Let over lambda (lol)

- Let-over-lambda refers to the having a let block whose return value is a lambda function
- Even outside the let block, the returned lambda function can access/update the block's variables, even across multiple calls to the lambda function
- The block's variables persist in memory as long as the lambda function is still accessible somewhere
- Effectively creates a set of hidden variables shared across calls to the lambda function, acting much like the fields of a class plus an access method

Lol example

- Have a let block (with local variables) return a lambda function, store that in a variable, f – the lambda function increments and displays the let block variable

```
(defvar f
   (let ((x 1))
   (lambda () (setf x (+ x 1)) (format t “x is ~A~%” x)))
```

- Call the function repeatedly through f

```
(funcall f)
X is 2
(funcall f)
X is 3
```

How/why it works

- All lists in lisp are dynamically allocated in the heap, and pointers are used to keep track of them
- A list won't be deallocated until there are no more pointers to it (or to elements in it), then lisp automatically deletes it
- The list of local variables in a let block is such a dynamically allocated list, and if the lambda function uses those variables then it has pointers into the list
- As long as the lambda function still exists, its pointers still exist, so the let block's local variable list is kept alive someplace in the heap

More useful lambdas with lol

- Suppose we add parameters to the lambda function that allow the user to specify different things they want done to the ‘hidden’ variables
- Perhaps one command parameter and an option parameter
- The user can call the lambda function repeatedly, having it take different actions on the data over time, e.g. increment, decrement, print, return, etc

Simple example with circles

- Let block variables store the radius of a circle, default value 1, and the lambda function can update it or print it (default action)

```
(defvar f (let ((r 1))
   (lambda (cmd &optional (arg 1))
      (cond ((equal cmd ‘set) (setf r arg))
      (t (format t “r is ~A~%” r))))))
(funcall f ‘set 5)
(funcall f ‘print)
r is 5
```

Combine lol with closures

- Now suppose we had a function, builder, containing the let block from the previous slide and returning its lambda
- The function could take a set of parameters that it used to initialize the let block variables and to customize the lambda function that would be returned, e.g.

```
(defun builder (initialRval areFloatsAllowed)
  ..setup code and a new fancier let block here..)
(defvar f (builder 23.5 t))
```

Our function acts like a constructor

- Every call to a function has its own local variable space
- so every call to builder has its own local variable space

```
(defvar f (builder 23.5 t))
(defvar g (builder 5 nil))
```

- F works on the local variable list allocated for the first call, while g works on the local variable list for the second call
- They’re completely independent ... builder is acting much like a constructor in OO languages

Circle example

- Let's have our lambda function maintain/process data about a circle: the x,y coordinates of the centre and the radius (we'll call the construction function buildCircle)
- The user can give the lambda function commands to print the info, update the coordinates, update the radius, or return the area

```
(defvar c1 (buildCircle 5 3 24)) ; x=5, y=3, r=24
(defvar c2 (buildCircle 0 0 1))  ; x=0, y=0, r=1
(funcall c1 'print)
(5,3):24
```

buildCircle “constructor”

```
(defun buildCircle (&optional (xInit 0) (yInit 0) (rInit 1))
   (let ; start with valid default values
      ((x 0) (y 0) (r 1))
      ; update from parameters if they are valid
      (if (realp xInit) (setf x xInit))
      (if (realp yInit) (setf y yInit))
      (if (and (realp rInit) (> rInit 0)) (setf r rInit))
      ; lambda function expects a command and possibly an arg
      (lambda (cmd &optional (arg nil))
         (cond  ; check/process each command type
```

The lambda function

```
; check for/process print commands
((equalp cmd ‘print) (format t “(~A,~A):~A~%” x y r))

; check for/process area commands, return pi r^2
((equalp cmd ‘area) (* 3.14 r r))

; check for/process set-radius commands
((equalp cmd ‘radius)
   (if (and (realp arg) (> arg 0)) (setf r arg)
       (format t “Error: invalid radius ~A~%” arg)))
```

lambda function cont.

```
; check for/process set-coords commands
((equalp cmd ‘coords)
   ; need to make sure arg is a list of two reals
   (if (and (listp arg) (= (length arg) 2)
       (realp (car arg)) (realp (cadr arg)))
       ; arg looks ok, set x and y
       (setf x (car arg)) (setf y (cadr arg))
       (format t “Error: invalid coords ~A~%” arg)))
; anything else is a bad command
(t (format t “bad command ~A~%” cmd))))); end of buildCircle
```