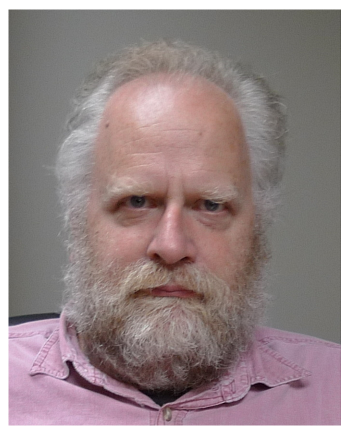
Allegro CL Certification Program

Lisp Programming Series Level I

Review

David
Margolies



Summary 1

- A lisp session contains a large number of objects which is typically increased by user-created lisp objects

- The user works with
  - a read-eval-print loop which is provided as part of the lisp session
  - an editor (preferably a lisp-knowledgable editor)
- Writes code
  - directly in the read-eval-print window
  - in the editor from which code can be saved and can be modified

- Brings the code into the lisp world by entering it directly to the read-eval-print loop or by loading it from the editor or files
- Code loaded into lisp may be
  - interpreted - loaded as source
  - compiled before loading

Summary 2

- Definitions written in Common Lisp can be compiled.
- A Common Lisp compiler can be applied to files or individual definitions
- Compiling a file of Common Lisp source code, say myfile.cl, creates a file myfile.fasl
- (load "myfile.cl")
- (compile-file "myfile.cl")
- (load "myfile.fasl")
- (load "my-app2/my-other-file.cl")
- (load "c:\\program files\\acl62\\still-another-file.cl")

- Compiling a file does NOT make it part of any lisp session
- A definition created by typing directly to the read/eval/print loop does not create compiled code.

```
(defun my-func (arg)
    (* 73 arg))
```

- The interpreted definition can be replaced in the same lisp session by calling compile on the name of the function

```
     (compile 'my-func)
```

- Incremental compilation while using Allegro CL both compiles the designated code and loads the newly compiled definitions into the current lisp session.

Summary 3

- A lisp application written in Allegro CL can be delivered as
  - source code to anyone else who has a copy of a compatible Common Lisp
  - one or more compiled files to anyone else who has the same version of Allegro CL for the same kind of operating system
  - a standalone application for use on the same kind and similar version of operating system

Format

```
cg-USER(43): (format t "Hi, I'm David")
Hi, I'm David
NIL
CG-USER(44): (format t
                      "~%Hi, I'm ~a"
                       'david)


Hi, I'm DAVID
NIL
CG-USER(45): (format
                 nil
                 "Hi, I'm ~a" 'david)
"Hi, I'm DAVID"
```

Format cont'd

```
CG-USER(50):
(let ((radius 14))
    (format
        t
       "~%The circumference of a circle with ~
        radius ~d is ~%~f"
        radius (* 2 pi radius))
    (format t "~%The area of that circle is ~f"
         (* pi (* radius radius))))

The circumference of a circle with radius 14 is
87.96459430051421d0
The area of that circle is 615.752160103599 4d0
NIL
```

Common Format Control Arguments

- ~A prints any lisp object (strings without quotes)
- ~S prints any lisp object (strings with quotes)
- ~D prints a decimal integer
- ~F prints a float
- ~% prints a newline
- ~<return> ignores the <return> and any following spaces

(format *standard-output* “~A ~5F ~A ~%” 5 pi 10)

5 3.142 10

Allegro CL Certification Program

Lisp Programming Series Level I

Conditionals

Conditionals

- If
- when
- unless
- cond
- case
- ecase

IF

- (If test-form then-form else-form)

```
(if (eql saved-symbol password)
    (print “pass”)
    (print “fail”))
```

- If the test returns non-NIL, executes the THEN part and returns its value
- Else executes the ELSE part and returns its value
- ELSE part is optional

Using if

```
(defun  sign-name  (number)
          (if  (>   number 0)
                 'positive
                 'not-positive))
```

- Boolean test returns NIL (false) or true
- (If <test> <then> <else>)

Using if, cont'd

```
(defun  sign-name  (number)
 (if  (>  number  0)
     “positive”
     (if  (=  number  0)
         "zero"
         "negative")))
(sign-name 10) -> “positive”
(sign-name -1) -> “negative”
```

Progn

- Compound statemement, equivalent to curly braces { } in Java, C, C++.
- Example

```
> (if (> 3 2)
      (progn (print 'a) (print 'b))
      (progn (print 'c) (print 'd)))
A   <<<< printed
B   <<<< printed
B   <<<< Return value
```

