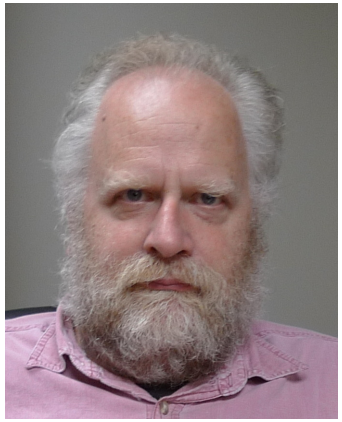
Allegro CL Certification Program

Lisp Programming Series Level I

Session 1.3.1

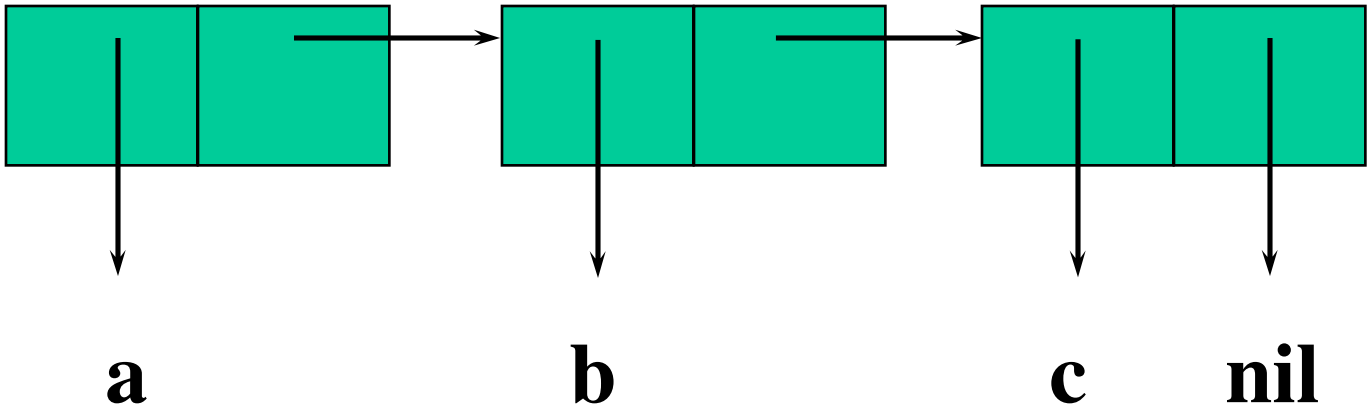
Manipulating Lists

David
Margolies



What is a List?

- An ordered (possibly empty) collection of things in a particular format
- (a b c)



CONS

- an object with two components
  - first (car)
  - rest (cdr)
- function CONS allocates one

```
(cons `a 7) -> (a . 7)

(cons 'a NIL) -> (a)

(cons 'a (cons 'b (cons 'c NIL))) -> (a b c)
```

Definition of a list

- symbol NIL, which represents the empty list
- cons whose cdr is a list

(cons ‘a (cons ‘b nil)) -> (a b)

The Empty List

- NIL
- ()
- NIL evaluates to itself

```
(car nil) => nil
(cdr nil) => nil
```

Making a List

```
(list 'boston 'distance (* speed time))
⇒ (BOSTON DISTANCE 1800)

;; When the elements are all the same:
(make-list 3 :initial-element 'denver)
⇒ (DENVER DENVER DENVER)
```

Uses for lists

- Lists can be simple or have structure
- association list
  - ((a 1) (b 2) (c 3))
  - ((a . 1) (b . 2) (c . 3))
  - ((a 1 mary) (b 2 donna)…(w 7 henry))
  - associates a key with a value
- property list
  - (:a 1 :b 2 :c 3)
  - another way of associating a key with a value

Overview of Lists

- Lists are an extremely useful data structure
- Size is unbounded (except by memory constraints)
- In other languages you must write your own list primitives
- Since lists are so fundamental to Lisp there are many built functions to manipulate lists

Simple List Operations

- (first '(a b c)) --> A
- (first nil) --> nil
- (rest '(a b c)) --> (B C)
- (rest nil) --> nil
- (second '(a b c)) --> B

  equivalent to (first (rest '(a b c)))
- (third '(a b c)) --> C
- (second nil) --> nil
- (third nil) --> nil

Simple List Operations cont'd

- (nth 0 '(a b c)) => a
- (nth 2 '(a b c)) => c
- (nth 5 '(a b c)) => nil

- (length '(power gnd clk1 clk2)) --> 4
- (length NIL) --> 0

Member of a list

- (member 'dallas '(boston san-francisco portland))

--> NIL

- (member 'boston '(boston san-francisco portland))

--> (BOSTON SAN-FRANCISCO PORTLAND)

- (member 'portland
  ‘(boston san-francisco portland)) -->
  (PORTLAND)
- Anything other than NIL is TRUE

