Allegro CL Certification Program

Lisp Programming Series Level 2

Franz Inc.

Goals for Level 2

- Build on Level 1
  - Assumes you can write functions, use the debugger, understand lists and list processing
- Detailed Exposure To
  - Functions
  - Macros
  - Object-oriented programming

Format of the Course

- One 2-hour presentation each week
  - Lecture
  - Question and answer
  - Sample code
- Lecture notes available online (http://www.franz.com/lab/)
- Homework
- One-on-one help via email

Session 1 (today)

- Advanced features of Lisp functions
- Structures, Hash Tables, Bits and Bytes
- Macros
- Closures

Session 2

Common Lisp Object System (CLOS)

- Top Ten things to do in CLOS
- Classes, instances, and inheritance
- Methods
- Class precedence list
- Programming style

Session 3

- Performance considerations with CLOS
- The garbage collector
- Error conditions
- Using the IDE to make Windows™ windows

Homework

- Lisp is best learnt by hands-on experience
- Many exercises for each session
- Will dedicate class time to reviewing exercises
- Email instructor for one-on-one assistance doing the homework or any other questions relevant to the class
  - training@franz.com

Getting Allegro Common Lisp

- This class is based on version 6.2.
- Trial version should be sufficient for this module
  - Download free from `http://www.franz.com/`
  - Works for 60 days
- I will be using ACL on Windows
- You can use ACL on UNIX, but the development environment is different and I won't be using it.

Allegro CL Certification Program

Lisp Programming Series Level 2

Session 2.1.1

Functions

Ways to define functions

- defun
- lambda
- flet
- labels

Defun

```
(defun double (x)
  "Doubles the number"
   (declare (fixnum x))
   (+ x x))
```

(defun fn-name arglist

optional-doc-string

optional-declarations

code)

Argument List

- Contains names for variables
  - that get their values from arguments provided in calls to the function
  - persist throughout the body of the function
  - are not typed
- May contain optional and keyword arguments

Keyword Arguments

```
(defun print-array-size (array
                         &key
                         (stream *standard-output*)
                         UseGraphics)
  (let ((size (length array)))
    (if UseGraphics
        (drawsize size stream) ; then
       (print size stream))    ; else
    size)))
```

- Call the function like this:
  - `(print-array-size *my-array*)`
- Or like this:
  - `(print-array-size *my-array* :stream my-open-file)`

Keyword Arguments

- Call the function with any number of the keyword arguments, in any order
- Arguments specified as name/value pairs
  - `:stream my-open-file`
- Costs a couple microseconds
  - Because Lisp must parse the argument list
- Use when some arguments are only rarely needed
  - Source code in callers is greatly simplified
- Or when different calls will need different groups of arguments

Keyword Detail

```
&key (stream *standard-output*) UseGraphics
```

- `&key` specifies the arguments that follow are keyword arguments
- Each argument can either be a two-element list, a three-element list, or a single symbol
  - Three-element list is used for *supplied-p* arguments, described later on
- Default value of an argument, when not passed by the caller, is the second element of the two-element list, or `NIL` if not otherwise specified

Optional Arguments

```
(defun print-array-size (array
                         &optional
                         (stream *standard-output*)
                         UseGraphics)
 (let ((size (length array)))
    (if UseGraphics
        (drawsize size stream) ; then
      (print size stream))     ; else
    size)))
```

- Call the function like this:
  - `(print-array-size *my-array*)`
- Or like this:
  - `(print-array-size *my-array* my-open-file)`

Optional Arguments

- Call the function with any number of the optional arguments
- Lisp determines which one is which ***by position***
  - To specify the second optional argument, you must always specify the first as well
- Costs a couple microseconds
  - Because Lisp must parse the argument list
- Use when some arguments are only rarely needed, and when there aren't very many of them
  - Source code in callers is greatly simplified

Other possibilities

```lisp
(defun incr (n &optional inc)
   ;; if the value of inc was not specified,
   ;; or if it was specified but not a number,
   ;; then set it to 1.
   (if (not (numberp inc)) (setq inc 1))
   (+ n inc))

(defun incr (n &optional (inc 1 increment-p))
  ;; increment-p TRUE when argument was specified
  ;; by caller
  (if (and increment-p (not (numberp inc)))
      (print "non-numeric increment"))
      (+ n inc))
```

