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

Concepts and functions

- Eval
- Apply
- Funcall
- Lambda

Evaluate

- Remember: Lisp code is just an s-expression
- You can call Lisp’s evaluation process with the eval function.

```lisp
> (setf s1 '(second '(one two three)))
(second 'ONE TWO THREE))
> (eval s1)

TWOTWO

> (eval (list 'cdr (car '((quote (a . b)) c))))

B
```

UMBC

An Honors University in Maryland
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
  > (defun add1 (n) (+ n 1))
  ADD1
  > (function +)
  #<SYSTEM::FUNCTION +>
  > (function add1)
  #<CLOSURE ADD1 (N) (DECLARE (SYSTEM::IN-DEFUN ADD1)) (BLOCK ADD1 (+ N 1))>
  ```

Just as we can use `quote` as an abbreviation for `quote`, we can use `#` as an abbreviation for `function`:
```lisp
> #'+
#<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:
  ```lisp
  > (apply #'+ '(1 2 3))
  6
  ```
- It can be given any number of arguments, so long as the last is a list:
  ```lisp
  > (apply #'+ 1 2 '(3 4 5))
  15
  ```
- A simplified version of `apply` could be written:
  ```lisp
  (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:
  ```lisp
  > (funcall #'+ 1 2 3)
  6
  ```
- It could be written as:
  ```lisp
  (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*.
- Think of a lambda expression as returning (when evaluated) an anonymous function.

---

**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:
  ```lisp
  > (setf f (lambda (x) (+ x 1)))
  #<CLOSURE :LAMBDA (X) (+ X 1)>
  > (funcall f 100)
  101
  ``

---

**Recap**

- Two ways to make functions
  - (defun foo (x) (+ x 1))
  - (setf bar (lambda (x) (+ x 1)))
- Calling a function
  - (foo 100) => 101
  - ((lambda (x) (+ x 1)) 100) => 101
  - (bar 100) => ???
- Getting a functional object
  - (function foo)
  - (function (lambda (x) (+ x 100)))
**Who cares?**

- What good is this, anyway?
- Having functions as “first class objects” allows us to do some interesting things.
- As we will see...

**Mapcar and friends**

- Sometimes it's useful to apply a function to every element of a list and return a list of the results.

  ```lisp
  > (mapcar #'+ '(-1 2 -3))
  (0 3 -2)
  > (mapcar #'abs (mapcar #'+ '(-1 2 -3)))
  (0 3 2)
  > (mapcar 'integerp '(a 2 (b) -3 nil 4))
  (nil t nil t nil t)
  ```

**Defining mapcar**

- Mapcar is built in but could be defined

  ```lisp
  (defun mapcar (f list)
    (cond ((null list) nil)
          ((consp list)
           (cons (funcall f (car list))
                 (mapcar f (cdr list))))
          (t (error "bad arg to mapcar"))
    ))
  ```

**Mapping functions**

- Mapcar has been generalized to functions which take more than one argument

  ```lisp
  > (mapcar #'> '(5 3 1 0) '(4 4 4 -4))
  (t nil nil t)
  > (mapcar #'cons '(a b c) '(1 2 3))
  ((a . 1) (b . 2) (c . 3))
  > (mapcar #'list '(a b c) '(1 2 3 4))
  ((A 1) (B 2) (C 3))
  > (mapcar #'(lambda (x y) (* (- x 1)(+ y 1)))
            '(1 2 3) '(2 3 4))
  (0 4 10)
  ```
Maplist

- The related function `maplist` takes the same arguments, but calls the function on successive CDRs of the lists:
  ```lisp
  > (maplist #'(lambda (x) (cons 'foo x)) '(a b c d))
  (foo a b c d) (foo b c d) (foo c d) (foo d)
  > (maplist #'(lambda (x) (if (member (car x) (cdr x)) 0 1))
    '(a b a c d b c))
  (0 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:
  ```lisp
  > (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:
  ```lisp
  > (every #'> '(1 3 5) '(0 2 4))
  T
  ```

Example: filter

```lisp
(defun filter (function list)
  ;; returns elements of list for which function is true.
  (cond ((null list) nil)
        ((funcall function (car list)) (cons (car list) (filter function (cdr list))))
        (t (filter function (cdr list)))))
```

```lisp
> (filter #'evenp '(1 2 3 4 5 6 7))
(2 4 6)
> (filter #'prime (integers 2 20))
(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:
  ```lisp
  > (reduce #'+ '(1 2 3 4 5))
  15
  > (reduce #'* 1 '(1 2 3 4 5))
  120
  ```
Example: reduce

(defun reduce (function final list)
  (if (null list)
      final
      (funcall function
        (first list)
        (reduce function final (rest list)))))

(defun sumlist (list) (reduce #'+ 0 list))
(defun mullist (list) (reduce #'* 1 list))

Examples: reduce

(defun copylist (list) (reduce #'cons nil list))
(defun appendlist (list) (reduce #'append nil list))

> (appendlist '((a b)(1 2)(c d)(3 4)))
(a b 1 2 c d 3 4)

Free variables

- Suppose we have a function with a free variable, like increment in this:
  (defun huh (x) (+ x increment))
- If we call (huh 100) what should happen?
- Lisp is lexically scoped, so free variables are looked up in the environment in which the function was defined.

> (setf increment 1000)
1000
> (huh 100)
1100

When are functions defined?

- Most of the time we use defun, we are defining functions “at the top level”
- So free variables are global variables
- But, lambda expressions allow us to define functions in a local environment
- => the need for closures
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)))))
```

```
> (setf c1 (make-counter))
#<CLOSURE:LAMBDA NIL (SETF COUNT (1+ COUNT))>
> (funcall c1)
1
> (funcall c1)
2
> (funcall c1)
3
```

Closure example 2

```lisp
(defun make-counter (&optional (increment 1))
  (let ((count 0))
    (lambda () (setf count (+ count increment))))
```

```
> (setf c2 (make-counter 2))
#<CLOSURE:LAMBDA NIL (SETF COUNT (+ COUNT #<CLOSURE:LAMBDA NIL (SETF COUNT (+ COUNT INCREMENT))>))>
> (funcall c2)
2
> (funcall c2)
4
```

Closure example 3

```lisp
(defun make-counter (&optional (increment 1))
  (let ((count 0))
    (lambda () (setf count (+ count increment))))
```

```
> (setf c2 (make-counter 2))
#<CLOSURE:LAMBDA NIL (SETF COUNT (+ COUNT INCREMENT))>
> (funcall c2)
2
> (funcall c2)
4
```
SO?

- Closure are quite powerful, as we will see.
- In effect, they allow you to suspend execution of a thread by creating a function that will resume it.
- And later call the function to resume computation.
- We’ll look at this when we look at streams.