

Lexical and dynamic variables

- ✓ Suppose we have these functions...

```
(defun scope1 ()  
  x)
```

```
(defun scope2 (x)  
  (scope1))
```

```
(defun scope3 ()  
  (let (x) (scope1)))
```

- ✓ What happens when `scope1` is called?

```
USER: (scope1)
```

```
ERROR: Unbound variable 'X'.
```

- ✓ What happens when `scope2` is called?

```
USER: (scope2 777)
```

```
ERROR: Unbound variable 'X'.
```

- ✓ What happens when `scope3` is called?

```
USER: (scope3)
```

```
ERROR: Unbound variable 'X'.
```

Lexical and dynamic variables

- ✓ Variables in Common LISP are usually *lexically scoped*
- ✓ A lexically scoped variable gets its binding from the most closely surrounding program code context
- ✗ “Lexical” from Gk. meaning “word”... Binding of a lexical variable depends on how the program text is written
- ✗ Variables in a lambda list or introduced in a let, dotimes, or dolist construct are examples of lexically scoped variables
- ✗ To find the binding of a lexical variable, look at the most closely surrounding lexical context which introduces it

```
(defun foo (X)
  (let ((X nil))
    (dotimes (X 13)
      (let ((X (crunch X)))
        (print X))))))
```

```
(defun scope1 ()
  (print X))
```

Lexical and dynamic variables

- ✓ Variables in Common LISP can be *dynamically scoped*
- ✓ A dynamically scoped variable gets its binding from the most recently activated program context
 - ✗ In Common LISP, variables are lexically scoped by default
 - ✗ Dynamically scoped variables must be explicitly or implicitly declared *special*
 - ✗ Binding of a special variable depends on how the program is run
 - ✗ Variables introduced with a `defvar`, `defparameter`, or `defconstant` construct, or declared special with a `declare` construct, are dynamic
 - ✗ To find the binding of a dynamic variable, look at the most recent binding of the variable

Dynamic binding of variables

```
(defun scope2 () X)

(defun scope4 (X) (scope2))

(defun scope5 (Y) (scope2))

USER: (scope4 777)

ERROR: unbound variable 'X'.

USER: (defvar x 888)
X

USER: (scope4 777)
777
USER: (scope5 777)
888
```

```
(defun scope2 () X)

(defun scope4 (X) (declare (special X)) (scope2))

(defun scope5 (Y) (scope2))

USER: (scope4 777)
777

USER: (scope5 777)

ERROR: unbound variable 'X'.
```

Dynamic binding of variables

```
(defun scope2 () X)
```

```
(defun scope4 (X) (declare (special X)) (scope2))
```

```
(defun scope5 (Y X) (scope2))
```

```
USER: (scope5 'baz 888)
```

```
ERROR: unbound variable 'X'.
```

```
USER: (scope4 777)
```

```
777
```

```
(defun scope2 () X)
```

```
(defun scope4 (X) (declare (special X)) (scope2))
```

```
(defun scope5 (Y X) (declare (special X)) (scope2))
```

```
USER: (scope5 'baz 888)
```

```
888
```

```
USER: (scope4 777)
```

```
777
```

Lexical and dynamic variables

- ✓ By convention, special (dynamic) variables in Common LISP are symbols that start and end with ‘*’

```
(defvar *default-database*)
```

- ✓ There are some system variables declared special:

```
*applyhook*, *break-on-signals*, *break-on-warnings*, ...
```

- ✓ This is just a convention: any symbol can be declared special
- ✓ Special variables should be used with care, because the effects of dynamic binding can be hard to debug

Closures

- ✓ A *closure* is a combination of a
 - ✗ function
 - ✗ a set of bindings of free variables in the function
- ✓ If the variable bindings are lexical, it's a *lexical closure*
- ✓ Closures are created with `defun` or with `#'` preceding a lambda expression

