Introduction to Lisp

based on slides by Dana Nau

Outline

- This is a quick introduction to Lisp
- I assume you know enough about computer languages that you can pick up new ones quickly, so I'll go pretty fast
- If I go too fast, please say so and I'll slow down

- Lisp Basics
  - Lisp syntax: parenthesized prefix notation
  - Lisp interpreter: read-eval-print loop
  - Nested evaluation
  - Preventing evalution (quote and other special forms)
  - Forcing evaluation (eval)

What does LISP stand for?

What does LISP stand for?

A speech defect in which you can't pronounce the letter 's'?

What does LISP stand for?

A speech defect in which you can't pronounce the letter 's'?

L*ooney* **I***diotic* **S***tupid* **P***rofessor?*

What does LISP stand for?

A speech defect in which you can't pronounce the letter 's'?

L*ooney* **I***diotic* **S***tupid* **P***rofessor?*

L*ong* **I***ncomprehensible* **S***tring of* **P***arentheses?*

What does LISP stand for?

A speech defect in which you can't pronounce the letter 's'?

L*ooney* **I***diotic* **S***tupid* **P***rofessor?*

L*ong* **I***ncomprehensible* **S***tring of* **P***arentheses?*

LIS*t* **P***rocessing?*

LISP

Lisp's primary data structure is a linked list

- ♢ Lots of built-in list functions
- ♢ Don't need a "pointer" data type; the functions work at a higher level
- ♢ Even Lisp programs are lists!
  Syntax for function calls: `(function arg1 arg2 ... argn)`
- ♢ Example: if `x = -3`, then
  the expression `(if (>= x 0) x (- x))` returns `3`

  I'll write this as `(if (>= x 0) x (- x))` $\Longrightarrow$ `3`

LISP

AI programs often need to do a combination of symbolic/numeric reasoning
- ♢ Lisp is the best language I know for this

Eric Raymond, the author of *The Cathedral and the Bazaar*, *The Art of Unix Programming*, *The New Hacker's Dictionary*, . . ., says:

> Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

Examples of LISP applications
- ♢ AutoCAD - widely used computer-aided design system
- ♢ Emacs Lisp - Emacs's extension language
- ♢ Remote Agent software - deployed on NASA's Deep Space 1 (1998)
- ♢ ITA Software's airline fare shopping engine - used by Orbitz
- ♢ Yahoo! Store - e-commerce software (about 20,000 stores as of 2003)

LISP

Originated by John McCarthy in 1959 as an implementation of recursive function theory.

First language to have

- ♢ Conditionals - if-then-else constructs
- ♢ A *function* type - functions are first-class objects
- ♢ Recursion
- ♢ Typed **values** rather than typed **variables**
- ♢ Garbage collection
- ♢ Commands = functional expressions
- ♢ A *symbol* type
- ♢ Built-in extensibility
- ♢ The whole language always available – programs can construct and execute other programs on the fly

These features have gradually been added to other languages

Common Lisp

- ♢ Features of other languages have gradually been added to Lisp
- ♢ Since Lisp is so easy to extend,
  many groups created their own Lisp dialects
    BBN-Lisp, Franz Lisp, Interlisp-10, Interlisp-D, Lisp 1.5,
    Lisp/370, Maclisp, Scheme, Zetalisp, . . .
- ♢ Common Lisp was created to unify the main dialects
  $\Longrightarrow$ contains multiple constructs to do the same things

You'll be using Allegro Common Lisp on solaris.grace.umd.edu
Documentation: links on the class page

Launching Allegro Common Lisp

Login to **solaris.grace.umd.edu** using your Glue account

◇ For more info: "How to use OIT's Detective Cluster" on the class page

---

You'll be using **Allegro Common Lisp**. Here's how to launch it. `tap allegro70`

```
alisp
```

Running Common Lisp elsewhere

◇ Allegro Common Lisp is installed on some of the CS Dept machines

◇ You can also get a Common Lisp implementation for your own computer
Check "implementations" on the class page

But **make sure** your program runs correctly using **alisp** on **solaris.grace.umd.edu**, because that's where we'll test it.