Set Operations

- (union '(1 2 3) '(2 3 4)) --> (1 2 3 4)
- (intersection '(1 2 3) '(2 3 4)) --> (2 3)
- (set-difference '(1 2 3 4) '(2 3)) --> (4 1)

car/cdr combinations

- old names can be combined, e.g.

```
(setq foo '(1 2 3))
(car (cdr foo))
```

⇒ **2**

- can be written as

```
(cadr foo)
```

- occasionally see combinations up to 4 a's and d's

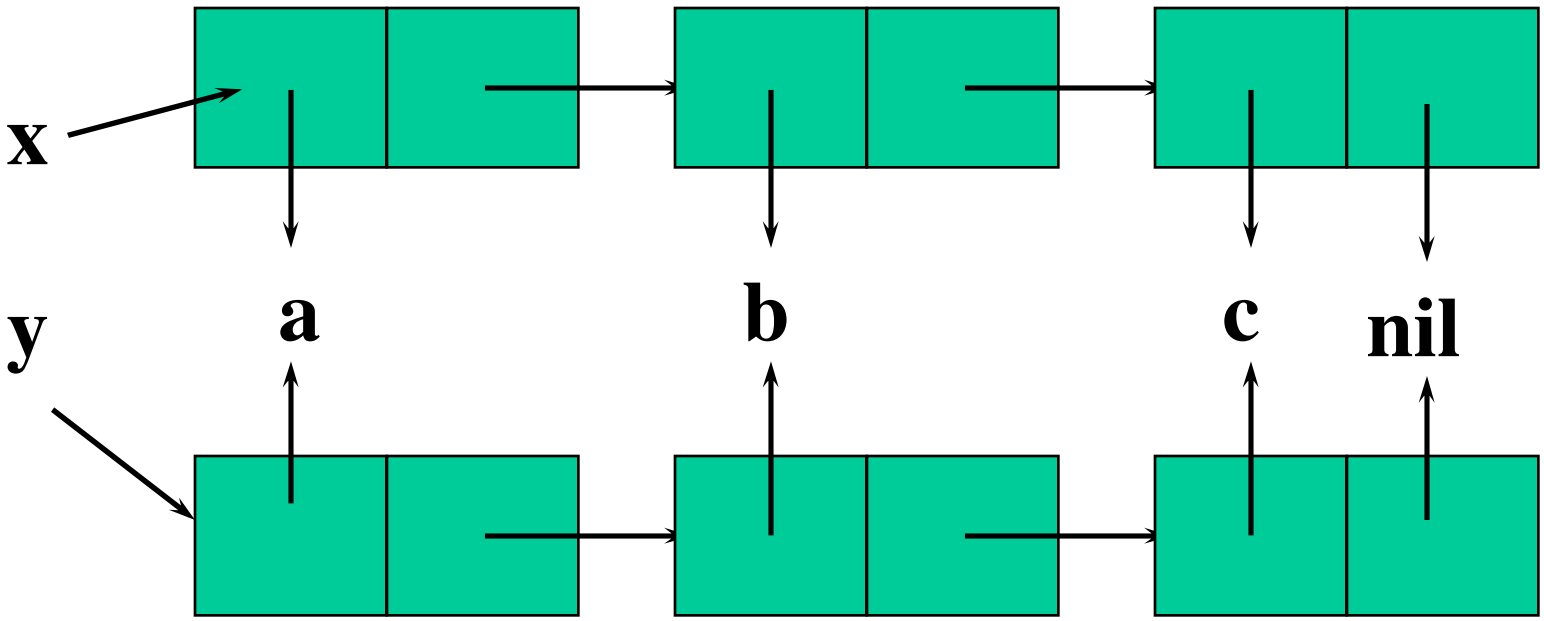
copy-list

- Copy a list

```
> (setq a '(1 2 3 4))
(1 2 3 4)
> (setq b (copy-list a))
(1 2 3 4)
> (eq a b)
NIL
> (equal a b)
T
```