Prog1 and Progn

Example

```
> (if (> 3 2)
      (progn (print 'a) (print 'b))
      (progn (print 'c) (print 'd)))
A
B
B
> (if (> 3 2)
      (prog1 (print 'a) (print 'b))
      (prog1 (print 'c) (print 'd)))
A
B
A
```

WHEN

- (when test code)

```
(when (eql saved-symbol password)
 (open-the-vault)
 (record-vault-contents)
 (close the vault))
```

- Equivalent to (if test then)
- Except no ELSE allowed
- Multiple body forms permitted

UNLESS

- (Unless test code)

```
(unless (equal string password)
    (call-the-police))
```

- Equivalent to (when (not …) …)

Compound Tests

- NOT: (not (> x 3))
- AND: (and (> x 3) (< x 10))
- OR: (or (> x 3) (< x 0) (= y 7) (< (+ x y) 5))

Other Types of Tests

- Numeric comparisons: >, >=, <, <=, =
- Equality of objects: EQ, EQL, EQUAL
- Equality of strings: string=, string-equal
  - (string-equal "Radar" "RADAR")
- Type tests:
  - (typep x 'integer)

COND

- Think of COND as if/elseif/elseif/elseif/endif
- Each clause has a test followed by what to do if that test is true.

```
(cond ((= x 1)
        (print 'single))
       ((= x 2)
        (print 'twin)
        (print “You WIN”))
       ((= x 3)
        (print 'triplet))
       (t
        (print 'unknown)
        (print “Too Bad”)
        x))
```

COND cont'd

- Tests are evaluated in sequence until the evaluation of one of them returns true (ie not nil)
- The last test may be the symbol t

```
(cond ((= x 1) (print 'single))
      ((= x 2) (print 'twin)
               (print "You WIN"))
      ((= x 3) (print 'triplet))
      (t (print 'unknown) (print "Too Bad") x))
```

CASE

- Key-form is evaluated to produce a test-key
- match is established if the result of the evaluation is eql to a key of the clause
- first element of final clause may be t or otherwise, either of which assures a match

```
(case x
  ((1 5)(print ′odd)(print “less than 7”))
  (2 (print 'two)(print 'twin))
  ((3 6 9)(print “multiple of 3”))
  (otherwise (print ′ok)))
```

Falling out of Case

- If no case is true, CASE simply returns NIL without doing anything.

```
(case x
  (1 (print 'single))
  (2 (print 'twin))
  (3 (print 'triplet)))
```

Case Example 1

```
(defun accept-bid-1 ()
  (format t "How many dollars are you offering ?")
  (let* ((offer (read))
         (counter-offer (+ offer 5))
         (field-width
            (1+ (length (format nil “~d”counter-offer)))))
     (format t "Would you consider raising that to ~v,'$d ?"
               field-width
               counter-offer)
     (case (read)
        ((y yes t ok) counter-offer)
        (otherwise offer))))
```

Case Example 2

```
(defun accept-bid-2 ()
   (format t "How many dollars are you offering? ")
   (let* ((offer (read))
          (counter-offer (+ offer 5))
          (field-width
           (1+ (length (format nil "~d" counter-offer)))))
      (if
       (y-or-n-p "Would you consider raising that to ~v,'$d ?"
                 field-width
                 counter-offer)
      counter-offer
      offer)))
```

ECASE

- If no case is true, ECASE signals an error.

```
(ecase x
  (1 (print 'single))
  ((2 4)(print 'twin))
  (3 (print 'triplet)))
```

```
"Error, 7 fell through an ECASE form.  The valid
  cases were 1, 2, 4, and 3.
```

Typecase

```
(typecase some-number
  (integer (print `integer))
  (single-float (print `single-float))
  (double-float (print `double-float))
  (otherwise (print `dunno)))
```

Etypecase

- Equivalent to TYPECASE with the otherwise clause signalling an error

```
(etypecase number
  (integer (print `integer))
  (single-float (print `single-float))
  (double-float (print `double-float)))
```

Allegro CL Certification Program

Lisp Programming Series Level I

Iteration and Recursion

dolist

- To iterate over elements of a list:

```
(defvar *lunch* '(apples oranges pears))

(dolist (element *lunch*)
  (print element))

(dolist (element *lunch* `done)
  (print element))
```

dotimes

Used to iterate over a some number of consecutive integers

```
(dotimes (I 5)
  (print I))

(setq lunch (list  'apples 'oranges 'pears))

(dotimes (I (length lunch))
  (print (nth i lunch)))
```

dotimes with return value

```
>(dotimes (I 4)
      (format t  “<~D>” i))
<0><1><2><3>
nil

> (dotimes (i 4 2)
        (format t "<~D>" i))
<0><1><2><3>
2
```

do

- A very general iteration method.
- Example: iterate by two's

```
(do ((I 0 (+ I 2))
     (J 7 (+ J .5)))
    ((> (+ I J) 50) `done)
   (print I)
   (terpri))
```