Starting Out

- When you run Lisp, you'll be in Lisp's command-line interpreter
- You type expressions, it evaluates them and prints the values

```
[posh:~] getoor% alisp
```

. . . *several lines of printout* . . .

```
CL-USER(1): (+ 2 3 5)
10
CL-USER(2): 5
5
CL-USER(3): (print (+ 2 3 5))

10
10
CL-USER(4): (exit)
; Exiting Lisp
[posh:~] getoor%
```

Some Common Lisps also have GUIs; check the documentation

Atoms

◇ Every Lisp object is either an **atom** or a **list**

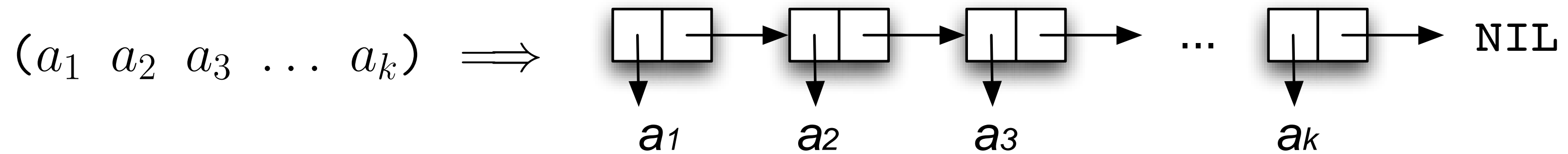
◇ Examples of atoms:

numbers	`235.4`	`2e10`	`#x16`	`2/3`
variables	`val39`	`2nd-place`	`*foo*`	
constants	`pi`	`t`	`nil`	`:keyword`
strings, chars	`"Hello!"`	`#\a`		
arrays	`#(1 "foo" A)`	`#1A(1 "foo" A)`	`#2A((A B C) (1 2 3))`	
structures	`#s(person first-name dana last-name nau)`			

◇ For Lisp atoms other than characters and strings, case is irrelevant:
`foo` = `FOO` = `Foo` = `FoO` = ...

Lists

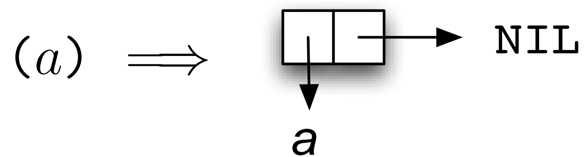
$(a_1\ \ a_2\ \ a_3\ \ \ldots\ \ a_k)\ \Longrightarrow$



$a_1, a_2, \ldots, a_k$ may be atoms or other lists

The empty list is called `()` or `NIL`; it's both a list and an atom

$(a)\ \Longrightarrow$

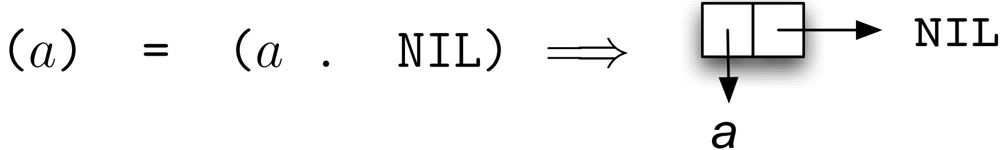


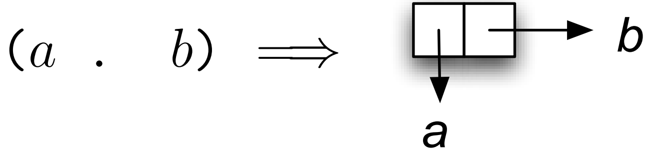
Examples:

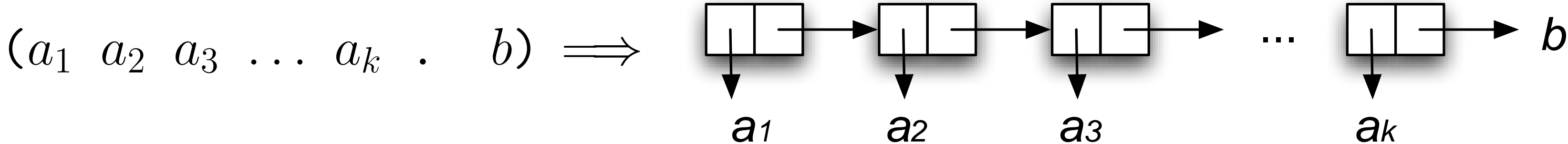
```
(235.4  (2e10  2/3)  "Hello, there!"  #(1 4.5 -7))

(foo  (bar ((baz)) asdf)  :keyword)
```

