
- e.g. smallest: we keep track of the smallest value so far, and keep checking the next value in the list until we reach the end
- Reduce lets you specify a function and a list of values, and keeps running the function on the front two values and replacing them with the new result

Reduce example

```
(reduce '+ '(10 5 17 20))
```

- runs (+ 10 5) and replaces them with the answer, now

```
(reduce '+ (15 17 20))
```

- runs (+ 15 17) and replaces them with the answer, now

```
(reduce '+ '(32 20))
```

- runs (+ 32 20) and replaces them with the answer, now
- just a single element left, 52, so returns that as the answer

Next steps

- Now that we can use higher order functions, we'll start developing lisp code that writes lisp code
- We can have our scripts generate and (though higher order functions) run new lisp functions or expressions
- We can create functions which analyze and rewrite existing lisp code, then run the revised version
- We can combine this with let blocks to create class/object-like behaviour in lisp