

Allegro CL Certification Program

Lisp Programming Series Level I

Presented by



About David Margolies

- Manager, Documentation, Franz Inc
- Been working with Lisp since 1984
- dm@franz.com



About Franz Inc.

- Founded in 1984
- Lisp & Lisp Tools/Applications
- <http://www.franz.com>
- For general information: info@franz.com
- For technical support: support@franz.com





Format of the Course

- 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



Getting Allegro Common Lisp

- This class is based on Allegro CL 8.2.
- Trial version should be sufficient for this module
 - Download free from <http://www.franz.com/>
- I will be using Allegro CL on Windows
- You can use Allegro CL on UNIX, but the development environment is different, and I won't be showing how to use it.



Getting Documentation

- Allegro Common Lisp:
 - <http://www.franz.com/support/documentation/8.2>
- ANSI Common Lisp specification
 - <http://www.franz.com/support/documentation/8.2/ansicl/ansicl.htm>
- Can't remember the name
 - But it contains the word “bit”
 - Permuted index under ‘bit’



Documentation in the IDE

- Help: Help on Selected Symbol
- Help:ANSI Common Lisp
- Help:Allegro CL Documentation
- Help:Symbol Index
- Help: Tree of Knowledge



Allegro CL Certification Program

Lisp Programming Series Level I

Session 1.1.1

Overview of Common Lisp





History of Lisp

- Born in 1958
- John McCarthy
- 2nd oldest language still in active use
- Still on the leading edge



History of Lisp, cont'd

- Earliest widespread dialect called Lisp 1.5
- Bloomed into serious development environments but fragmented
 - Interlisp, MacLisp
- Standardization effort led to Common Lisp, CLtL1, CLtL2, CLOS
- Common Lisp has itself evolved, most recently into an ANSI standard



Lists

- A list is an ordered collection of objects in a particular format
- The print representation of a list, that is what a printed list looks like, is zero or more elements enclosed by parentheses
- (17 red “green” 2/3 blue) is a list
- lisp source code consists of lists
- lisp data may be collected in lists



Starting Allegro CL with the IDE

- Start the lisp from a submenu of the Allegro CL 8.2 item on the Start | Programs menu
- Enter lisp forms (function calls, numbers, ...) to the read-eval-print loop at a prompt in the Debug Window
- You can call any function which is already in the lisp (whether there at lisp startup or added since)



Count-digits-of-factorial


```
(defun count-digits-of-factorial (positive-integer-arg )
  (if (or (not (integerp positive-integer-arg))
        (< positive-integer-arg 0))
      (error "argument is not a non-negative integer")
      (let* ((fact-of-arg (factorial
                           positive-integer-arg))
              (string-of-factorial (write-to-string
                                     fact-of-arg))
              (count-list (compute-list-of-occurrences
                           string-of-factorial)))
        (format t
                  "~%In factorial of ~d, ~
                  the frequency of digits is as shown:"
                  positive-integer-arg)
        (print-data count-list))))
```



factorial

```
(defun factorial (n)
  (cond ( (or (not (integerp n))
              (< n 0))
        (error
         "argument is not a non-negative integer")))
  (t (fact n))))
```

```
;;; assumes a non-negative integer
;;; computes factorial of its argument
(defun fact (arg)
  (if (zerop arg)
      1
      (* arg (fact (1- arg))))) ;recurse
```



compute-list-of-occurrences

```
# |  
  ;;; compute the frequency of all decimal digits  
(defun compute-list-of-occurrences (string-of-digits)  
  (let ((list-of-occurrences (make-list 10)))  
    (dotimes (i 10 list-of-occurrences)  
      (setf (nth i list-of-occurrences)  
            (count (coerce (write-to-string i) 'character)  
                    string-of-digits)))))  
| #  
(defun compute-list-of-occurrences (string-of-digits)  
  (loop for i from 0 to 9  
        collect  
        (count (coerce (write-to-string i) 'character)  
                string-of-digits)))
```


print-data

```
;;; computes, generates and displays the table
(defun print-data ( list-of-counts )
  (format
    t
    "~% digit      frequency      percent of total")
  (let ((total (apply #' + list-of-counts))
        (field-width
          (length
            (write-to-string ;calculate field width
              (reduce #'max list-of-counts))))))
    ;;generate and display the table
    (dotimes (i 10)
      (let ((count (nth i list-of-counts)))
        (format t "~%      ~d          ~vd          ~5,2f%"
                  i field-width count
                  (* 100 (/ count total)))))))
```