Do Syntax

```
(do ((variable1 init1 step1)
     (variable2 init2 step2)
     …)
    (endtest result)
  Body)
```

- Both dotimes and dolist could be implemented using do

Loop Without Keywords

```
(let ((I 0))
  (loop
    (when (> I 10) (return))
    (setq I (+ I 1))
    (print I)))
```

- Loop iterates forever (unless you call RETURN)

Loop with Keywords

```
CG-USER(14): (loop for i from 1 to 7
                   collect (* i i))
(1 4 9 16 25 36 49)

CG-USER(15): (loop for j from 0 to 3
                                by .5
                    sum j)
10.5
```

Iteration with Loop

- Many many options
  - give lots of power
  - can be misused
- For example, can collect, sum, maximize and minimize all in one loop
- Won't cover the full range of loop keywords in this class

Looping using from/to

```
(defun mycount (start-num end-num)
  (loop
    for num from start-num to end-num
     do
      (print num)))

(mycount 1 4)
1
2
3
4
NIL
```

Iteration without loop

- You can write code using do, dotimes, and dolist to accomplish the programming tasks addressed by loop keyword capabilities
- For example, you can write code to collect, sum, maximize and minimize

Summing a List of Numbers

- You can accumulate and return a sum

```
(defun sum (list)
  (let ((result 0))
    (dolist (item list)
      (setq result (+ result item)))
    result))

(sum '(1 2 3 4))
⇒10
```

Finding the Maximum

- You can search for a maximum value

```
(defun maximum (list)
  (let ((result (first list)))
    (dolist (item (rest list))
      (when (> item result)
        (setq result item)))
    result))

(maximum '(1 2 3 4))
⇒ 4
```

Iteration using conditionals

- You can "do" the body only under certain conditions

```
(defun print-even-numbers (list)
  (dolist (item list)
    (if (evenp item) (print item))))

(print-even-numbers '(10 1 2 4 7 8))
```

Recursion

- What is recursion ?
  - A special kind of iteration
  - a procedure in which a function calls itself
- A recursive function
  - terminates if some condition is met
  - calls itself with different arguments if condition is not met

Recursion cont'd

```
(defun find-even (list)
    (let ((item (first list)))
       (if (and (numberp item)(evenp item))
           item
           (find-even (rest list)))))

(find-even '(5 7 8 9 11))
(trace find-even)
(find-even '(5 7 8 9 11))
```

Note problem: what if no evens?

Recursion con'd, trace output

```
CG-USER(14): (find-even  '(5 7 8 9 11))
 0[1]: (FIND-EVEN (5 7 8 9 11))
   1[1]: (FIND-EVEN (7 8 9 11))
     2[1]: (FIND-EVEN (8 9 11))
     2[1]: returned 8
   1[1]: returned 8
 0[1]: returned 8
8
```

Recursion cont'd 2

```
(defun find-even (list)
    (if list
     (let ((item (first list)))
       (if (and (numberp item)(evenp item))
           item
           (find-even (rest list))))))

(find-even '(5 7 8 9 11))

(find-even '(5 7 9 11))

(find-even nil)
```

Recursion Components

```
(defun find-even (list)
  (if list
     (let ((item (first list)))
       ;; First, see if you are done.
       (if
        (and (numberp item)(evenp item))
        item
          ;; If not, call the same
          ;; function with a different
          ;; argument list.
        (find-even (rest list)))))
```

Factorial

```
(defun factorial (N)
  ;; First, see if you are done.
  (if (< N 2)
      N
      ;; If not, call the same function
      ;; with a different argument list.
      (* N (factorial (- N 1)))))

(factorial 4)
(trace factorial)
(factorial 4)
```

factorial con'd, trace output

```
CG-USER(17): (factorial 4)
 0[1]: (FACTORIAL 4)
   1[1]: (FACTORIAL 3)
     2[1]: (FACTORIAL 2)
       3[1]: (FACTORIAL 1)
       3[1]: returned 1
     2[1]: returned 2
   1[1]: returned 6
 0[1]: returned 24
24
```