Optional and Keyword Arguments

- Ways to specify the argument
  - <name> -- defaults to `nil`
  - (<name> <default value>)
  - (<name> <default value> <supplied-p>)
- Use optionals for small numbers of non-required arguments that are easy to remember
- Use keywords with either large numbers of non-required arguments or ones that are hard to remember

&rest example

```
(defun add (&rest numbers)
  (let ((sum 0))
    (dolist (n numbers)
      (setq sum (+ sum n)))
    sum))
```

- Call it this way:
  - `(add 1 2 4)`
- Or this way:
  - `(add 9 1 8 4 8 6)`

&rest Arguments

- Caller may pass any number of arguments, there is no limit
- Lisp combines all the arguments into a single list
- Needed only rarely
- Generally used when all the arguments have the same type and will participate equally in the operation

Pointers to Functions

- The usual way of calling a function:
  - `(+ 3 4)`
- Alternative way:
  - `(funcall #'+ 3 4)`
- `#'+` is a pointer to the function "+"
- Function pointers can be passed as arguments, returned from functions, or stored as data

Pointers to Functions

- `#’add`
  - Is a reference to the function named ADD

```
(defun combiner (n)
  (if (typep n 'list) #'list #'+))
;; Example of returning a function pointer
;; from a function.

(setq *combiner* (combiner 3))
```

Calling a Function by its Pointer

- #'+
  - #<FUNCTION +>

- (funcall #'+ 3 4 5 6)
  - 18
- (apply #'+ (list 3 4 5 6))
  - 18

Using Function References

- Functions can be used just like any other type object

```
(defun combiner (x)
   (typecase x (number #'+) (list #'append) (t #'list)))
COMBINER
USER(49): (defun combine (&rest args)
               (apply (combiner (first args)) args))
COMBINE
USER(50): (combine 2 3)
5
USER(51): (combine '(a b) '(c d))
(A B C D)
USER(54): (combine 'a 'b 'c)
(A B C)
```

Function References

- Commonly used in sorting routines
  - `(sort '(5 1 4 6) #'<)`
  - `#'<` small numbers first
  - `#'>` big numbers first
  - `#'string<` A before Z
  - `#'string>` Z before A
- Often used when mapping over a collection (vector, list, hash table, etc.)
  - `(mapcar #'print '(5 1 4 6))`

Lambdas

- Nameless fns
- `(setq function #'(lambda (x y) (* x y)))`
- `(setq function #'(lambda (x y) (+ x y)))`
- `(funcall function 3 4)`

Lambda - Example

- Example

```
(defun increment-all (&rest numbers)
  (mapcar #'(lambda (x) (+ x 1))
          numbers))
```

- (increment-all 5 6 7 8)
  - (6 7 8 9)

Where do you use lambdas

- When a function will have only one caller and it is relatively trivial
- Saves using up you namespace
- Seeing the code in place may improve readability in some situations
- Commonly used with mapping functions

Other Functions of Functions

- Many tools available to investigate your environment

fboundp

- Determines if a symbol is associated with a function definition or function binding

```
(fboundp 'double)
#<Interpreted Function DOUBLE>
```

fdefinition

- `(fdefinition '+)` retrieves the function object associated with the argument. Same as symbol-function for symbols, but also works on names that are lists (such as `(setf foo)`).
- Function-lambda-expression retrieves the definition, if it is available (but the argument must be a function object, not a function name)

fmakunbound

- `(fmakunbound 'add)`
- makes the function named ADD become undefined
- analogous to `makunbound` for variables

Symbol-function

- Returns a symbol's fn
- Note: you can also setf this

```
> (setf (symbol-function 'double) #'(lambda (x) (+ x x)))
#<Interpreted Function DOUBLE>
> (double 5)
10
```

Global .vs. Local Functions

- Defun’s are global
- Lambda’s can only be used in place unless the are assigned or stored on something
- flets and labels can be used to create fns that are only available in a local context

Local Functions - labels

- Enables you to define a function and use it within the scope of the labels
- Think of it like a let* for functions

```
(defun test-3rd ()
   (labels ((3rd (lst)
                   (first (rest (rest lst)))))
     (print (3rd '(1 2 3 4)))
     (print (3rd '(a b c d)))))
```

