Programming in Lisp

Lecture #3
Kenneth W. Flynn
RPI CS

Outline

- Items from last time
- More on lists
- Arrays
- Structures
- Input / Output
- Control

New Syllabus

- Homework #1 is now due 9/18/98
- Homework #2 is now due 10/2/98
- Homework #3 is still due 10/18/98
- Days for particular topics have also changed, and may do so again as the course develops. The homework dates are firm.
- Exam #1 will be ????

Items From Last Time

- 'T is the symbol for truth. Don't use it as a variable!
- Be careful of parens...
  - (let* ((x 1) (y (+ 2 3))) (* x y))
  - Variable bindings are in a list
  - *let* takes as arguments the code to be evaluated
    - Code should be inside (let)

Items From Last Time II -- Syntax

- *or*
  - Evaluates arguments from left to right; returns first argument that is true. If none are true, returns *nil*.
- *and*
  - Evaluates arguments from left to right; returns *nil* if it encounters a false argument, otherwise returns value of last argument.
  - (and 1 3) returns 3

Items From Last Time III

- >, <, >=, <=
- Examples:
  - (> 3 1) asks: is 3 > 1 ?
  - (> 3 2 1) asks: is 3 > 2 > 1 ?
  - (< 1 2 3) asks: is 1 < 2 < 3
- Similiar for others
- Graham, 353 is incorrect. Try Steele, 293 for more info and examples

Items From Last Time IV

- Recursion
  - Typically try basis case first. Prevents many common errors.
- (quote 13 (/ 1 0)) returns 13. So quote is correct!

Mapping Functions

- All about *mapcar*
- *mapcar* is used to apply a function to each element in one or more lists
- *mapcar*'s first argument is a function
- One by one, the nth arguments of each list are passed to the function

Function Passing: #'

- #' Sharp Quote
- All functions can be passed as parameters
  - #'+
  - #'-
  - #'list
  - #'my-function
- Used in many standard functions
- Generics...

lambda Functions (A rose without a name...)

- Sometimes you create a function just to pass it to something like *mapcar*
- Instead of naming the function, you can create a function with no name -- a *lambda* function
- Simply use the special symbol *lambda* instead of the function name
- #'(lambda (x y) (+ x y)) is our old friend the adder

mapcar Examples

```
> (mapcar #'+ '(1 2) '(1 2))
(2 4)
```

```
> (mapcar #'(lambda (x y)
              (+ x y)
            )
           '(1 2)
           '(1 2)
  )
(2 4)
```

member

- (*member* object list) returns a cons begining with object if present
- *member* takes several *keyword arguments*
- Keyword arguments are of the form :keyword key-value
- :test equilvalance-function
- :key function-to-be-applied-first
- Order is irrelevant

member Examples I

```
> (member 2 '(1 2 3))
(2 3)
> (member 3 '(1 2 3)
          :key #'(lambda (x)
                   (+ x 1)
                  )
   )
(2 3)
```

member Examples II

```
> (member '(1 2)
          '((2 3) (1 2)))
NIL
> (member '(1 2)
          '((2 3) (1 2))
          :test #'equal)
((1 2))
```

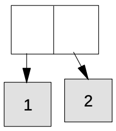
Sequences

- *length*
  - (length '(1 2 3)) returns 3
- *reverse*
  - (reverse '(1 2 3)) returns (3 2 1)
- (*sort* list sort-function)
  - (sort '(3 1 2) #'>) returns (3 2 1)

Dotted Lists

- Proper list refers to a list in which every cdr points to another cons (or nil)
- Dotted list refers to the case when this is not true

(1 . 2)

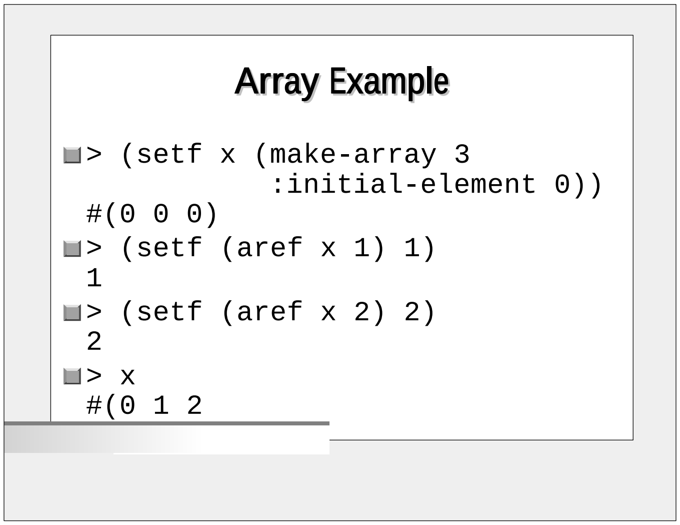


To Dot Or Not To Dot?