Dot notation







Example:

```
(235.4  (2e10  2/3)  "Hello, there!"  #(1 4.5 -7) . foobar)
```

Defining Lisp Functions

```
(defun fib (n)
  (if (< n 3)
      1
    (+ (fib (- n 1))
       (fib (- n 2)))))
```

Suppose the definition is in a file called `fibonacci.cl`

```
[posh:~] getoor% alisp
International Allegro CL Enterprise Edition
... several more lines ...
CL-USER(1): (load "fibonacci")
; Loading /homes/research/getoor/fibonacci.cl
T
CL-USER(2): (list (fib 1) (fiB 2) (fIb 3) (fIB 4) (Fib 5) (FiB 6))
(1 1 2 3 5 8)
CL-USER(3):
```

Style

```
;;; This is a comment formatted as a block of text
;;; outside of any function definition

(defun fib (n)
  ;; A comment on a line by itself
  (if (< n 3)
      1                  ; A comment on the same line as some code
      (+ (fib (- n 1))
         (fib (- n 2)))))

(setq *global-variable* 10)
(let (local-variable)
     (setq local-variable 15))
```

Read Norvig's tutorial on Lisp programming style

There's a link on the class page

Compiling and Loading

```
CL-USER(3): (compile-file "fibonacci")
;;; Compiling file fibonacci.cl
;;; Writing fasl file fibonacci.fasl
Warning: No IN-PACKAGE form seen in
         /homes/research/getoor/fibonacci.cl.  (Allegro Presto will be
         ineffective when loading a file having no IN-PACKAGE form.)
;;; Fasl write complete
#p"fibonacci.fasl"
T
NIL
CL-USER(3):
```

Compiling will make your programs run faster, and may detect some errors

◇ Creates a binary file called `fibonacci.fasl` but *doesn't* load it

◇ `(load "fibonacci")` will do the following:

  load `fibonacci.fasl` if it exists, else load `fibonacci.cl` if it exists, else load `fibonacci.lisp` exists, else error

◇ Can also do `(load "fibonacci.cl")`, etc.

Editing Lisp files

Use a text editor that does parenthesis matching!

◇ Emacs is good *if you know how to use it*, because it knows Lisp syntax

◇ Emacs's built-in Lisp is **not** Common Lisp. Don't use it for your projects!

◇ Here's a way to run Common Lisp in an Emacs buffer

Put this in your `.emacs` file:

```
(load
   "/afs/glue.umd.edu/software/allegro/7.0/Solaris/acl70/eli/fi-site-init")
```

Launch Emacs, and type `M-x fi:common-lisp`

For additional info: "running Lisp inside Emacs" on the class page

Lisp functions

Next, I'll summarize some basic Common Lisp functions

◇ For additional details and additional functions, see *ANSI Common Lisp* and the Allegro documentation (URL on the class page)

Numeric functions

`+ -`	sum, difference	`(* 2 3 4)` $\Longrightarrow$ 24
`* /`	product, quotient	`(/ (+ 2 1 1) (- 3 1))` $\Longrightarrow$ 2
`sqrt`	square root	`(sqrt 9)` $\Rightarrow$ 3
`expt`	exponentiation	`(expt 3 4)` $\Longrightarrow$ 81
`min`, `max`	minimum, maximum	`(min -1 2 -3 4 -5 6)` $\Longrightarrow$ -5
`abs`, `round`	absolute val, round	`(abs (round -2.4))` $\Longrightarrow$ 2
`truncate`	integer part	`(truncate 3.2)` $\Longrightarrow$ 3
`mod`	remainder	`(mod (5.6 5)` $\Longrightarrow$ 0.6
`sin`, `cos`, `tan`	trig functions (radians)	`(sin (/ pi 2)` $\Longrightarrow$ 1.0

Special Forms

Used for side-effects; don't follow the normal Lisp rule of evaluating all args before applying the function