SETF Functions

- `setf` is useful when you have a pair of operations for getting and setting a value
- In other languages, you would name the pair Get-xxx and Set-xxx
- With `setf`, you have only one name
- `(first list) ; gets first element of list`
- `(setf (first list) 17) ; sets first element to 17`

SETF Function Definition

```
;;; Get the first element
(defun 1st (list)
  (car list))

;;; Set the first element
(defun (setf 1st) (new list)
  (rplaca list new)
  new)
```

Using SETF

```
(setq a '(one two three))
(1st a)
  -> ONE
(setf (1st a) 1)
  -> 1
(1st a)
  -> 1
a
  -> (1 two three)
```

Many Operations in Lisp itself are setf-enabled

- `first`, `second`, `third`, `last`, `nth`, `elt`
- `aref`, `fill-pointer`
- Elements or fields of a structure
- Elements or slots of a CLOS instance
- `symbol-value`, `symbol-function`
- `gethash`

The Idea of Mapping Functions

- You pass a “pointer to a function” as one of the arguments
- The “pointer to a function” is applied to each element of a collection
- The results of the individual calls may be collected up into a list

mapcar

- `(mapcar fn list &rest more-lists)`
- Example:

```
> (mapcar #'print '(a b c))
A
B
C
(A B C)
> (mapcar #'cons '(a b c) '(1 2 3))
((A . 1) (B . 2) (C . 3))
```

maphash

- `(maphash` function hash-table`)`
  - Hash tables covered in more detail later

```
(maphash #'(lambda (key value)
              (format t "~&k=~A,v=~A" key value))
              ht)
```

The Idea of Multiple Values

- Sometimes you want a function to return more than one value
- Option 1: Make an object that contains the values
  - `(defun foo () (list 1 2))`
- Option 2: Use Lisp multiple values
  - `(defun foo () (values 1 2))`
  - 1 is said to be the "first value"
  - 2 is said to be the "second value"

multiple-value-setq

- Use it like `setq` to capture multiple values

```
(let ((x 0)(y 0))
   (print x) (print y)  ; values before

   (multiple-value-setq (x y) (foo))

   (print x) (print y)  ; values after
   )
```

Multiple-Value-Bind

- Use it like `let` or `let*` to capture multiple values in new local variables

```
(multiple-value-bind (left right) (foo)
   (print left)
   (print right))
```

Allegro CL Certification Program

Lisp Programming Series Level 2

Session 2.1.2

Structures and Hash Tables

User-defined structures

- Can either use CLOS instances or defstructs
  - both store data in a user-defined form
  - instances can have behavior in addition
  - defstructs are more efficient, but less powerful
- Designed to look similar in code
  - slot references look like function calls
  - can switch between them during development

Structures

- Define with defstruct
- create one with (make-<name> …)
- access slot with <name>-<slotname>

```
> (defstruct route-segment
    node-number
    start
    end)
ROUTE-SEGMENT
> (setf rs (make-route-segment :node-number 5 :start 2 :end 7))
#S(ROUTE-SEGMENT NODE-NUMBER 5 START 2 END 7)
> (route-segment-start rs)
2
```

defstruct with default values

- Example

```
> (defstruct point
    (x 0)
    (y 0))
POINT
> (setf pt (make-point))
#S(POINT X 0 Y 0)
```

Structures

- Standard Lisp Structures are really just vectors.
- Accessing an element of a structure is as fast as accessing an element of an array.
- The reader functions generated by `defstruct` get compiled into an vector access of a specific position (fast!)
- If you redefine the positions, you have to recompile your code

defstruct with type

- Example

```
> (defstruct (point (:type list))
      (x 0)
      (y 0))
POINT
> (setf pt (make-point :x 10 :y 20))
(10 20)
```

defstruct with type and name

- Example

```
> (defstruct (point (:type list) (:named t))
        (x 0)
        (y 0))
POINT
> (setf pt (make-point :x 10 :y 20))
(POINT 10 20)
```

Hash tables

- Pair-oriented: Associate keys with values, just like alists and plists
- Could use lists for small ones, but search time grows proportional to size of list
- hash table computes hash function to use as index
  - speed largely independent of size
- have to build your own in many other languages