- Lisp displays a list with the dot only for the cons which is not proper
- So a cons can be used as a two field data structure.
- Better ways of doing this, though... Structures come to mind

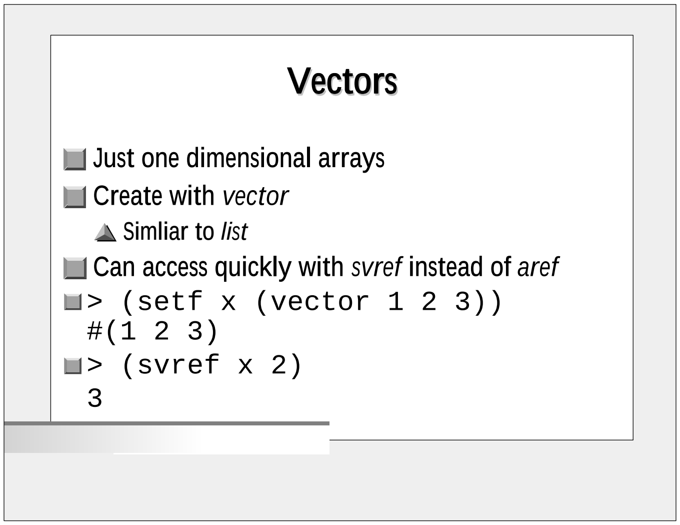
Arrays

- Creation
  - *make-array*
    - 1 required argument -- list of dimensions or integer
    - :initial-element initializes array
- Retrieval
  - *aref*
    - Returns reference to element



Array Example

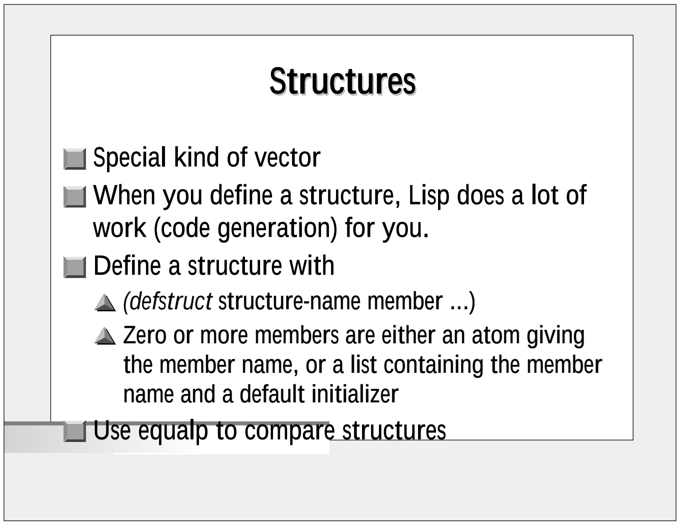
```
> (setf x (make-array 3
             :initial-element 0))
#(0 0 0)
> (setf (aref x 1) 1)
1
> (setf (aref x 2) 2)
2
> x
#(0 1 2
```