`defun`	define a function	(defun *name* (*args*) *body*)
`defstruct`	define a structure	(defstruct *name* *fields*)
`setq`	assign a value to a variable	(setq foo #(1 2 3 4)) ⟹ foo = #(1 2 3 4) (setq bar foo) ⟹ bar = #(1 2 3 4) (setq bar 'foo) ⟹ bar = FOO
`setf`	like `setq`; works on arrays, structures	(setf foo #(1 2 3 4)) (setf (elt foo 0) 5) ⟹ foo = #(5 2 3 4)
`'`, `quote`	return the arg without evaluating it	(+ 2 3) ⟹ 5 (quote (+ 2 3)) ⟹ (+ 2 3) '(+ 2 3) ⟹ (+ 2 3) (eval '(+ 2 3)) ⟹ 5

List functions

`first`, `car`	1st element	`(first '(a b c d))` $\Longrightarrow$ `a`
`second`, ..., `tenth`	like `first`	`(third '(a b c d))` $\Longrightarrow$ `c`
`rest`, `cdr`	all but 1st	`(rest '(a b c d))` $\Longrightarrow$ `(b c d)`
`nth`	nth element, n starts at 0	`(nth 2 '(a b c d))` $\Longrightarrow$ `c`
`length`	no. of elements	`(length '((a b) c (d e)))` $\Longrightarrow$ `3`
`cons`	inverse of `car` & `cdr`	`(cons 'a '(b c d))` $\Longrightarrow$ `(a b c d)` `(cons '(a b) 'c)` $\Longrightarrow$ `((a b) . c)`
`list`	make a list	`(list '(a) '(b c) (+ 2 3))` $\Longrightarrow$ `(a (b c) 5)`

`append`	append lists	`(append '(a) '(b c) '(d))` $\Longrightarrow$ `(a b c d)`
		`(append '(a) '(b c) 'd)`
		$\Longrightarrow$ `(a b c . d)`

Predicates

`numberp`, `integerp`, `stringp`, `characterp` `evenp`, `oddp`	test whether arg is a number, integer, string, character, etc.	`(numberp 5.78)` $\Longrightarrow$ `T`
		`(integerp 5.78)` $\Longrightarrow$ `NIL`
		`(characterp \#a)` $\Longrightarrow$ `T`

`listp`, `atom`, `null`, `consp`	test whether arg is a list, atom, empty/nonempty list	`(listp nil)` $\Longrightarrow$ `T` `(consp nil)` $\Longrightarrow$ `NIL`
`<`, `<=`, `=`, `>=`, `>`	numeric comparisons	arg must be a number
`string<`, `string<=`, ...	string comparisons	arg must be string or char

`eql`, `equal`	equality tests; they work differently on lists and strings	`(setq x '(a))`
		`(eql x x)` $\Longrightarrow$ `T`
		`(eql x '(a))` $\Longrightarrow$ `NIL`
		`(equal x '(a))` $\Longrightarrow$ `T`

`and`, `or`, `not`	logical predicates; `not` and `null` are identical	`(not (evenp 8))` $\Longrightarrow$ `NIL` `(and 3 'foo T)` $\Longrightarrow$ `T`

More special forms: conditionals

`if`	if-then-else	`(if` *test* *expr1* `[`*expr2*`])` if *test* is non-`NIL` then return *expr1*, else return *expr2* (or `NIL`)

`cond`	extended if-then-else	`(cond (`*test1* *result11* *result12* ... `)`
		`(`*test2* *result21* *result22* ... `)`
		... `)`

`case`, `ecase`	like C's "switch"; `case` allows default values; `ecase` signals a continuable error	`(case x (a 5)`
		`((d e) (print "d or e") 7)`
		`((b f) 3)`
		`(otherwise 9))`

More special forms

`(print foo)` prints the value of `foo`

`(format t "~s equals ~s" 'foo 3)` prints `FOO equals 3`

Lots of options; see *Ansi Common Lisp*

`(read)` reads and returns (but does not evaluate) a Lisp expression

`let` and `let*` declare local variables simultaneously and sequentially

```
(setq a 1) (setq b 2)
(let (b) (list a b)) ⟹ (1 NIL)
(let ((b 5)) (list a b)) ⟹ (1 5)
(let ((a b) (b a)) (list a b)) ⟹ (2 1)
```

`(let* ((`x_1 v_1`) ... (`x_n v_n`))` e_1 e_2 `...` e_n`)` is the same as

`(let (`x_1 `...` x_n`) (setq` x_1 v_1`) ... (setq` x_n v_n`)` e_1 e_2 `...` e_n`)`