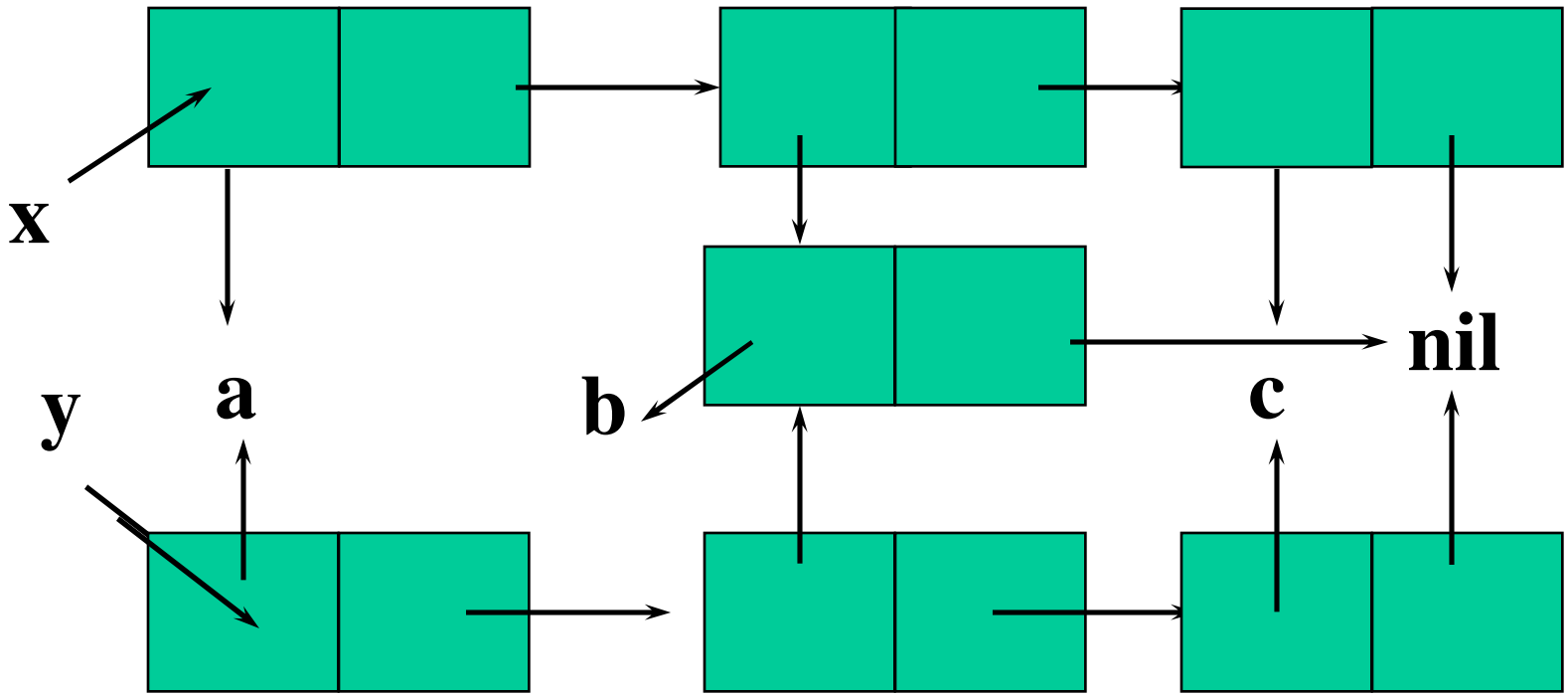
What about copy-list

- (setf x '(a b c))
- (setf y (copy-list x))



What about copy-list cont'd

- (setf x '(a (b) c))
- (setf y (copy-list x))



last

- Returns the last CONS of a list. Result is NOT the last element.
- Optional count arg, returns nth from last

```
> (setq a '(1 2 3 4))
(1 2 3 4)
> (last a)
(4)
> (last a 2)
(3 4)
> (car (last a 2))
3
```

append

- Appends lists
- Copies all args except last

```
> (setq a '(1 2 3 4))
(1 2 3 4)
> (setq b '(5 6))
(5 6)
> (setq c '(7))
(7)
> (setq d (append a b c))
(1 2 3 4 5 6 7)
```

nconc

- Destructive append

```
> (setq a '(1 2 3))
(1 2 3)
> (setq b '(4 5 6))
(4 5 6)
> (setq c (nconc a b))
(1 2 3 4 5 6)
> c
(1 2 3 4 5 6)
> b
(4 5 6)
> a
(1 2 3 4 5 6)
```

rplaca

- rplaca x y - Changes first of x to y
- (setf (first x) y) is equivalent

```
> (setq a  (list 1 2 3))
(1 2 3)
> (rplaca a 4)
(4 2 3)
> a
(4 2 3)
```

rplacd

- Rplacd x y - Replaces the rest of x with y
- (setf (rest x) y) is equivalent

```
> (setq a '(1 2 3))
(1 2 3)
> (rplacd a '(8 9))
(1 8 9)
> a
(1 8 9)
```

Pushing and Popping

```
(setq *my-network* '(power gnd input))
    (POWER GND INPUT)
(push 'clk1 *my-network*)
    (CLK1 POWER GND INPUT)
*my-network*
    (CLK1 POWER GND INPUT)
(pop *my-network*)
    CLK1
*my-network*
    (POWER GND INPUT)
```

SETF

- setq only takes a symbol as first argument
- setf takes a form that if executed would access a value
  - called a generalized reference
  - accessor if called by itself would retrieve a value
  - setf of a generalized reference stores a replacement value
- (setq a (list 'red 'green 'blue))
- (setf (second a) ‘magenta)

push

- Push item onto place
  - place is a generalized reference to a list

```
> (setq a (list 1 (list 2 3)))
(1 (2 3))
> (push ‘x a )
(x 1 (2 3))
> a
(x 1 (2 3))
> (push  9 (third a))
(9 2 3)
> a
(x 1 (9 2 3))
```

