

Quick Review

- Numbers, symbols, conses, lists
- Some functions: cons, list, append, car, cdr, nth, reverse, list-length, +, -, *, /
- Some predicates: **null**, listp, atom, =, equal, member
- cond

Assigning values to variables

```
>(setq a 5)
5
> a
5
>(let ((a 6)) a)
6
>(+ a 6)
11
```

Lexical and Dynamic Variables

- Two kinds of variables:
 - Lexical (static)
 - Dynamic
- At any time, either or both kinds of vars with the same name may have a current value

General Rule

- If the symbol occurs textually within a program construct that creates a binding for a var of the same name, then the reference is to the var specified by the binding.
- Otherwise, it is taken to refer to the dynamic var of the same name

Scope and Extent

- A lexically bound var can be referred to only by forms occurring at any place textually within the program construct that binds the var
- A dynamically bound var can be referred to at any time from the time the binding is made

Parameters

- Formal parameters are bound to actual parameters for the duration of the function call
- Parameters are passed by value - but **beware of destructive functions**

let vs let*

```
(let ((var1 val1)
      (var2 val2)
      ...)
  Body-1
  Body-2
  Body-3
  ...
  Body-n)
```

val1, val2, etc are evaluated in order, then assigned to vars in parallel

```
(let* ((var1 val1)
       (var2 val2)
       ...)
  Body-1
  Body-2
  Body-3
  ...
  Body-n)
```

Vars are bound to their corresponding values in order

Printing

- `print` function prints its argument and then returns it:

```
>(print 3)
3
3
```

- First 3 was printed; second was returned

format

- For formatted output and output to locations other than the screen

```
(format t "An atom: ~S-%and a list:
~S-%and an integer: ~D-%" nil (list 5)
6)
```

```
An atom: NIL
And a list: (5)
And an integer: 6
```

Data Structures

- **Stacks**
 - List can be used as a stack
 - Macros `push` and `pop` are defined
- **Arrays**
 - Function `make-array`
 - Many useful functions for manipulating arrays
- **Structures**
 - Named record structures with named components
 - Constructor, access, and assignment constructs automatically defined
 - All the benefits of type-checking, modularity, etc.

```
(format t "An atom: ~S-%and a list: ~S-%and an
integer: ~D-%" nil (list 5) 6)
```

- First arg is either `t`, `nil`, or a stream
- `nil` means not to print anything - just return a string containing the output
- Second arg is a formatting template; gen'lly contains formatting directives:
 - `~A` ascii, `~S` s-expression (list or atom) `~D` decimal, `~%` newline, etc

defstruct example: water jug problem

```
(defstruct waterjug
  capacity
  current)
(setq 3gal (make-waterjug
           :capacity 3
           :current 0))
(setf (waterjug-current 3gal) 2)
(waterjug-current 3gal)
```

Functions as arguments

- Funcall
 - calls its first argument on its remaining arguments
 - Function may be Lisp- or user-defined

Examples

```
>(funcall '+ 3 4)
7
>(defun myfctn (x) x)
>(funcall 'myfctn 'a)
>(setq mine 'myfctn)
>(funcall mine 'a)
```

mapcar

- Applies a function to each element of a list
- Collects results in another list
- First argument of mapcar must be a function of one argument

Examples

```
>(mapcar 'not '(t nil t nil t nil))
(nil t nil t nil t)
>(mapcar #'(lambda(x) (+ x 2)) '(1 2 3 4))
(3 4 5 6)
```

Lambda can be used to create temporary, nameless functions.

Alternative for cond: if

- First argument determines whether second or third should be executed
 - Third argument is optional
- ```
>(if t 5 6)
5
>(if nil 5 6)
6
```
- Note:
- and, or, not are all lisp functions
  - and/or are short-circuiting

## Iteration

- do, do\*, dolist
  - do and do\* allow for the specification of a return value
  - Result of dolist is always nil

But don't use these for Assignment 1