Results

```
cg-user(12): (fact 100)
```

```
933262154439441526816992388562667004907159682643816214685929638952175999932299156089  
414639761565182862536979208272237582511852109168640000000000000000000000000000
```

```
cg-user(13): (count-digits-of-factorial 100)
```

In factorial of 100, the frequency of digits is as shown:

digit	frequency	percent of total
0	30	18.99%
1	15	9.49%
2	19	12.03%
3	10	6.33%
4	10	6.33%
5	14	8.86%
6	19	12.03%
7	7	4.43%
8	14	8.86%
9	20	12.66%

```
nil
```



Results 2

```
cg-user(14): (count-digits-of-factorial 1000)
```

In factorial of 1000, the frequency of digits is as shown:

digit	frequency	percent of total
0	472	18.38%
1	239	9.31%
2	248	9.66%
3	216	8.41%
4	229	8.92%
5	213	8.29%
6	231	9.00%
7	217	8.45%
8	257	10.01%
9	246	9.58%

nil



Adding code to a lisp session

- You can enter definitions directly at the prompt or, preferably, write your definitions in an editor (from which they can be saved) and load them into that or any other lisp session
- Entering a definition at the lisp prompt adds that definition to that lisp session



Loading a file into a lisp session

- To load a file into Common Lisp
 - `(load "myfile.cl")` in any Common Lisp
 - `:ld myfile.cl` in any Allegro CL
 - File | Load in the Allegro CL IDE



Loading a function into a lisp session

- To load a single definition from the IDE editor into Allegro CL on Windows
 - put the mouse cursor on the open parenthesis of the definition
 - enter the <enter> key of the numeric keypad



Compiling a lisp file

- The Common Lisp compiler is used to compile Common Lisp source files
 - The result of compiling a file of Common Lisp code is another file which
 - typically loads and runs faster than the source file from which it was made
 - isn't in a format which anyone can read



Compiling and loading a lisp file

- (compile-file "myfile.cl")
- :cf myfile.cl
- For the file to affect the lisp session in which it was compiled or any other lisp session, the compiled file has to be loaded into the session in which you want access to the definitions in the file.
 - (load "myfile.fasl")
 - :ld myfile.fasl



Compiling and loading a lisp file cont'd

- There are compile and load items in the File menu of the Allegro CL IDE



Compiling and loading a lisp file

- To both compile a source file and load the newly created compiled file:
 - `:cl myfile.cl`
 - File | Compile and Load from the Allegro CL IDE menu-bar
 - select the loaded truck icon of the Allegro CL IDE menu-bar



Dynamic Programming Language

- ‘Dynamic’ means you can add or redefine functions while the program is running
- Change a function, compile it, load it, and test it without restarting the application
- Very fast edit-debug cycle
- Frequent development strategy: use stand-in functions for something complicated that will be needed eventually
 - stand-in goes into debugger
 - stand-in returns dummy values



Lisp Issues

- Weird syntax: $(+ \ a \ b)$ instead of $a + b$ takes some getting used to but
 - $(+ \ a \ (/ \ b \ c))$ less ambiguous than $a + b / c$
 - Parsing lisp programs is trivially easy due to the prefix notation
- Garbage collection
 - If you notice it at all, something is probably wrong




Lisp Syntax

- Prefix notation
 - Function name followed by zero or more args
- Delimited by parentheses
- (+ 2 3 4)
- (* (- 7 1) (- 4 2) 2)
- (print “Hello, world”)



Examples of Lisp Syntax

- `(solve-polynomial 1 5 7 9)`
- `(if (eq weather 'cold)
 (turn-on-heat)
 (turn-on-ac))`
- `(dotimes (i 5)
 (print i))`



Lists are used to define and call functions

- To define a function use the operator `defun`
- Specify function name, list of argument names, then body of function

```
(defun  my-square (x)
  ( *   x   x) )
```

```
(my-square 7) ==> 49
```



Lisp includes a large and extensible number
of types of objects



Lisp Data types

- Data Types found in Other Languages
 - Numbers: 123, 1.45, -10e2, 3/5, #C(2 3)
 - Strings: "This is a fine day"
- Data types more specific to Lisp
 - Symbols: HELLO, FOO, START
 - Lists: (a (b c) d)
 - Characters: #\A, #\z, #\space



Program Data is Freed Automatically

The job of the garbage collector is

- to notice objects that are not referenced anywhere
- to make the space those objects occupy available for reuse



Evaluation

- When a lisp program is “run”, the call to run the program is evaluated. That is, it is processed by a Common Lisp function called eval.