Hash table examples

```
> (setf ht (make-hash-table))
#<HASH-TABLE #xDD7410>
> (gethash 'color ht)
NIL
NIL
> (setf (gethash 'color ht) 'brown)
BROWN
> (gethash 'color ht)
BROWN
T
```

gethash's Multiple values

- `gethash` returns two values
  - value associated with key or `NIL`
  - Whether or not the value was found
- Second value helps you distinguish between
  - `NIL` as the value of the key
  - `NIL` as the value you get when the key has no value

Hash Table Fns

- Examples

```
> (hash-table-p ht)
(#<STRUCTURE-CLASS HASH-TABLE #x891668>)
> (gethash 'color ht)
BROWN
T
> (remhash 'color ht)
T
> (gethash 'color ht)
NIL
NIL
```

clrhash

- Examples:

```
(setf ht (make-hash-table))
#<HASH-TABLE #xE02D8C>
> (setf (gethash 'color ht) 'brown)
BROWN
> (gethash 'color ht)
BROWN
T
> (clrhash ht)
#<HASH-TABLE #xDD8280>
```

Hash table iteration example

```
(let ((test (make-hash-table)))
   (setf (gethash 'a test) "This is the a value")
   (setf (gethash 'b test) "This is the b value")
   (maphash #'(lambda (sym str)
                 (format t "~&~A = ~S" sym str))
            test))
B = "This is the b value"
A = "This is the a value"
NIL
```

maphash

- Iterate over the contents of the hash table, pair by pair

- `(maphash #'(lambda (key value)  ..code..) hash)`

Allegro CL Certification Program

Lisp Programming Series Level 2

Session 2.1.3

Bits and Bytes

Bits and Bytes

- Normally represented as lisp integers
- Often used for efficiency
  - Speed: some operations may compile into a single machine instruction
  - Size: a bit vector is much smaller than a general vector
- Often used in combination with foreign function calls
  - Arguments to C++ and WIN32 libraries are often several "flags" passed as a single integer

Bits of Integers

- The `#b` prefix means a binary notation

```
USER(1): #b10
2   ;  decimal integer 2
USER(2): *print-base*
10     ; numbers normally print in decimal
USER(3): (let ((*print-base* 2)) (print #b10) nil)
10     ;  decimal integer 2 printed in binary
NIL
USER(4):
```

Bit Combination

- Inclusive OR of bits

```
USER(1): (logior #b100 #b110)
#b110
```

- AND of bits

```
USER(1): (logand #b100 #b110)
#b100
```

- Less commonly: `logxor`, `logeqv`, `lognand`, `lognor`, `logandc1`, `logandc2`, `logorc1`, `logorc2`

Bit Testing

- `(defvar *mask* #b1010)`
- `(logtest flags *mask*)`
  - True if the second or fourth bit in FLAGS is "on"
- `(logbitp 1 flags )`
  - True if the second bit in FLAGS is "on"
- `(logcount flags)`
  - Counts the number of bits that are "on"

Byte Manipulation with ldb

- `USER(1): (setq flags #b111000111)`
- `USER(2): (ldb (byte 4 0) flags)`
- `#b0111   ; lowest (rightmost) four bits`
- `USER(3): (ldb (byte 4 4) flags)`
- `#b1100   ; next four bits`
- `USER(4): (ldb (byte 8 0) flags)`
- `#b11000111 ; lowest eight bits`

Byte Manipulation

```
USER(5): (setf (ldb (byte 4 4) flags) #b0011)
USER(6): flags
#b100110111
```

- This line modifies the second four bits of the bit field.

Shift Operation

- `ash` -- arithmetic shift (left)
  - `(ash 1 10) --> 1024`
  - `(ash 255 -6) --> 3`

- Note that there is no assumption of integer size. You eventually get a bignum if you keep shifting left.

How Many Bits?