List Recursion

- Lists are recursive data structures
- Most algorithms on lists are recursive

```
(defun my-copylist (list)
  (if (or (not list) (not (listp list)))
      list
      (cons (my-copylist (first list))
            (my-copylist (rest list)))))

(my-copylist '(5 6 7 8))
```

List Recursion cont'd 1

```
(defun sum-em (somelist)
  (if (null (rest somelist))
      (first somelist)
    (+ (first somelist)
       (sum-em (rest somelist)))))

(defun sum-em2 (somelist)
  (let ((first-el (first somelist))
        (rest-of-em (rest somelist)))
    (if (null rest-of-em)
     first-el
      (+ first-el (sum-em2 rest-of-em)))))
```

List Recursion cont'd 2

```
(defun sum-em3 (somelist accumulator)
  (let ((rest-of-em (rest somelist)))
    (if
      (null rest-of-em)
      (+ accumulator (first somelist))
      (sum-em3 rest-of-em
            (+ accumulator
               (first somelist))))))
```

List Recursion cont'd 3

```
(defun sum-em4 (somelist)
  (let ((sum 0))
    (dolist (el somelist )
      (setf sum (+ sum el)))
    sum))

(defun sum-em5 (somelist)
  (let ((sum (first somelist)))
    (dolist (el (rest somelist) sum)
      (setf sum (+ sum el)))))
```

List Recursion cont'd 4

```
(defun sum-em6 (somelist)
  (let ((first-el (first somelist)))
    (if (null first-el)
     0
      (if (numberp first-el)
       (+ first-el
          (sum-em6 (rest somelist)))
     (+ (sum-em6 first-el)
         (sum-em6 (rest somelist)))))))

(sum-em6 '((1 2 3) 7 (4 5 6)))
```

Allegro CL Certification Program

Lisp Programming Series Level I

Nonlocal Exits

non-local exits

- a non-local exit is a return to the caller from the middle of some construct, rather than the end
- return, return-from, block
- catch, throw

Return-from is a lot like GOTO

- Return-from requires a block tag argument.

```
(defun try1 (item)
   (let ((result nil))
      (block search
         (dolist (object *objects*)
            (when (matchp item object)
               (setq result object)
               (return-from search nil))))
      (print result)))
```

- Block gives you a named place to go to.

Return-from cont'd

```
(defun try2 (item)
   (let ((result nil))
      (dolist (object *objects*)
         (when (matchp item object)
            ;;call to setq below is useless
            (setq result object)
            (return-from try2 nil)))
      ;;if called, the line below will print nil
      (print result)))
```

block and return-from

- block establishes a named context
- name is a symbol (might be the symbol NIL)
- the normal return is value of the last form of the block
- return-from allows early return, second arg is value to return

return

- Return from a block named nil
- do and other do<something> iterators create a block named nil around the code body

```
(defun try (item)
   (dolist (object *objects*)
      (when (matchp item object)
            ;; Return from the dolist:
            (return object))))
```

return cont'd

```
(defun try3 (item)
 (do* ((how-many (length *objects*))
       (index 0 (1+ index))
       (object (nth index *objects*)
               (nth index *objects*))
       (match-found nil))
      ((or (setf match-found (matchp item object))
           (>= index how-many))
      match-found)))
```

catch and throw

```
(defun alpha (arg1 arg2)
   (if (<= arg1 arg2)
       (throw 'spaghetti)))

(defun beta (recorded-average score handicap)
   (catch 'spaghetti
      (alpha (+ score handicap) recorded-average)
      'terrific))
(beta 100 90 20) -> TERRIFIC
(beta 100 70 20) -> nil
```

Catch and throw example

```
(defun catch-test (n)
  (catch 'location
    (prin1 "before thrower call")
    (terpri)
    (thrower n)
    (prin1 "after thrower call"))
    (terpri)
  (prin1 "after catch frame")
  t))
(defun thrower (n)
  (if (> n 5) (throw 'location)))
```

Catch and throw example 2

When THROWER throws to location, forms after the call to thrower in the catch frame are not executed

```
cg-user(42): (catch-test 10) ;; THROWER will throw
"before thrower call"

"after catch frame"
t
cg-user(43): (catch-test 0) ;; THROWER won't throw
"before thrower call"
"after thrower call"
"after catch frame"
t
cg-user(44):
```

catch and throw cont'd

- tag
  - is customarily a symbol
  - should not be a number
- establishes a catch block named with that object
- first argument of throw is the catch tag, second is the value to return.
- Throw doesn't need to be done in lexical scope of catch.