Evaluation cont'd

- There are three categories of objects which can be evaluated.
 - Symbol
 - a list of which the first element is the name of an operator (a compound form)
 - a self-evaluating object



Self-Evaluating Objects

- Numbers
- Strings
- ...



Evaluation of a list

If the expression is a list of which the first element names a function, then:

- Each of the elements in the list after the function name is evaluated in the order in which they appear in the list
- The result of each such evaluation is passed to the function as an argument.



Evaluation of a list cont'd

- $(+ \ 2 \ 3 \ 4)$
 - the symbol $+$ names a function
 - $(+ \ 2 \ 3 \ 4)$ is a function call
 - 2, 3 and 4 are passed as arguments to the function
- Evaluating a form always returns a value



Evaluation of a function call

- If the first element of a list to be evaluated is a symbol which names a function, each of the list elements after the function name is evaluated to provide an argument for the function named
- $(* (- 7 1) (- 4 2) 2)$
 - returns 24

Try Out the Evaluator


USER (2) : (+ 2 3 4)

9

USER (3) : (* (- 7 1) (- 4 2) 2)

24

- Type expression to the Lisp prompt and hit Enter
- Lisp reads what you typed, evaluates it, and prints the return value



A Symbol is a Lisp Object

- That may name a variable, in which case it has an associated value.
- That may (or may also) name a function
- has some other information associated with it including a property list



print representation of a symbol

- consists of a number of consecutively displayed characters, usually alpha-numeric or hyphens
- in an ANSI standard Common Lisp, the alphabetic characters are usually upper-case



ANSI Lisp is Case-Insensitive

- When entered from the keyboard or read from a file, the following are equivalent and are all read as references to the symbol RED
 - red
 - Red
 - RED
- By convention, most lisp code is lower case
- Allegro Common Lisp lets you create a case-sensitive lisp



Evaluating a Symbol

When a symbol is evaluated the binding (or value) of the variable named by the symbol is returned.

(+ number-of-apples number-of-bananas)

(+ pi pi)

(+ pi internal-time-units-per-second)



t and nil

- There are distinguished symbols t and nil
- Variables named by the symbols t and nil ALWAYS evaluate to themselves
 - t evaluates to t
 - nil evaluates to nil
- You cannot set the value of t or nil to any other value so (setq t 100) signals an error



Quote

When the first character of anything to be evaluated is `'`, the value returned is the object which followed the `'`.

- `a` is not the same as `'a`
- `(a b c)` is not the same as `'(a b c)`



Using Symbols

- (if (eq today 'Monday) ...)
- (list 'July (+ 3 1) this-year)
- (find 'meat ingredients)



symbol and variable creation

- a symbol is created when the lisp reader reads an appropriate sequence of characters that don't correspond to the name of some symbol already in the lisp or when a compiled file that references a new symbol is loaded
- a variable is created with
 - defvar, defparameter, defconstant
 - let, let* or some other means of creating local bindings



special variable creation

- `defvar variable-name [initial-value]`
- `defparameter variable-name initial-value`

```
(defvar *alpha*)
```

```
(defvar *beta* (* 2 3))
```

```
(defparameter *gamma*  
  (* 2 7))
```



special variable creation 2

CL-USER(4): *alpha*

Error: Attempt to take the value of the
unbound variable `*ALPHA*`.

[condition type: UNBOUND-VARIABLE]

[1] CL-USER(5): :pop

CL-USER(6): *beta*

6

CL-USER(7): *gamma*

14



special variable creation 3

```
CL-USER(8): (defvar *alpha* 3)  
*ALPHA*
```

```
CL-USER(9): (defvar *beta* 5)  
*BETA*
```

```
CL-USER(10): (defparameter *gamma* 7)  
*GAMMA*
```

```
CL-USER(11): *alpha*  
3
```

```
CL-USER(12): *beta*  
6
```

```
CL-USER(13): *gamma*  
7
```

Local variable creation

most local variables are created and bound with
let or let*

```
(let* ((x 0)
      (y (* 2 3))
      z)
  (print z)
  (format t "~% (~a, ~a)" x y))
```

NIL

(0, 6)

NIL



let

- Mostly equivalent to let*.
- Local variables defined by the same call to let can't depend on each other in the initialization clauses.

```
(let* ((x 0)
      (y (+ x 5)))
  (print y))
```



Binding

- A variable is bound when it is initialized
- The binding of a variable can be changed with `setq` or `setf`
- If two variables have the same name, the inner binding shadows the outer one.

```
(let ((a 1))  
  (+ a (let ((a 10))  
        (+ a 1))))
```

⇒12



Two Kinds of Variables

- Lexical or “static”
 - A local variable
 - References can be made only within the program construct where it is created or bound
- Special or “dynamic”
 - A global variable
 - References can be made at any time, anywhere



Local variables have lexical scope

```
(defun gamma (v)
  (let ((w 7))
    (print v)
    (print w)
    (delta (+ v w)))))
```

```
(defun delta (delta-arg)
  (print delta-arg)
  (print w) ; this won't work
  (print v) ; this won't work either
  )
```



Why Use a Special Variable?