- `(integer-length #b1000)  => 4`
- Use it to print a binary number:

```
(defun binary-to-string (bits)
  (let* ((L (integer-length bits))
         (string (make-string L
                      :initial-element #\0)))
   (dotimes (I L)
      ;; Note that bit zero is on the right
      ;; of the string (character L-1).
      (when (logbitp (- L I 1) bits)
         (setf (char string I) #\1)))
    string))
```

Vectors of Bits

```
(setq vector (make-array 1024 :element-type 'bit
                              :initial-element 0))

;; Access and modify as any vector or array
(setf (aref vector 0) 1)

;; But elements must be either zero or one
(setf (aref vector 0) 2)  ; ERROR
```

Allegro CL Certification Program

Lisp Programming Series Level 2

Session 2.1.4

Macros

What are Macros?

- Macros take lisp code as input and return lisp code as output. For example,

When evaluating: `(incf x)`

Evaluate this instead: `(setf x (+ 1 x))`

```
(defmacro incf (place)
  (list 'setf place (list '+ 1 place)))
```

Macroexpansion

- When the evaluator sees `(incf a)`
  - It notices that `INCF` names a macro
  - It “runs” or macroexpands the macro, which transforms the line of code into:
    - `(setf a (+ 1 a))`
  - It evaluates that expression instead
  - So when you type `(incf a)` to the lisp listener, it is as if you had typed `(setf a (+ 1 a))`

Macro Evaluation is Different

- for functions
  - gets the function name
  - evaluates all the args
  - applies the function to the eval'ed args
- for macros
  - passes arguments without evaluating them
  - the macro function returns another expression
  - evaluator evaluates that expression instead of the original

Recursive Macros

- Macros can macroexpand into other macros
- For example
  - `WHEN` macroexpands into `COND`
  - `COND` macroexpands into `IF`
- The evaluator (and the compiler) recursively macroexpand an expression until there are no more macros left

Macroexpand function

- `macroexpand` is function which lisp uses to call macro function and get result
  - it keeps recursively macro-expanding till no macros are left
- `macroexpand-1` just does one step of macroexpansion
- `(macroexpand-1 '(incf x))`
  - `(setq x (+ x 1))`

macro functions

- stored in same function cell of symbol
- stored in a different format so that the system can tell it is a macro function
- macro-function <symbol> will return `nil` if the symbol has a normal function definition or none, but will return the expansion function if the symbol names a macro

Macro Examples

- Macros are just functions that transform expressions
- Use `macroexpand-1` to see definition

```
> (defmacro nil! (x)
     (list 'setf x nil))
NIL!
> (setq x 5)
5
> (nil! x)
NIL
> x
NIL
> (macroexpand-1 '(nil! x))
(SETF X NIL)
```

Backquote

- Used extensively in macros
- Used by itself is equivalent to quote
- Protects args from evaluation
- comma (`,`) will unprotect

```
> (setq a 1 b 2)
2
> `(a is ,a b is ,b)
(A IS 1 B IS 2)
```

Backquote example

```
(defmacro incf (place)
  `(setf ,place (+ 1 ,place)))
```

- Compared to earlier definition of **`INCF`**, this version is shorter, more concise, and easier to understand (but equivalent)

```
(defmacro incf (place)
  (list 'setf place (list '+ 1 place)))
```

,@

- Like comma but splices in list

```
> (setq lst '(1 2 3 4))
(1 2 3 4)
> `(here are the numbers ,@lst)
(HERE ARE THE NUMBERS 1 2 3 4)
```

&body

- `&body` is like `&rest`, but typically reserved for macros
- Example: `WITH` (shorthand for `LET`, use it to create one local variable)

```
(defmacro with ((var &optional val) &body body)
  `(let ((,var ,val))
      ,@body))

(with (a)
  (print a))   ;; this example transforms into:

(let ((a nil))
   (print a))
```

with-open-file example

More complex example. There is a built-in lisp macro of the same name that does almost exactly this.

```
(defmacro with-open-file ((var &rest args) &body body)
   `(let ((,var (open ,@args)))    ; open file
      (unwind-protect
         (progn ,@body)            ; execute body
         (when (streamp ,var) (close ,var)))) ; close
```

- `unwind-protect` is talked about later on, but it ensures the file is closed even if an error occurs

Macro Examples

- Ordered-bounds

```
(defmacro order-bounds (left bottom right top)
  `(progn (if (> ,left ,right) (rotatef ,left ,right))
          (if (> ,bottom ,top) (rotatef ,bottom ,top))))

(setq left 10)
(setq right 0)
(setq top 50)
(setq bottom 4)
(order-bounds left bottom right top)
;; Now LEFT is 0, RIGHT is 10,
;; TOP is 4, BOTTOM is 50
```