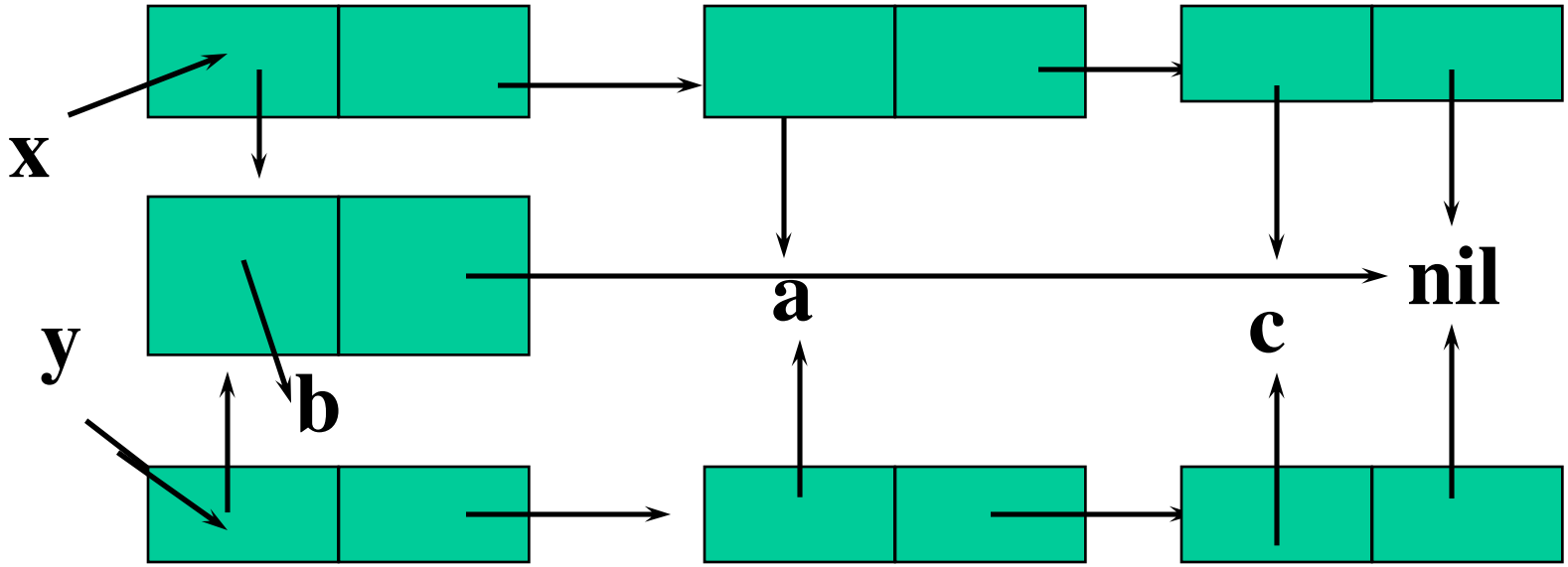
pushnew

- Push if it isn't there already

```
> (setq a (list 1 (2 3) (4 5)))
(1 (2 3) (4 5))
> (pushnew 9 (second a))
(9 2 3)
> a
(1 (9 2 3) (4 5))
> (pushnew 9 (second a))
(9 2 3)
```

More on copies

- (setf x '((b) a c))
- (setf y (copy-list x))

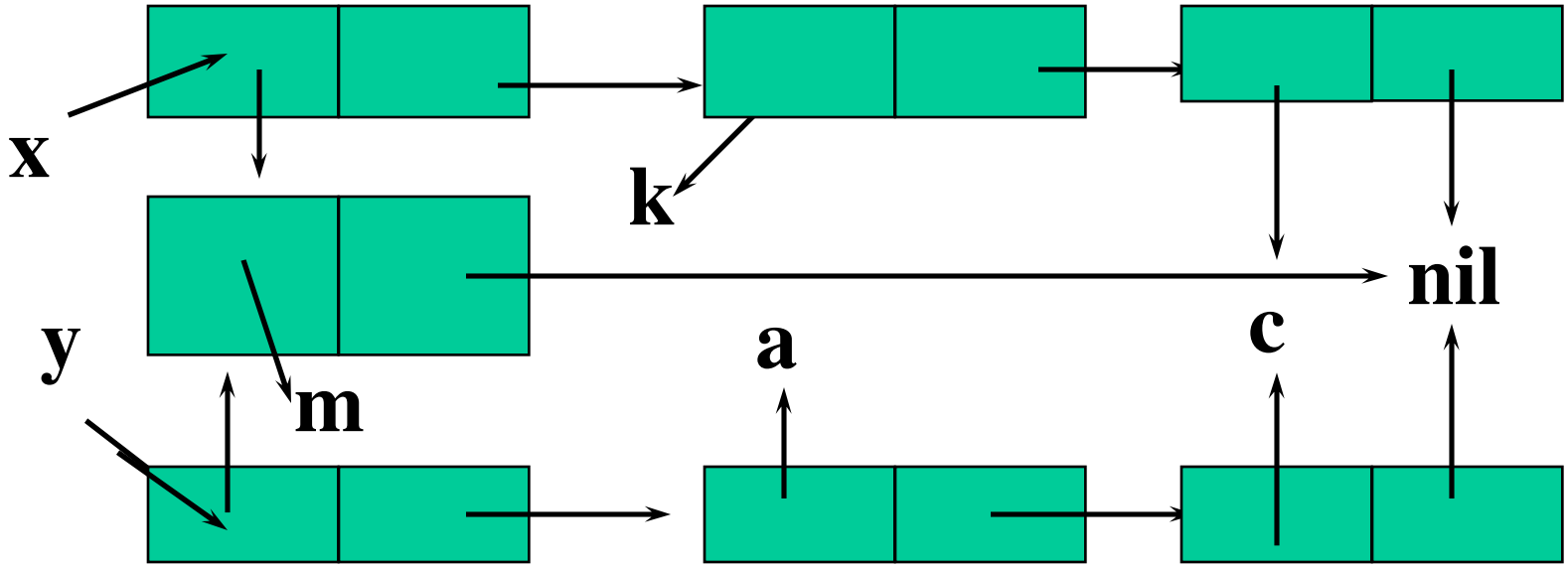


We set some values

```
(setq x '((b) a c))
(setq y (copy-list x))
x => ((b) a c)
y => ((b) a c)
(setf (second x) 'k) => k
x => ((b) k c)
y => ((b) a c)
(setf (car (first y)) 'm) => m
x => ((m) k c)
y => ((m) a c)
```