Packages

- A Lisp package
  - is a namespace for related functionality
  - establishes a mapping from names to symbol
- There is always a current package which is the value of the Common Lisp symbol *package*
- a symbol in the current package can be referenced by its name
- a symbol accessible in the current package can be referenced by its name

Packages cont'd 1

- a symbol accessible in the current package can be referenced by its name
- a symbol which is not accessible in the current package can be referenced by prefixing a package qualifier to its name
- the Common Lisp symbol *package*, which, like other symbols specified by the Common Lisp standard, is in the Common Lisp package and can always be referenced with common-lisp:*package* and cl:*package*

Packages cont'd 2

- packages have a kind of inheritance by which within any package the symbols of some other packages designated to be available externally can be referenced without a package qualifier
- if the current package is package a and package a "uses" package b, then the symbols of package b do not need package qualifiers
- most packages "use" the Common Lisp package

Packages cont'd 4

- Every Common Lisp implementation must provide the packages
  - COMMON-LISP: a package for ANSI Common Lisp symbols; you can't add to it or change it
  - COMMON-LISP-USER: a package for user's symbols
  - KEYWORD-PACKAGE for symbols that are used as markers

Packages cont'd 5

- The keyword package is for symbols used as markers
- a symbol in the KEYWORD package
  - is printed with a : (but nothing else) before the characters in the symbol's name

    :from-end

    :test
  - is self-evaluating

Packages cont'd 6

- The initial value of cl:*package* is
  - COMMON-LISP-USER except in the Allegro CL IDE
  - COMMON-GRAPHICS-USER when using the IDE
- The COMMON-LISP-USER package uses the COMMON-LISP package, as does COMMON-GRAPHICS-USER

Getting package information

```
CL-USER(1): *package*
#<The COMMON-LISP-USER package>
CL-USER(2): (package-nicknames *package*)
("CL-USER" "USER")
CL-USER(3): (find-package :user)
#<The COMMON-LISP-USER package>
CL-USER(4): (package-name
              (find-package :cl-user))
"COMMON-LISP-USER"
CL-USER(5): (package-use-list
              (find-package :cl-user))
(#<The COMMON-LISP package>
 #<The EXCL package>)
```

Getting package information 2

```
CG-USER(1): *package*
#<The COMMON-GRAPHICS-USER package>
CG-USER(2): (package-nicknames *package*)
("CG-USER")
CG-USER(3): (find-package :cg-user)
#<The COMMON-GRAPHICS-USER package>
CG-USER(4): (package-name (find-package :cg-user))
"COMMON-GRAPHICS-USER"
CG-USER(5): (package-use-list (find-package :cg-user))
(#<The COMMON-LISP package> #<The EXCL package>
#<The ACLWIN package>
 #<The COMMON-GRAPHICS package>)
```

Creating a package

```
 CG-USER(6): (defpackage :my-first-package)
#<The MY-FIRST-PACKAGE package>
CG-USER(7): (package-use-list
                        (find-package :my-first-package))
(#<The COMMON-LISP package>)
CG-USER(8): (in-package :my-first-package)
#<The MY-FIRST-PACKAGE package>
MY-FIRST-PACKAGE(9): (defun my-function (a b)
                  (* a b))
MY-FUNCTION
MY-FIRST-PACKAGE(10): (my-function 2 3)
6
```

Creating a package cont'd

```
MY-FIRST-PACKAGE(11): (describe 'my-function)

MY-FUNCTION is a SYMBOL.
  It is unbound.
  It is INTERNAL in the MY-FIRST-PACKAGE package.
  Its function binding is
     #<Interpreted Function MY-FUNCTION>
   The function takes arguments (A B)

MY-FIRST-PACKAGE(12): (in-package :cg-user)
#<The COMMON-GRAPHICS-USER package>

CG-USER(13): (my-first-package::my-function 3 5)
15
```

Packages cont'd 7

- Every symbol in a package
  - is either an internal symbol of that package or an external symbol of that package
  - can be referenced with :: between the package qualifier and the symbol name
- An external symbol
  - Is part of the package's public interface
  - Has been exported from that package.
  - Can be referenced with a single colon between the package qualifier and the symbol name

Packages cont'd 8

- It is advisable that every file of lisp code have exactly one call to in-package and that the call to in-package be at the top of the file, preceded only when needed by a call to defpackage
- Applications should have their own packages

Allegro CL Certification Program

Lisp Programming Series Level I

Basic Lisp Development in the IDE

Class Info

- One 2-hour presentation each week
- Lecture notes and homework available, online at http://www.franz.com/lab/
- One-on-one help via email at training@franz.com