- Global
- Dynamic extent of bound special variables

Special variables have dynamic extent

```
(defvar *v* 3)
(defvar *w* 19)

(defun gamma (*v*) ;will shadow the outer value
  (let ((*w* 7)) ;will shadow the outer value
    (print *v*)
    (print *w*)
    (delta (+ *v* *w*)))))

(defun delta (delta-arg)
  (print delta-arg)
  (print *w*)
  (print *v*))
```



Example of let binding a special

```
(defvar *default-file*) ; no value
(defvar *default-directory "c:/")

(defun process-file (file directory)
  (let ((*default-file* file)
        (*default-directory* directory))
    (print *default-file*)
    (print *default-directory*)
    (loader)))

(defun loader ()
  ;; This will signal an error unless called
  ;; by process-file
  (load (merge-pathnames *default-file*
                          *default-directory*)))
```



macros and special operators

- some Common Lisp operators are not functions
- when the name of a macro or special operator appears as the first element in a list to be evaluated, the rules for how the other elements of the list are dealt with depend on the particular macro or special operator



setting the value of a variable

- The special operator `setq` and the macro `setf` are used to set the value of a variable named by a symbol
- `(setq x 37.0)`
- `(setf z (+ 2 3))`
- `(setq eye-color 'blue)`



Variable Assignment Using setq and setf

- The second element of the list (variable name) is NOT evaluated
- The third element of the list is evaluated
- (setq x 35.0)
- The “variable” being “set” is named by the symbol x



Program Data is Typed, Variables are NOT typed

- `(setq x 35)` ; value of x is integer
- `(setq x 35.0)` ; value of x is float
- There are no type restrictions on the value of a variable
- The value of a variable may be any lisp object
- Every lisp object has a type



ANSI Lisp is Case-Insensitive cont'd

- These lines are all equivalent:
 - (setq color 'red) ; downcase
 - (Setq Color 'Red) ; capitalized
 - (setq color 'RED) ; uppercase



Function Definitions

- Use defun
- Specify function name, list of argument names, then body of function

```
(defun my-square (x)  
  (* x x) )
```

```
(my-square 7) ==> 49
```



Function definition

```
(defun board-dimensions ;name  
      (length width)      ;lambda-list  
      (* length width))   ;body forms  
  
(board-dimensions 12 14)
```



Function definition optional arguments

```
(defun board-dimensions  
  (length width &optional (extra 0 ))  
  (* (+ length extra)  
     (+ width extra)))
```

```
(board-dimensions 10 12 1)
```

```
(board-dimensions 10 12)
```



Function definition keyword arguments

```
(defun board-dimensions  
  (length width  
    &key (extra-width 0) (extra-length 0))  
    (* (+ length extra-length)  
      (+ width extra-width)))
```

```
(board-dimensions 8 12 :extra-length 1)
```



Function definition rest arguments

```
(defun board-dimensions  
  (length width  
    &rest who-is-to-do-the-work)  
  (print “The list of those to do the work follows:”)  
  (print who-is-to-do-the-work)  
  (* length width))
```

```
(board-dimensions 8 12 ‘donald ‘louie ‘dewey )
```



Specifying &optional and &key arguments

Each &optional and &key argument may be specified in any of 3 ways

- symbol, in which case the default value is nil
- list of symbol and default value
- list of symbol, default value and another symbol, which in any call to the function will have the value t or nil depending on whether the function call included a value for this &optional or &key argument



3 element list specification of &optional or &key argument

```
(defun do-it (previous-balance additional-funds
              &optional (report nil report-p))
  (let ((new-balance
        (+ previous-balance additional-funds)))
    (when report-p
      (format t
               "~%Accounting has requested that we ~
               ~:[ do not issue ~; issue~] a report"
               report)
      (if report
          (format t
                   "~%current balance is ~,2f"
                   new-balance)))
    new-balance))
```