More on copies

- x => ((m) k c)
- y => ((m) a c)



Printing lists

- *print-level*, *print-length* control printing depth and length (number or nil for no restriction)
- In ACL, tpl:*print-level*, tpl:*print-length* control top-level printing
- PPRINT always prints the whole list
- Do not set top-level variables to nil

List printing example

```
(setq lis '(a (b (c (d (e))))) 2 3 4 5 6 7 8 9 10 11))
((a (b (c (d #)))) 2 3 4 5 6 7 8 9 10 ...)
cg-user(25): (print lis)

((a (b (c (d (e))))) 2 3 4 5 6 7 8 9 10 11)
((a (b (c (d #)))) 2 3 4 5 6 7 8 9 10 ...)
cg-user(26): (pprint lis)

((a (b (c (d (e))))) 2 3 4 5 6 7 8 9 10 11)
cg-user(27):
```

Allegro CL Certification Program

Lisp Programming Series Level I

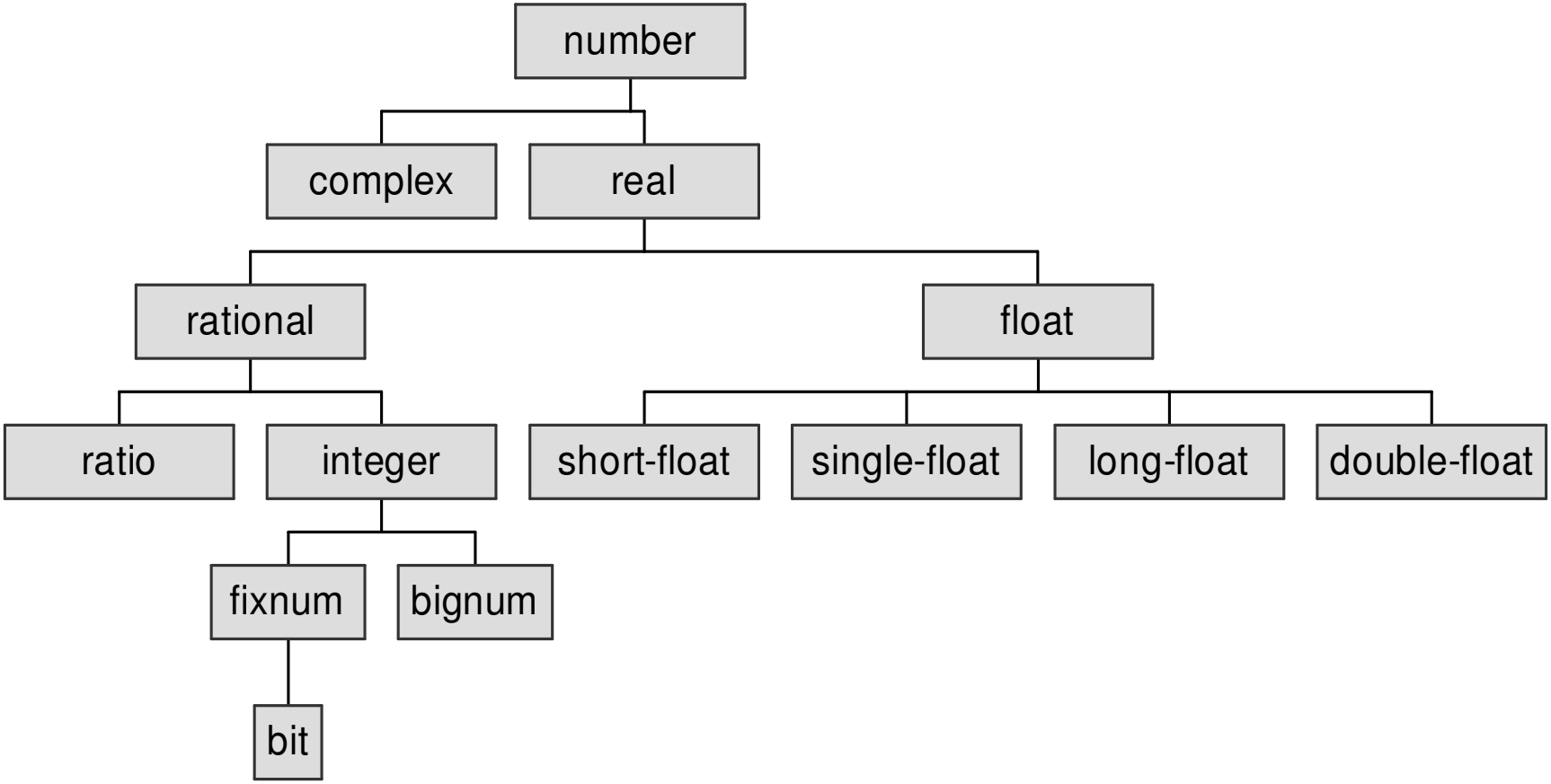
Session 1.3.2

Numbers

Many types of numbers

- Integers
  - fixnum, 123
  - bignum, 123123123123123123123123123123
  - automatically change from one to other as needed
- Ratios, 7/3
- Float, 2.333333333333, 8.4e2
- Complex, #c(0 1), #c(7.3 4.5)