Macro Examples

- Add onto the end of the list

```
(defmacro push-last (item list)
  `(setf ,list (nconc ,list (list ,item))))
```

Macro Examples

- Atomic Operations

```
(defmacro atomic-pop (list)
  `(without-interrupts
     (pop ,list)))

(defmacro atomic-push (item list)
  `(without-interrupts
     (push ,item ,list)))
```

Iteration Macro

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
       ,@body))

;; Prints even numbers
(setq I 0)
(while (< I 10)
  (print I)
  (incf I 2))
```

Macro Argument Lists

- Use of `&key`, `&optional`, `&rest` is common in macros

```
(defmacro with-resource-string ((resource-string
                                  &key (size 80))
                                  &body body)
  `(let ((,resource-string
          (allocate-resource-string :size ,size)))
     ,@body
     (free-resource-string ,resource-string)))
```

Macros with Logic

- A macro need not be just a backquoted list
- A macro is an arbitrarily complex function for transforming one expression into another

```
(defmacro incf (place)
  (if (symbolp place)
      `(setq ,place (+ 1 ,place))
      `(setf ,place (+ 1 ,place))))
```

Macro writing problems

- A macro is not a function
  - Certain uses are not allowed
- multiple evaluation problem
  - Inadvertently evaluate args multiple times
- variable capture problem
  - Inadvertently shadow a variable name

Macros are not Functions

- `(apply #'when (> x 3) x)`
  - This is an error because APPLY only works on functions

Don't evaluate more than once

- A macro similar to OR

```
(defmacro or1 (a b)
  `(if ,a ,a ,b))
```

- What happens for: (or1 (print 1) (print 2))

```
   (if (print 1) (print 1) (print 2))
```

- To avoid multiple evaluation:

```
(defmacro or2 (a b)
  `(let ((temp ,a))
     (if temp temp ,b)))
```

FRANZ INC.

Variable Capture

- How would you implement Lisp's OR macro?

```
(defmacro or2 (a b)
  `(let ((temp ,a))
     (if temp temp ,b)))

(let ((x nil)
      (temp 7))
  (or2 x temp))
;; Returns NIL (there are two TEMPs)
```

Generate symbols that can't be captured

- Gensym

```
(defmacro or3 (a b)
  (let ((symbol (gensym)))
    `(let ((,symbol ,a))
       (if ,symbol ,symbol ,b))))
```

Turning Functions into Macros

- Do it to eliminate a function call
- Do it when the function is not recursive

```
(defun second (x) (cadr x))
(defmacro second (x) `(cadr ,x))

(defun sum (&rest numbers) (apply #'+ numbers))
(defmacro sum (&rest numbers) `(+ ,@numbers))
```

When to Use Macros

- Macros help avoid code duplication

```
(defmacro with-resource-string ((resource-string
                                 &key (size 80))
                                 &body body)
  `(let ((,resource-string
          (allocate-resource-string :size ,size)))
     ,@body
     (free-resource-string ,resource-string)))
```

When to use macros

- You have to use macros when
  - you need to control evaluation
    - binding (like local variables in LET)
    - conditional evaluation (like AND or OR or IF)
    - looping (like DO)
    - Simplification without a function call (like (SETF CAR) expanding into RPLACA)
- You can use macros to
  - do computation at compile-time
  - expand in place and avoid a function call
  - save typing or code duplication, and to clarify code

Problems with using Macros

- You cannot use a macro if you have to funcall or apply it
- Macro definitions are harder to read
- Macro definitions can be harder to debug
  - The code you see in the backtrace may bear little resemblance to your source code
- Although macros can expand recursively into other macros, you can't usually write a recursive algorithm with them.

Redefining Macros

- Code for macro expansion captured in compiled files of callers of the macro
- If you change the definitions of the macro itself, you have to recompile the callers
- Defsystem tool allows you to record these dependencies once

learning more

- A lot of Common Lisp is really implemented as macros
- Looking at the expansions of these can teach you a lot about how macros work
- `(pprint (macroexpand-1 '(defun foo (a) (+ a 1))))`

Allegro CL Certification Program

Lisp Programming Series Level 2

Session 2.1.5

Macro Pitfalls and Issues

A Macro is not a Function

- `(apply #'when (> x 3) (print x))`
  - This is an error

Macro Recursion is not like Function Recursion