Function definition lambda-list keywords

- Must follow all required arguments
- &optional must precede &key or &rest
- Inadvisable to use both &optional and &key in the same lambda-list
- Using &key and &rest in the same lambda-list requires better understanding



Data Equality

- internally Lisp refers to most objects via pointers
- fundamental equality operation is EQ
 - only true if the two arguments point to the same object
 - test is very efficient
- EQL, which is slightly less restrictive than eq, is the default test for equality in most functions which do comparison of objects



arithmetic functions take all appropriate numeric arguments

CG-USER(34): (+ 60 5.0 1/2)
65.5

CG-USER(35): (- 60 5.0 1/2)
54.5

CG-USER(36): (* 60 5.0 1/2)
150.0

CG-USER(37): (/ 60 5.0 1/2)
24.0



Creating Lists

- `(list 'this 'is 'a 'list) --> (THIS IS A LIST)`
- `'(so is this) --> (SO IS THIS)`
- `(+ 2 3)`
- `(defun square-it (it) (* it it))`



printed output

- `print <object to print>`
- `println <object to print>`
- `princ <object to print>`
- `format`
 - `t`, `nil` or where to print
 - format string
 - format arguments



printed output cont'd

CG-USER(1): (print "abCd")

"abCd"

"abCd"

CG-USER(2): (prin1 "abCd")

"abCd"

"abCd"

CG-USER(4): (princ "abCd")

abCd

"abCd"



printed output cont'd 2

CG-USER(22): (format t

 "~%~s and ~a paid \$~5,2f for ~d pupp~:@p"
"John" "Mary" 25.99 1)

"John" and Mary paid \$25.99 for 1 puppy

NIL

CG-USER(23): (format t

 "~%~s and ~a paid \$~5,2f for ~d pupp~:@p"
"John" "Mary" 25.99 2)

"John" and Mary paid \$25.99 for 2 puppies

NIL



Input

- (read *standard-input*)
 - Reads one Lisp object from *standard-input*
 - Examples of “one Lisp object”:
 - String: “hello, world”
 - Symbol: RED
 - List: (one two three)
 - Number: 3.1415
 - argument shown in the call above is optional



Output

- `(print 'hello *standard-output*)`
 - Prints HELLO to *standard-out*
 - Returns the symbol HELLO as its value
 - Stream argument shown above is optional

USER (2) : `(print 'hello)`

HELLO

HELLO

USER (3) :



Summary

- Lisp is a very rich language
 - there are very many predefined useful functions
 - can't remember all details of all, but don't need to
- Extremely easy to add your own functions

Allegro CL Certification Program

Lisp Programming Series Level I

Session 1.1.2

Overview of Allegro Common Lisp





Delivering Applications

- Standard Allegro CL license allows you to develop and run the application on the licensed machine(s)
- There is no "executable" per se as with C++ or Visual Basic
- The "application" is a collection of files including the proprietary Allegro lisp engine
- Delivered applications must include the core lisp engine (interpreter, garbage collector, core language functions, etc.) but may have to exclude the development environment and possibly the compiler



Downloading Patches

- Franz regularly releases patches to their various products.
- There is a function that downloads
(sys:update-allegro)
- Follow the directions displayed when (sys:update-allegro) completes.
- Install | New Patches in IDE is dialog-based way to call sys:update-allegro



What's On My Bookshelf

- Common Lisp, The Language, 2nd Edition. Guy L. Steele Jr. (1990, Digital Equipment Corporation).
- ANSI Common Lisp (Paul Graham 1996, Prentice Hall). Excellent instructive text for basic language features.
- On Lisp, Advanced Techniques for Common Lisp (Paul Graham 1994, Prentice Hall). Excellent instructive text for advanced techniques.



More On My Bookshelf

- Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. Peter Norvig (1992, Morgan Kaufman). Excellent book on AI with well written lisp code.
- Object-Oriented Programming in Common Lisp, A Programmer's Guide to CLOS. Sonya E. Keene (1989, Symbolics).
- The Art of the Metaobject Protocol. Kiczales et. al. (1991, MIT). Advanced book on concepts of object-oriented programming in Common Lisp. Excellent, but don't bother until you have a year or two of experience.



More On My Bookshelf

- Practical Common Lisp, Peter Seibel (Apress 2005)
- Basic Lisp Techniques. David J. Cooper, Jr.



Class Information

- One 2-hour presentation each week: next class 10:00 AM PDT (California Time) next Wednesday
- Lecture notes and homework available, online at <http://www.franz.com/lab/>
- One-on-one help via email at training@franz.com