Type Hierarchy



fixnum

- most-positive-fixnum
- most-negative-fixnum
- efficiently maps into underlying hardware's integer operations

Type examples

- (typep (+ most-positive-fixnum 1) ‘bignum) --> T
- (typep 1 ‘fixnum) --> T

Arithmetic operations are generic

- Operate on all appropriate numeric types
- Coerce to appropriate types
  - rational canonicalization
  - floating point contagion
  - complex contagion
  - complex canonicalization

floats

- 1.234e7 and 12340000.0 are read in the float format which is the value of

read-default-float-format

- s, f, d, or l in place of e gets specific type
- Limit variables
  - most- or least-
  - -negative- or -positive-
  - float type (short-float, single-float, double-float, long-float)

“Conversion” functions

- Float - maps a real number to a floating point number
- truncate, round, floor, ceiling - map to integers

float

(float 1) --> 1.0

(float 2/3) --> 0.6666667

(float 2/3 1.0d0) --> 0.6666666666666666d0

(float 0.666667 1.0d0) -->
0.6666666853488l6d0

Floor, round, ...

- floor -- truncates toward negative infinity
- ceiling -- truncates toward positive infinity
- truncate -- truncates toward 0
- round -- nearest integer, if halfway to even

Examples

- (floor 2.5) --> 2 (floor -2.5) --> -3
- (ceiling 2.5) --> 3 (ceiling -2.5) --> -2
- (truncate 2.5) --> 2 (truncate -2.5) --> -2

- (round 2.49) --> 2 (round 2.51) --> 3
- (round 2.5) --> 2 (round -2.5) --> -2
- (round 1.5) --> 2 (round -1.5) --> -2

floor and related functions

- Take a second, optional, argument which is a number by which to divide the first argument; defaults to 1
- return two values
  - integer part
  - remainder

Examples

```
> (floor 5 2)
2
1
> (floor 5.0 2)
2
1.0
> (floor 5 2.0)
2
1.0
> (floor 7/5)
1
2/5
```

Ratios

- no loss of precision

> (numerator 7/3)
7
>denominator 7/3)
3
> (/ 5 2)
5/2
> (/ 5 2.0)
2.5

Complex numbers

(complex 0 1) --> #c(0 1)

(complex 1 0) --> 1

(complex 3 2.5) --> #c(3.0 2.5)

(realpart #c(0 1)) --> 0

(imagpart #c(0 1)) --> 1

- The components of the complex number can be any type of real number

Math Fns

- (expt x n) --> x^n
- (log x n)
- (log x)   ;log base e
- (exp x) --> e^x
- (sqrt x)
- (isqrt x)

Trig Fns

- (sin theta-in-radians)
- (cos theta-in-radians)
- (tan theta-in-radians)

incf and decf

- Example

```
> (setq a 5)
5
> (incf a) ; equivalent to (setf a (+ a 1))
6
> (incf a 4) ; equivalent to (setf a (+ a 4))
10
> (decf a) ; equivalent to (setf a (- a 1))
9
```

Digression on macroexpansion

- Incf and decf are macros (as are push and pop)
- macroexpand transforms a macro call into underlying code

```
cg-user(29): (setq a 1)
1
cg-user(30): (macroexpand '(incf a))
(setq a (+ a 1))
t
cg-user(31):
```

Allegro CL Certification Program

Lisp Programming Series Level I

Session 1.3.3

Data Structures

Type functions

- (type-of <object>) --> most specific type
- (typep <object> <type>) --> T or nil

Some data structure types

- lists (chains of CONS cells)
- arrays
- hash-tables
- CLOS classes
- structures

Arrays

- Implementation must allow
  - at least 8 dimensions - array-rank-limit
  - at least 1024 elements per dimension
    - array-dimension-limit
- can either store general objects (of type T) or more specialized objects
  - arrays of characters
  - arrays of single-floats or of double-floats

Array examples

```
> (setf arr (make-array '(2 3) :initial-element nil))
#2A((NIL NIL NIL) (NIL NIL NIL))
> (aref arr 0 0)
NIL
> (setf (aref arr 0 0) 'b)
B
> (aref arr 0 0)
B
> arr
#2A((B NIL NIL) (NIL NIL NIL))
```

Vectors

- Vectors are 1-dimensional arrays

```
> (setf vec (make-array 4 :initial-element 0))
#(0 0 0 0)
> vec
#(0 0 0 0)
> (setf vec (vector "a" 'b 5))
#("a" B 5)
> vec
#("a" B 5)
> (aref vec 2)
5
> (svref vec 2)
5
```

Strings

- Strings are vectors of characters
- access with char function

```
> (setf string1 "abc”)
"abc"
> (aref string1 1)
#\b
> (char string1 1)
#\b
> (char string1 0)
#\a
> (setf (char string1 2) #\Z)
#\Z
> string1
"abZ"
```

String Functions 1

- string-equal
- string-lessp
- string-greaterp

```
> (string= "foo" "Foo")
NIL
> (string-equal "foo" "Foo")
T
```

String Functions 2

```
(string-trim '(#\space) " five ten   ")
⇒"five ten“
(string-trim '(#\space #\$)
             " $5.73, $8.40   ")
"5.73, $8.40"
⇒(string-upcase “five”)
⇒“FIVE”
(string-capitalize “five”)
⇒“Five”
(string `five)
⇒“FIVE”
```