`(progn` e_1 e_2 `...` e_n`)` is the same as `(let nil` e_1 e_2 `...` e_n`)`

`(prog1` e_1 e_2 `...` e_n`)` is like `progn` but returns the value of e_1 instead of e_n

Macros

Macros expand inline into other pieces of Lisp code

`push` and `pop` use lists as stacks

```
(push x foo)  =  (setq foo (cons x foo))
(pop foo)  =  (prog1 (first foo) (setq foo (rest foo)))
```

`when` and `unless` are like "if" and "if not" without the "else"

(when $test$ e_1 e_2 ... e_n) = (if $test$ (progn e_1 e_2 ... e_n))

(unless $test$ e_1 e_2 ... e_n) = (if (not $test$) (progn e_1 e_2 ... e_n))

Lisp also lets you define your own macros, but I won't discuss this

---

Lisp operator: a function, special form, or macro

Some of the debugging messages differ
Some funcs take other funcs (but not special forms or macros) as args

Loops

`do` and `do*` are somewhat like C's "for"

`do` sets all of the loop iterators at once, `do*` sets them sequentially

The iterators are local to the `do` or `do*`

This example prints 2, 4, 6, 8, 10 on separate lines, and returns DONE

```
(do* ((a 1 (+ a 1))       ; start a at 1, add 1 each time
      (b 2 (* a 2)))      ; start b at 2, set it to 2*a each time
     ((> a 5) 'done)      ; quit when a>5, and return DONE
   (print b))
```

```
(dolist (foo '(a 1.5 "bar")) ...code...) ; use foo = a, 1.5, "bar"

(dotimes (foo 5) ...code...)             ; use foo = 0, 1, 2, 3, 4
```

`return` can exit from the middle of a loop:

```
(dolist (foo '(a 1.5 "bar"))
   (if (numberp foo)
       (return foo)))
```

More loops

`loop` is an all-in-one iteration macro

Graham doesn't like it, because complex cases can be very hard to understand – see *ANSI Common Lisp*, pp. 239-244

But simple cases can be easier to use than `do`

```
(loop for x in '(a b c d)
      for y in '(1 2 3 4)
      collect (list x y))

⟹ ((A 1) (B 2) (C 3) (D 4))
```

```
(do ((x '(a b c d) (cdr x))
     (y '(1 2 3 4) (cdr y))
     (z nil (cons (list (car x) (car y))
                  z)))
    ((null x) (reverse z)))
```

```
(loop for x in '(a b c d)
      for y in '(1 2 3 4)
      collect x collect y)

⟹ (A 1 B 2 C 3 D 4))
```

```
(do ((x '(a b c d) (cdr x))
     (y '(1 2 3 4) (cdr y))
     (z nil (cons (car x)
                  (cons (car y) z))))
    ((null x) (reverse z)))
```

More loops

Example: use an infinite loop to write your own Lisp interpreter:

```
(loop
   (format t "~%> ")
   (format t "~&~s" (eval (read))))
```

For more info about `loop`, see the links for `loop` on the class page

Interacting with Allegro Common Lisp

- Allegro Common Lisp has a command-line interface
- When it prompts you for input, you can type
  any Common Lisp expression or any Allegro command
- Allegro command syntax: a colon followed by the command name
  - `:cd foo` changes the current directory to foo
  - `:help cd` prints a description of the :cd command
  - `:help` prints a list of all available commands
- **The Allegro commands aren't part of Common Lisp itself**
  They won't work inside Lisp programs
  They are only available interactively, at Allegro's input prompt
- What Allegro commands are available depends on whether you're
  at the top level or inside the debugger

Debugging

- `(trace foo)` or `:trace foo`
  Lisp will print a message each time it enters or exits the function foo
  Several optional args; see the Allegro documentation
- To turn it off: `(untrace foo)` or `:untrace foo` or `(untrace)` or `:untrace`
- `(step` *expression*`)` or `:step` *expression*
  will single-step through the evaluation of *expression*
  Doesn't work on compiled code
- For more info about debugging, see Appendix A of *ANSI Common Lisp* and "debugging" on the class page
- Transcribing your Lisp session – links on the class page

