Overview

- In a functional programming language, functions are first class objects.
- You can create them, put them in data structures, compose them, specialize them, apply them to arguments, etc.
- We’ll look at how functional programming things are done in Lisp

Functions as objects

- In lisp, functions are regular objects, like symbols, or strings, or lists.
- If we give the name of a function to `function`, it will return the associated object.
- Like `quote`, `function` is a special operator, so we don’t have to quote the argument:
  ```lisp
  > (setf s1 '(cadr '(one two three)))
  (CADR '(ONE TWO THREE))
  > (eval s1)
  TWO
  > (eval (list 'cdr (car '((quote (a . b)) c))))
  B
  ```

Functional Programming and Lisp

- Remember: Lisp code is just an s-expression
- You can call Lisp’s evaluation process with the `eval` function.
  ```lisp
  > (eval '(s1 '(cadr '(one two three))))
  (CADR '(ONE TWO THREE))
  > (eval s1)
  TWO
  > (eval (list 'cdr (car '((quote (a . b)) c)))))
  B
  ```

Eval

- Remember: Lisp code is just an s-expression
- You can call Lisp’s evaluation process with the `eval` function.
  ```lisp
  > (> (setf s1 '(cadr '(one two three)))
    (CADR '(ONE TWO THREE)))
  > (> (evaleval s1)s1)
  TWO
  > (> (evaleval (list '(list 'cdrcdr (car '((quote (a . b)) c))))
    (car '((quote (a . b)) c))))
  BB
  ```
Functions as objects
- Just as we can use ` as an abbreviation for quote, we can use ` as an abbreviation for function.
  > `+
  ' <SYSTEM-FUNCTION +>
- This abbreviation is known as sharp-quote.
- Like any other kind of object, we can pass functions as arguments.
- One function that takes a function as an argument is apply.

Apply
- Apply takes a function and a list of arguments for it, and returns the result of applying the function to the arguments:
  > (apply `+ (1 2 3))
  6
- It can be given any number of arguments, so long as the last is a list:
  > (apply `+ 1 2 '(3 4 5))
  15
- A simple version of apply could be written as
  (defun apply (f list) (eval (cons f list)))

Funcall
- The function funcall is like apply but does not need the arguments to be packaged in a list:
  > (funcall `+ 1 2 3)
  6
- It could be written as:
  (defun funcall (f &rest args)
  (eval (cons f args)))

Lambda
- The defun macro creates a function and gives it a name.
- However, functions don't have to have names, and we don't need defun to define them.
- We can refer to functions literally by using a lambda expression.
Lambda expression

A lambda expression is a list containing the symbol `lambda`, followed by a list of parameters, followed by a body of zero or more expressions:

> (setf f (lambda (x) (+ x 1)))
> (funcall f 100)

101

Lambda expression

A lambda expression can be considered as the name of a function.

Like an ordinary function name, a lambda expression can be the first element of a function call:

> ((lambda (x) (+ x 100)) 1)
101

and by affixing a sharp-quote to a lambda expression, we get the corresponding function:

> (funcall #'(lambda (x) (+ x 100)) 1)
101

Mapping functions

Common Lisp provides several mapping functions.

Mapcar is the most frequently used.

It takes a function and one or more lists, and returns the result of applying the function to elements taken from each list, until one of the lists runs out:

> (mapcar #'abs '(3 -4 2 -5 -6))
(3 4 2 5 6)
> (mapcar #'cons '(a b c) '(1 2 3))
((a . 1) (b . 2) (c . 3))
> (mapcar #'(lambda (x) (+ x 10)) '(1 2 3))
(11 12 13)
> (mapcar #'list '(a b c) '(1 2 3 4))
((A 1) (B 2) (C 3))

Maplist

The related function maplist takes the same arguments, but calls the function on successive cdrs of the lists:

> (maplist #'(lambda (x) (if (member (car x) '(a b c))) 0 1))
'(0 1 0 1 1 1)

There is also mapcan, mapc, and mapl. Use the on-line Common Lisp the Language to discover what these mapping functions do.
Every and Some

- *every* and *some* take a predicate and one or more sequences.
- When given just one sequence, they test whether the elements satisfy the predicate:
  > (every #'oddp '(1 3 5))
  T
  > (some #'evenp '(1 2 3))
  T
- If given >1 sequences, the predicate must take as many arguments as there are sequences, and arguments are drawn one at a time from them:
  > (every #'> '(1 3 5) '(0 2 4))

Example: filter

(defun filter (list function)
  ;; returns a list of elements of list for which function is true.
  (cond ((null list) nil)
    ((null list) nil)
    ((funcall function (car list))
      (cons (car list)
        (reduce function final (cdrcdr list))))
    (t (reduce function final (cdrcdr list)))
  )

> (filter '(1 2 3 4 5 6) #'evenp)
(2 4 6)
> (filter '(1 2 3 4 5 6 7) #'evenp)
(2 4 6)
> (filter '(integers 2 20) #'prime )
(2 3 5 7 11 13 17 19)

Example: reduce

- Reduce takes (i) a function, (ii) a final value, and (iii) a list.
- Reduce (+ 0 [v1 v2 v3 ... vn]) is just V1 + V2 + V3 + ... Vn +0
- In Lisp notation:
  > (reduce #'+ 0 '(1 2 3 4 5))
  15
  (reduce #'* 1 '(1 2 3 4 5))
  120

(defun reduce (function final list)
  (if (null list) (if (null list)
    final
    ((funcall function (car list))
      (cons (car list)
        (reduce function final (cdrcdr list))))
    (t (reduce function final (cdrcdr list)))
  )
)

(defun sumlist (list) (reduce #'+ 0 list))
(defun mullist (list) (reduce #'* 1 list))
(defun copylist (list) (reduce #'cons nil list))
(defun appendlist (list) (reduce #'append nil list))
Closures

- Lisp is a lexically scoped language.
- Free variables referenced in a function those are looked up in the environment in which the function is defined.
- Free variables are those a function (or block) doesn’t create scope for.
- A closure is a function that remembers the environment in which it was created.
- An environment is just a collection of variable bindings and their values.

Closure example

```lisp
> (defun make-counter ()
  (let ((count 0))
    (lambda () (setf count (1+ count)))))
MAKE-COUNTER
> (setf c (make-counter))
#<CLOSURE :LAMBDA NIL (SETF COUNT (1+ COUNT))>
> (funcall c)
1
> (funcall c)
2
> (funcall c)
3
```

Closure example 2

```lisp
> (defun make-counter (&optional (increment 1))
  (let ((count 0))
    (lambda () (setf count (+ count increment))))
MAKE-COUNTER
> (setf c2 (make-counter 2))
#<CLOSURE :LAMBDA NIL (SETF COUNT (+ COUNT #<CLOSURE :LAMBDA NIL (SETF COUNT (+ COUNT INCREMENT))>))>
> (funcall c2)
2
> (funcall c2)
4
```