String Functions 3

```
;; Alphabetizing strings:
(string< "apples" "bananas")
⇒ 0
;; Also string>, string<=, string>=, string/=
(string< "this" "that")
⇒ nil
(string> "this" "that")
⇒ 2
(string< "abc" "abcd")
⇒ 3
```

Sequence Functions

- What are sequences?
  - Lists
  - Vectors (and therefore strings)
- Some of the list functions already mentioned are sequence functions

List vs Vector pros and cons

- List pros
  - Easy to allocate a small piece of list structure and splice it in
    - Have to allocate whole vector at one time
- List cons
  - Takes time proportional to n to get nth member
  - can't go back to previous element from one element

elt

- Sequence function similar to aref, svref, char

```
> (elt '(a b c) 1)
B
> (elt "abc" 1)
#\b
> (elt (vector 1 2 3) 1)
2
> (elt nil 1)
ERROR
```

position

- Returns the position of its first argument in the sequence which is its second argument
- Examples:

```
> (position #\r "microprocessor")
3
> (position #\r "microprocessor" :from-end t)
13
> (position 'b '(a b c))
1
```

find

- Searches for an item in a sequence
- Returns the item found

```
> (find #\a "cat")
#\a
> (find 1000 '( 1 2 4 8 1000))
1000
> (find 5 '(1 2 3 4 1000))
NIL
```

find cont'd

```
> (find 17 '(13 61 34 33 45))
NIL
> (find 17 '(13 61 34 33 45) :test '<)
61
> (defun is-double-of (num1 num2)
       (= (* 2 num1) num2))
is-double-of
> (find 17 '(13 61 34 33 45)
            :test 'is-double-of)
34
> (defun car-is-double-of (num-arg list-arg)
       (= (* 2 num-arg) (car list-arg)))
car-is-double-of
> (find 17
         '((13 31) (61 16) (34 43) (33 33))
         :test  'car-is-double-of)
(34 43)
```

Search: find a subsequence match

- (search “abc” “123abc456”)
  returns 3 (location of beginning of seq1 in seq2)
- returns nil if no match
- subsequence must not have intervening elements:
  (search ’(a b c) ’(a 1 b 2 c)) -> nil

Getting part of a sequence

- (subseq '(a b c d) 1) --> (B C D)
- (subseq '(a b c d) 1 2) --> (B)

Kinds of sequence operations

- Non-destructive
  - copies top level of list
- Destructive
  - modifies how list elements are hooked together

Sorting

- (sort '(1 3 5 7 2 4 6) '<) --> (1 2 3 4 5 6 7)
- (sort '(1 3 5 7 2 4 6) '>) --> (7 6 5 4 3 2 1)
- sort is a destructive operation
- You want to capture the result

  I.e. (setq a (sort a ‘<))

remove

- Removes items from a sequence non-destructively, i.e. copies the sequence

```
> (setq a '(1 2 3 4 3 2 1))
(1 2 3 4 3 2 1)
> (remove 2 a)
(1 3 4 3 1)
> a
(1 2 3 4 3 2 1)
```

delete

- Removes items from a sequence destructively, i.e. splices the sequence

```
> (setq a (list 1 2 3 4 3 2 1))
(1 2 3 4 3 2 1)
> (setq a (delete 2 a))
(1 3 4 3 1)
> a
(1 3 4 3 1)
> (delete 1 a)
(3 4 3)
> a
(1 3 4 3)
```

remove-duplicates

- Also a sequence fn:

```
> (remove-duplicates "abracadabra")
"cdbra”
> (remove-duplicates "abracadabra”
                                :from-end t)
"abrcd "
> (remove-duplicates '(a a b 5 7 7 "hello"))
(A B 5 7 "hello")
```

delete-duplicates

- Delete duplicates destructively

```
> (setq a (list 1 2 3 4 4 5 5 6))
(1 2 3 4 4 5 5 6)
> (delete-duplicates a)
(1 2 3 4 5 6)
> a
(1 2 3 4 5 6)
```

reverse

- Returns a sequence in which the order of the elements in the argument sequence is reversed

```
> (setq a '(1 2 3 4))
(1 2 3 4)
> (reverse a)
(4 3 2 1)
> a
(1 2 3 4)
```

nreverse

- Destructive version of reverse

```
> (setq a (list 1 2 3 4))
(1 2 3 4)
> (nreverse a)
(4 3 2 1)
> a
(1)
```

Allegro CL Certification Program

Lisp Programming Series Level I

Session 1.3.4

Streams and Files

Streams

- A stream is an object that is a source of input and/or a destination for output
- Types of streams include:
  - streams to and/or from the terminal window
  - streams to and/or from a character or binary file
  - streams to and/or from an X window or Windows window
  - socket streams

With-open-file example

```
(with-open-file (I "foo.lisp")
   (loop
      (let ((line (read-line I nil nil)))
         (if (eq1 line nil) (return))
         (write-line
            line
            *standard-output*))))
```

Standardized Stream Variables

- *standard-input*
  - Default stream for read-line, read-char, read, etc.
- *standard-output*
  - Default stream for write-line, write-string,print, etc.
- *terminal-io*
  - Value is ordinarily connected to the user's console
  - Useful place to print log messages and warnings
- *trace-output*
  - Default steam for trace utility
- *query-io*
  - Default stream for stream-y-or-n-p