Vectors

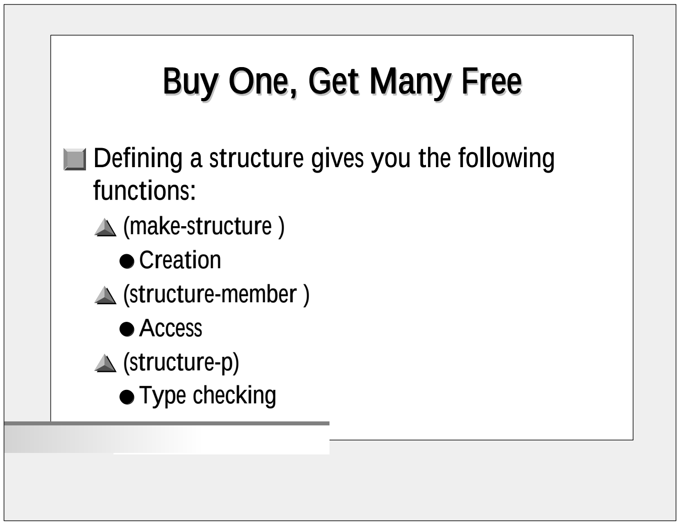
- Just one dimensional arrays
- Create with *vector*
  - Simliar to *list*
- Can access quickly with *svref* instead of *aref*

```
> (setf x (vector 1 2 3))
#(1 2 3)
> (svref x 2)
3
```



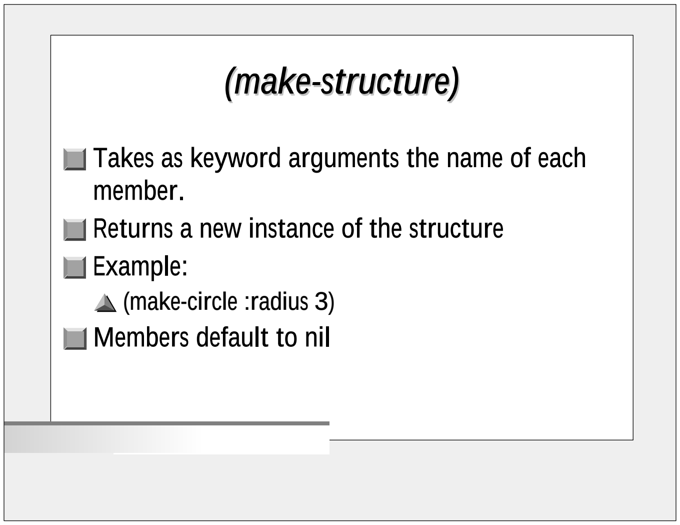
Structures

- Special kind of vector
- When you define a structure, Lisp does a lot of work (code generation) for you.
- Define a structure with
  - *(defstruct* structure-name member ...)
  - Zero or more members are either an atom giving the member name, or a list containing the member name and a default initializer
- Use equalp to compare structures



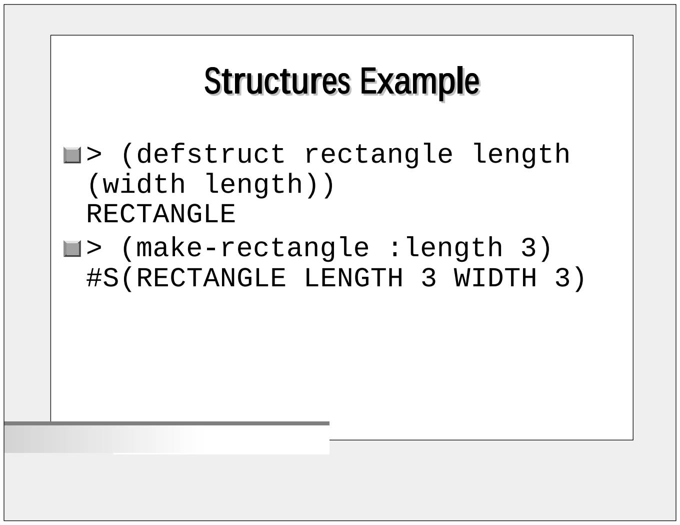
Buy One, Get Many Free

- Defining a structure gives you the following functions:
  - (make-structure)
    - Creation
  - (structure-member)
    - Access
  - (structure-p)
    - Type checking



(make-structure)

- Takes as keyword arguments the name of each member.
- Returns a new instance of the structure
- Example:
  - (make-circle :radius 3)
- Members default to nil



Structures Example

```
> (defstruct rectangle length
(width length))
RECTANGLE
> (make-rectangle :length 3)
#S(RECTANGLE LENGTH 3 WIDTH 3)
```

Structures Example II

-
  ```
  > (setf x (make-rectangle
  :length 3))
  #S(RECTANGLE LENGTH 3 WIDTH 3)
  ```

-
  ```
  > (rectangle-p x)
  (#<STRUCTURE-CLASS RECTANGLE...
  ```

-
  ```
  > (rectangle-p nil)
  NIL
  ```

-
  ```
  > (rectangle-p 5)
  NIL
  ```