```
(defmacro nth* (n list)
  `(if (= ,n 0) (car ,list)
       (nth* (- ,n 1) (cdr ,list))))
```

- macroexpanding `nth*` will macroexpand forever when compiled in a function like `(defun foo (x l) (nth* x l))`
- Think of the code you want the macro to expand into

Valid Macro Recursion

```
(defmacro or* (&body body)
          (cond ((null body) 'nil)
                ((null (cdr body)) (car body))
                (t (let ((temp (gensym)))
                     `(let ((,temp ,(car body)))
                        (if ,temp ,temp
                          (or* ,@(cdr body))))))))
```

- `(or* a b)` expands into

```
(let ((#:g24 a))
  (if #:g24 #:g24 b))
```

Multiple Evaluation

This definition of OR evaluates A twice

```
(defmacro or* (a b)
  `(if ,a ,a ,b))
```

Do it this way instead

```
(defmacro or* (a b)
  `(let ((temp ,a)) (if temp temp ,b)))
```

Order of Evaluation

- Evaluation order should be left to right

```
(defmacro and* (a b)
  `(let ((temp2 ,b) (temp1 ,a))
     (if (not temp1) nil
         (if (not temp2) nil temp2))))

(and* (setq x 2) (setq x 3))
;; Returns 3 but x is 2!
```

Avoid Destroying Arg Lists

```
(defmacro sum-plus-1 (&rest args)
   (cons '+ (nconc args (list 1))))
(defun foo () (sum-plus-1 2 3))
(foo) returns 6
(foo) returns 7
(foo) returns 8
```

- Because the macro is destructively modifying the source code of the caller
- The source code stops changing when you compile it

Variable Capture

- Using the `OR*` macro from a few slides back…

```
(setq x nil)
(let ((temp 7))
  (or* x (+ temp 1)))
```

- This code attempts to add NIL to 1
- Because the macroexpansion of `OR*` causes TEMP to get rebound to NIL

Global Variable Capture

```
(defvar width 5)  ; global variable
(defmacro notice-width (w)
  `(setq width ,w))
(defun draw-rectangle (x y width height)
  (notice-width width)
  (notice-height height)
  . . .)
```

- The macro does not affect the global variable as intended

Avoiding Global Variable Capture

- `(defvar *width* 5)`

- Use naming conventions that distinguish local from global variables.

Avoiding Variable Capture with Gensym

```
(defmacro or* (a b)
  (let ((symbol (gensym)))
    `(let ((,symbol ,a))
       (if ,symbol ,symbol ,b))))
```

Avoiding Variable Capture with Scope

```
(defmacro sum-squares-w (x y)
  `(let* ((x0 ,x)     ; WRONG (X0 captured)
          (y0 ,y)     ; problem occurs in this line
          (x2 (* x0 x0))  ;; if form y refers
          (y2 (* y0 y0))) ;; to x0
    (+ x2 y2)))
(defmacro sum-squares-r (x y)
  `(let ((x0 ,x)      ; RIGHT
         (y0 ,y))
     (let ((x2 (* x0 x0))
           (y2 (* y0 y0)))
       (+ x2 y2))))
```

Variable capture example

```
(let ((x0 5))
    (* (sum-squares-w
           1 (- (setq x0 10) 9)) x0))
505
(let ((x0 5))
     (* (sum-squares-r
           1 (- (setq x0 10) 9)) x0))
20
```

SETF: a special kind of macro

```
> (setq a nil) ; setq only for symbols
a
> (setf a '(one two three)) ; setf of a symbol
(ONE TWO THREE)
> (setf (first a) 1) ; setf of a "place"
1
> A                   ; list was permanently changed
(1 2 3)
```

Builtin SETF operations

- Lisp knows how to use `SETF` with many things (but not everything)
  - Lists: `first`, `second`, `elt`, `nth`, `car`, `cdr`
  - Arrays: `aref`
  - Objects: `slot-value`
  - Bits: `ldb`
  - Hash tables: `gethash`

Rolling Your Own

- Define your own SETF procedures in one of two ways:

- By functions and methods:

```
(defmethod (setf x) (new (object point))
  (setf (slot-value object 'x) new))
```

- By `defsetf` macros (next slide)

Using DEFSETF

```
(defsetf car (x) (new)
  `(progn (rplaca ,x ,new) ,new))

(defsetf x (point) (new)
  `(setf (aref ,point 1) ,new))
```

Allegro CL Certification Program

Lisp Programming Series Level 2

Session 2.1.6

Closures

What Are Closures?

- Closures are
  - Executable functions
  - Objects with state
- They usually appear as lambda expressions
- Nothing like them in C, C++, Java, VB, or any other traditional language
- They are functions with a "memory"

A Simple Closure

```
(setq closure
  (let ((list '(one two three four five)))
    #'(lambda () (pop list))))
⇒ #<Interpreted Closure (unnamed) @ #x2083818a>
(funcall closure)
⇒ ONE
(funcall closure)
⇒ TWO
(funcall closure)
⇒ THREE
```

What Happened?

- `LAMBDA` created an unnamed function
- To observe the rules of lexical scoping, the function can continue to reference `LIST` even after returning from the `LET`
- The function makes a “snapshot” of the variable at the time it is evaluated
- The function carries that snapshot with it as a segment of data

Another Example

```
(let ((list '(one two three four five)))
  (setq closure1 #'(lambda () (pop list)))
  (setq closure2 #'(lambda () (pop list))))

;; two closures, both with a reference to LIST

(funcall closure1)
⇒ ONE
(funcall closure2)
⇒ TWO
```

Now What Happened?

- The two closures both reference a single, shared closure variable
- They each can see modifications that the other closure makes

Implicit Closures

- Closures happen implicitly when ever a function refers to something in the lexical environment

```
> (defun add-to-list (num lst)
     (mapcar #'(lambda (x) (+ x num)) lst))
ADD-TO-LIST
```

Closures and Garbage

- Note that creating a closure allocates memory and can be a source of garbage and slowness.
- If the closure can be allocated on the stack, then do so using dynamic-extent.

```
(defun add-to-list (num list)
  (labels ((adder (x) (+ x num)))
     (declare (dynamic-extent #'adder))
     (mapcar #'adder list)))
```

Closure Example 1

- The adder fn:

```
> (defun add-to-list (num lst)
     (mapcar #'(lambda (x) (+ x num)) lst))
ADD-TO-LIST
> (defun make-adder (n)
     #'(lambda (x) (+ x n)))
MAKE-ADDER
> (setf add5 (make-adder 5))
#<closure 1 #xDDF914>
> (setf add15 (make-adder 15))
#<closure 1 #xDE2F34>
> (funcall add5 1)
6
> (funcall add15 1)
16
```

Closure Examples 2

- Closures that share variables

```
> (let ((counter 0))
       (defun reset ()
          (setf counter 0) counter)
       (defun stamp ()
          (incf counter) counter))
STAMP
> (list (stamp) (stamp) (reset) (stamp))
(1 2 0 1)
```

Closure Examples 3

- Complement Example

```
> (defun my-complement (fn)
     #'(lambda (&rest args)
           (not (apply fn args))))
MY-COMPLEMENT
> (mapcar (my-complement #'oddp)
                   '(1 2 3 4 5 6 7 8))
(NIL T NIL T NIL T NIL T)
>
```

Cool Example

- Object-oriented programming with closures
- Invented by Guy Steele in 1976

```
(defun make-account (&key (balance 0.0))
  "Create an ACCOUNT object with one slot, BALANCE"
  #'(lambda (message &rest args)
      (case message
        ;; Object supports three methods or messages
        (:deposit (incf balance (car args)))
        (:withdraw (decf balance (car args)))
        (:balance balance))))

(defun send (object message &rest args)
  (apply object message args))
```

Example

```
USER(15): (setq my-account
             (make-account :balance 125.00))
#<Closure (:INTERNAL MAKE-ACCOUNT 0) @ #x208490b2>
USER(16): (send my-account :balance)
125.0
USER(17): (send my-account :deposit 10.0)
135.0
USER(18): (send my-account :balance)
135.0
```

URL for homework and slides

http://www.franz.com/lab/