The debugger

```
CL-USER(55): (fib (list 3 5))
Error: `(3 5)' is not of the expected type `REAL'
  [condition type: TYPE-ERROR]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this process.
[1] CL-USER(56): (fib "asdf")
Error: `"asdf"' is not of the expected type `REAL'
  [condition type: TYPE-ERROR]

Restart actions (select using :continue):
 0: Return to Debug Level 1 (an "abort" restart).
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this process.
[2] CL-USER(57):
```

At this point, you're two levels deep in the Lisp debugger
You can type Lisp functions or Allegro commands

Allegro debugging commands

- Type `:continue 0` or `:continue 1` or `:continue 2` to do what's specified
- `:pop` or control-D goes up one level; `:pop 2` goes up two levels
- `:zoom` prints the current runtime stack
- `:local` or `:local n` prints the value of `fib`'s parameter `n`, which is `"asdf"`
- `:set-local n` sets the local variable `n`'s value
- `:current` prints `(< "asdf" 3)`, the expression that caused the error
- `:return` returns a value from the expression that caused the error, and continues execution from there
- Type `:help` for a list of other commands

Let's do `:continue 0` to return to the top level,
then `(trace fib)` to turn on tracing for `fib`, then `(fib "asdf")` again . . .

```
 0[1]: (FIB "asdf")
Error: ‘"asdf"’ is not of the expected type ‘REAL’
  [condition type: TYPE-ERROR]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] CL-USER(71): :current
(< "asdf" 3)
[1] CL-USER(72): :set-local n 4
[1] CL-USER(73): :return nil
   1[1]: (FIB 3)
     2[1]: (FIB 2)
     2[1]: returned 1
     2[1]: (FIB 1)
     2[1]: returned 1
   1[1]: returned 2
   1[1]: (FIB 2)
   1[1]: returned 1
 0[1]: returned 3
3
```

Break and continue

- `break` will make a breakpoint in your code; its syntax is like `format`
- `:continue` will continue from the breakpoint

```
CL-USER(12): (defun foo (n)
                (format t "Hello")
                (break "I'm broken with n = ~s" n)
                (format t "I'm fixed with n = ~s" n))
FOO
CL-USER(13): (foo 3)
Hello
Break: I'm broken with n = 3

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this process.
[1c] CL-USER(14): :continue
I'm fixed with n = 3
NIL
```

Functions that take functions as arguments

- #'*func* quotes *func* as a function

```
(setq y (list #'+ #'cons)) ⟹ (#<Function +> #<Function CONS>)
```

- If *expr* is an expression whose value is *func*, then
  (funcall *expr* e_1 e_2 ... e_n) = (*func* e_1 e_2 ... e_n)

```
(funcall #'+ 1 2 3) ⟹ 6
(funcall (first y) 1 2 3) ⟹ 6
(funcall #'append '(A B) '(C D) '(E F G)) ⟹ (A B C D E F G)
```

- (apply *expr* (e_1 e_2 ... e_n)) = (*func* e_1 e_2 ... e_n)

```
(apply #'+ '(1 2 3)) ⟹ 6
(apply #'append '((A B) (C D) (E F G))) ⟹ (A B C D E F G)
```

- (apply *expr* a_1 ... a_j (b_1 b_2 ... b_k)) = (*func* a_1 ... a_j b_1 ... b_j)

```
(apply #'+ 1 1 1 '(2 2)) = (apply #'+ '(1 1 1 2 2)) ⟹ 7
```

Mapping functions

Like before, suppose *expr* is an expression whose value is *func*

- (mapcar *expr* *list*) calls *func* on each member of *list* and returns a list of the results

```
(mapcar #'sqrt '(1 4 9 16 25)) ==> (1 2 3 4 5)
(mapcar #'first '((A B) (C) (D E))) ==> (A C D)

(setq y (lambda (x) (+ x 10)))
(mapcar y '(1 2 5 28)) ==> (11 12 15 38)
```

- If *func* is n-ary, you can do (mapcar *expr* $list_1$ $list_2$ ... $list_n$)
  This takes *func*'s i'th arg from the i'th list

```
(mapcar #'+ '(1 2 3 4) '(5 6 7 8)) ⟹ (6 8 10 12)
(mapcar #'list '(a b c) '(e f g)) ⟹ ((A E) (B F) (C G))
```