Create file streams using OPEN

- arguments are filename and keyword arguments
  - :direction -- :input, :output, :IO, or :probe
  - :element-type -- :character or ‘(unsigned-byte 8)
  - :if-does-not-exist -- :error , :create, or nil
  - :if-exists -- :supersede, :overwrite, :error, :append or nil
- returns stream object

File streams must be closed

- (close stream)
- File may appear to be empty until you close it
- With-open-file is the preferred way to open and close a stream

```
(let ((stream (open "c:\\junk.txt"
                    :direction :output)))
  (print 'hello-world stream)
  (close stream))

;; Or equivalently:
(with-open-file (stream "c:\\junk.txt"
                        :direction :output)
  (print 'hello-world stream))
```

Reading Lines

You often have to parse data that is not already in Lisp format. A standard way to parse text data is to read it as strings with read-line:

```
read-line arguments: &optional stream
  eof-error-p eof-value recursive-p
(read-line stream)
=> "Four score and seven "
;; Signals an error when EOF is reached
(read-line stream nil nil)
;; Returns NIL when EOF is reached
```

Read-line example

File data:

=====>Mail arrives Jun 23 13:51:40
xxx.some.com sendmail[14867]: [ID 822293 mail.info] i5NIpeJu014867: from=<user2@some.com>, size=6554, msgid=<B34440A345677F5F480CA4C4E3C54B@officer.corp.some.com>, proto=SMTP, ...

Read the text

```
(let ((text "") line from id f1 f2 i1 i2)
   (block reading
     (loop
      (setq line
        (read-line stream nil nil))
      (if line (setq text
                     (concatenate `string
                       text line))
        (return-from reading))))
    ;; now TEXT is a string containing
    ;; all message lines
```

Parse the text

```
(setq f1 (+ 6 (search "from=<" text)))
(setq f2 (position #\> text :start f1))
(setq from (subseq text f1 f2))
(setq i1 (+ 7 (search "msgid=<" text)))
(setq i2 (position #\> text :start i1))
(setq id (subseq text i1 i2))
(values from id)
 ) ;; closes off let form
user2@some.com
B34440A345677F5F480CA4C4E3C54B@officer.corp.some.com
```

Writing Lines

```
write-line:
  arguments: string &optional stream
             &key start end

(write-line "We hold these truths "
            stream)
;; Writes text and newline marker
(write-line "We hold these truths "
            stream :start 0 :end 3)
;; Writes 'We ' and newline marker
```

Reading and Writing Characters

```
(write-string "Hello world" stream)
;; Writes string but no newline

(read-char stream)
=> #\H
;; Reads a single character

(write-char #\H stream)
;; Writes a single character

(terpri stream)
;; Writes a newline
```

Object Input and Output

```
(write object stream)
;; Outputs the print representation of OBJECT.
:: Returns its object argument.

(read stream)
;; Reads an object from an ascii stream

;; Most objects are print-read compatible:
;; Lists, numbers, strings, symbols, vectors,
;; arrays, lisp source code.
```

Other Ways to Write

- Using the function WRITE is rather uncommon

```
; print readably with prepended #\newline
(print "Five" stream)

"Five"
"Five"
; print readably, without prepended #\newline
(prin1 "Five" stream)
"Five"
"Five"
; print it pretty
(princ "Five" stream)
Five
"Five"
```

Binary Streams

```
(with-open-file  (stream "c:\\junk.bin"
                         :direction :output
                         :element-type
                         '(unsigned-byte 8))
  (write-byte 255 stream))

(with-open-file  (stream "c:\\junk.bin"
                         :element-type
                         '(unsigned-byte 8))
(read-byte stream))
```

Pathnames

- A pathname is an object
- #p"<namestring>"
- File stream creating and file functions use pathnames as arguments

Using pathnames

```
> (setq p (pathname
                "c:\\windows\\system\\user32.dll"))
#p"c:\\windows\\system\\user32.dll"
> (pathname-name p)
"user32"
> (pathname-type p)
"dll"
>(pathname-version p)
:unspecific
```

Using pathnames cont'd

```
> (pathname-directory p)
(:absolute " windows " " system ")
> (pathname-host p)
nil
> (pathname-device p)
" c "
```

A Word on Backslash

- The backslash character(\) is used to "escape" the next character in a string or symbol
  - "His friend was \"curious\"."
  - "c:\\windows\\system\\user32.dll"
  - 'test\a => |TESTa|

Make-pathname

```
(make-pathname
    :device "C"
    :name "user32"
    :type "dll"
    : directory `(:absolute "windows"
                            "system"))

#p"c:\\windows\\system\user32.dll"
```

merge-pathnames

- Missing fields in first argument are provided by second argument

```
(merge-pathname
   (make-pathname :name "odbc")
   #p"c:\\windows\\system\\user32.dll")
#p"c:\\widows\\system\\odbc.dll"
```

- Second argument defaults to the value of *default-pathname-defaults*

File Operations

- rename-file -- original new
- delete-file
- probe-file -- returns true if file is there (actual return value is truename of file)
- file-write-date
- directory -- list of pathnames in directory
- load -- loads a lisp source of fasl file

Directory Operation

- (directory "C:/windows/")
  - Returns a list of directory contents

This was the last class

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