Structures Example III

-
  ```
  > (rectangle-length x)
  3
  ```

Input / Output

- Several steps:
  - Create *pathname* -- *(make-pathname :name* name)
  - Create stream -- *(open pathname :direction :input)*
  - Do I/O -- *(read-line stream* input-string)
  - Close stream *(close stream)*
- Or:
  - Create *pathname*
  - Use *(with-open-file)*

Input / Output

- *open* and *with-open-file* take arguments to control stream type
  - :direction [:input | :output]
- > (with-open-file (in-stream (make-pathname :name hello.txt) :direction :input)
  (format t "~A~%" (read in-stream)))
- For more information, refer to Chapter #7 of Graham or stop by my or Jin's office hours.

Control

- Iteration
- Conditionals
- Multiple Values
- A Note On Scope

Iteration: Do

- do

  -
    ```
    (do ((Variable initial-binding update-expression)
          (Variable initial-binding ...) ...) ;Variables
         ((ending-predicate) return-value) ;Returns
         (expression) ...
    )
    ```
- Also do* (evaluates bindings in order each time)

Do, a loop, a useful loop...

- Order
  - Initial values are bound
  - Loop condition is checked (if reached, return)
  - Evaluate expressions
  - Update variables
  - Check loop condition...

Do examples! Now!

```
> (do* ((num 9 (- num 1))
        (root (sqrt num) (sqrt num))
        (lst (cons root 'nil)
             (cons root lst))
       )
       ((= num 1) lst)
       ()
  )
```

```
(1.0 1.4142135623731 1.73205080756888
2.0 2.23606797749979 2.44948974278318
2.64575131106459 2.82842712474619 ...)
```

Iteration Also

- *dolist*
  - Iterates through list items
- dotimes
  - Your basic for loop
- If you understand *do*, you can follow these.
- Refer to p. 88 of Graham for gory details...

Conditionals

- (cond ((predicate) (expressions))
  ((predicate) (expressions)) ...
)
- Powerful! Replaces if then else if then else ...

```
> (cond ((and t nil) 'Nope)
        ((or nil nil) 'Still-nope)
        ((or 13 (/ 1 0)) 'Ah-ha!)
        (t 'Default)
  )
```

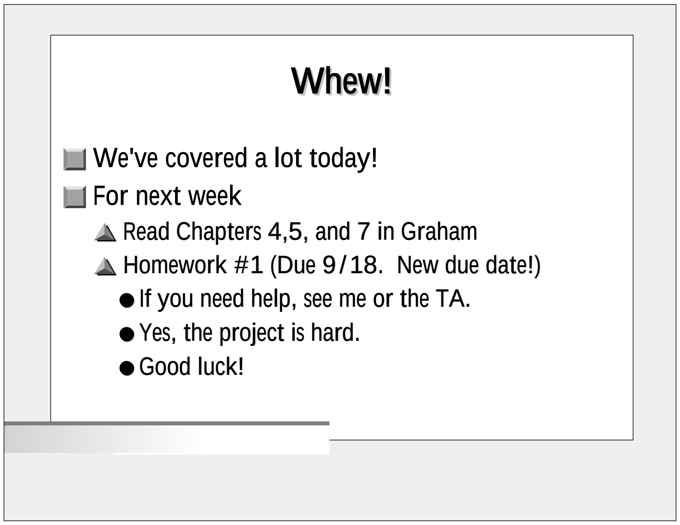
```
AH-HA
```

Multiple Values

- For functions that return multiple values, use *(multiple-values-bind)*
- By example:
  - `> (multiple-value-bind (x pos) (read-from-string "123") (format t "~%Read the number: ~A up to position: ~A~%" x pos))`
  - `Read the number: 123 up to position: 3`

A Note On Scope

- *let*, *defun* both create a new *lexical context*
- Scope!
- Local variables overide globals, just like in C
- Just something to be aware of...



Whew!

- We've covered a lot today!
- For next week
  - Read Chapters 4,5, and 7 in Graham
  - Homework #1 (Due 9/18. New due date!)
    - If you need help, see me or the TA.
    - Yes, the project is hard.
    - Good luck!