Closures

```
(defun foo (x) (list x y))
```

FOO is a closure with **Y** dynamically bound

```
USER: (foo 999)
```

```
ERROR: Unbound variable Y
```

```
USER: (let ((y 555)) (foo 999))
```

```
ERROR: Unbound variable Y
```

```
USER: (let ((y 555)) (declare (special y)) (foo 999))  
(999 555)
```

```
(let ((y 333)) (defun foo (x) (list x y)))
```

FOO is a lexical closure with **Y** lexically bound

```
USER: (foo 999)
```

```
(999 333)
```

```
USER: (let ((y 555)) (foo 999))
```

```
(999 333)
```

Closures

```
(setq C #'(lambda (x) (list x y)))
```

C has as value a closure with **Y** dynamically bound

```
USER: (funcall C 999)
ERROR: Unbound variable Y
```

```
USER: (let ((y 555)) (funcall C 999))
ERROR: Unbound variable Y
```

```
USER: (let ((y 555))
        (declare (special y))
        (funcall C 999))
(999 555)
```

```
(let ((y 333)) (setq C #'(lambda (x) (list x y))))
```

C has as value a lexical closure with **Y** lexically bound

```
USER: (funcall C 999)
(999 333)
```

```
USER: (let ((y 555)) (funcall C 999))
(999 333)
```

Lexical closures

- ✓ Various programming languages permit a function to return a function

```
(defun make-adder (n)
  (cond
    ((= n 1) #'(lambda (x) (+ 1 x)))
    ((= n 2) #'(lambda (x) (+ 2 x)))
    ((= n 3) #'(lambda (x) (+ 3 x)))
    (t (error "don't know how to make that adder"))))
```

- ✓ Lexical closures in LISP permit creating more flexible functions with local state

```
(defun make-adder (n)
  #'(lambda (x) (+ n x)))
```

- ✓ Note that this requires a lexical closure... if `n` in `#'(lambda (x) (+ n x))` were dynamically bound, its value would depend on the dynamic environment when `#'(lambda (x) (+ n x))` is applied... and the dynamic environment is difficult to control

Lexical closures

✓ Some interesting uses of lexical closures...

- ✗ A function that takes a predicate as argument, and returns a predicate that is its logical complement

```
(defun complement (pred)
  #'(lambda (&rest args)
      (not (apply pred args))))
```

Now, instead of

```
(remove-if-not #'oddp '(1 2 3 4 5 6))
```

you can write

```
(remove-if (complement #'oddp) '(1 2 3 4 5 6))
```

Lexical closures

- ✓ Creating resettable counters...

```
(defun make-counter ()  
  (let ((count 0))  
    (list  
      #'(lambda () (setq count 0))  
      #'(lambda () (setq count (+ 1 count)))  
      #'(lambda () count)))))
```

- x Now `(make-counter)` returns a list of three lexical closures
- x The first element of the list is a function which resets the counter to 0
- x The second element of the list is a function which increments the counter
- x The third element of the list is a function which returns the value of the counter
- x Each of the three functions acts on the same lexical variable
- x But this lexical variable is different from the variable in the closures returned by another call to `make-counter`...

Lexical closures

```
USER: (setq counter1 (make-counter))
```

```
USER: (setq counter2 (make-counter))
```

```
USER: (funcall (second counter1))  
1
```

```
USER: (funcall (second counter1))  
2
```

```
USER: (funcall (third counter1))  
2
```

```
USER: (funcall (third counter2))  
0
```

Lexical closures

- ✓ Creating generators...
- ✓ A generator is a function that returns the next object in a sequence each time it is called
 - ✗ the whole sequence may be infinite... and so it cannot be created all at once

```
(defun make-generator ()  
  (let ((state (initialize-state)))  
    #'(lambda ()  
        (let ((result (hairy-computation state)))  
          (setf state (next-state state))  
          result))))
```

Now...

```
USER: (setq gen (make-generator))
```

```
USER: (funcall gen)  
<first value of sequence>
```

```
USER: (funcall gen)  
<second value of sequence>
```

```
USER: (setq gen2 (make-generator))
```

```
USER: (funcall gen2)  
<first value of sequence>
```

```
USER: (funcall gen)  
<third value of sequence>
```