- (maplist *expr* *list*) calls *func* on successive `cdr`s of *list*

```
(maplist #'identity '(a b c)) ==> ((A B C) (B C) (C))
```

More list functions

```
(member '(1 2) '((0 1) (1 2) (2 3)))  ==> NIL

(member '(1 2) '((0 1) (1 2) (2 3)) :test #'equal)  ==> ((1 2) (2 3))

(member 2 '((0 1) (1 2) (2 3)) :key #'second)  ==> ((1 2) (2 3))

(subsetp '(a b) '(x a y b z))  ==> T

(union '(a b c d) '(d c e f))  ==> '(B A D C E F)

(intersection '((a x) (b y)) '((b z) (c w)) :key #'first)  ==> ((B Y))

(set-difference '(a b c) '(b c))  ==> (A)

(assoc 'B '((A B) (B C) (C D)))  ==> (B C)
```

(`copy-list` *expr*) returns new list whose elements are the ones in *expr*
(`copy-tree` *expr*) copies the entire tree structure

Functions of sequences

Some functions work on any sequence (list, a character string, or vector)

```
(elt (list 'a 'b 'c 'd 'e) 0) ==> A
(elt "abcde" 0) ==> #\a

(subseq '(a b c d e) 2 4) ==> (C D)
(subseq #(a b c d e) 2 4) ==> #(C D)

(find #\d "abcde") ==> #\d
(position #\d "abcde") ==> 3

(find '(A B) '((A B) (w x) (y z)) ==> NIL
(find '(A B) '((A B) (w x) (y z)) :test #'equal) ==> (A B)
(find '(A B) #((A B) (w x) (y z)) :test #'equal) ==> (A B)

(defun Almost-Equal (Num1 Num2)
  (<= (abs (- Num1 Num2)) 0.1))
(find pi #(2.9 3.0 3.1 3.2 3.3) :test #'Almost-Equal) ==> 3.1
```

Functions of sequences, continued

```
(find A '((A B) (C D E) (F)) :key #'first) ==> (A B)
(find 3 '((A B) (C D E) (F)) :key #'length) ==> (C D E)

(find '(D E) '((A B) (C D E) (F)) :key #'rest :test #'equal) ==> (C D E)

(find-if #'evenp #(1 2 3 4)) ==> 2
(find-if #'oddp '((0 1) (1 2) (2 3)) :key #'first) ==> (1 2)
```

Some sequence functions that work like `find`, `find-if`, and `find-if-not`:

```
position     position-if     position-if-not
remove       remove-if       remove-if-not
replace      replace-if      replace-if-not
subseq       subseq-if       subseq-if-not
substitute   substitute-if   substitute-if-not
subst        subst-if        subst-if-not
```

Destructive versus nondestructive functions

◊ The functions on the previous page are *nondestructive*
They copy the sequence rather than modifying the original

◊ There are destructive versions of the same functions
e.g., `delete` is the destructive counterpart of `remove`
Also, can use `setf` to do destructive modifications

◊ Destructive modifications can have unexpected side-effects

```
(setq x '(a b c))    ==> (A B C)
(setq y '(d e f))    ==> (D E F)
(setq z (nconc x y)) ==> (A B C D E F)
x                    ==> (A B C D E F)
```

Avoid doing destructive modifications unless (i) there's a **very** good reason why they're needed and (ii) you're **very** sure you know what you're doing

Lots of other features

Common Lisp has a huge feature set

`random` - return a random number

`make-hash-table` - return a hash table

packages

object-oriented programming

continuable errors

catch/throw

...

Example: compute a set's power set

```
(defun power1 (x pset)
   (mapcar #'(lambda (a) (cons x a)) pset))

(defun powerset (x)
  (let ((pset (list nil)))
    (dolist (e x)
      (setq pset
            (append (power1 e pset)
                    pset)))
    pset))
```

or:

```
(defun powerset (x)
  (let ((pset (list nil)))
    (mapcar #'(lambda (e)
                (setq pset (append (power1 e pset) pset)))
            x)
    pset))
```

Copy-list examples 1 and 2

From the Lisp FAQ

1.
```
(defun copy-list (list)
  (let ((result nil))
    (dolist (item list result)
      (setf result
            (append result (list item))))))
```

2.
```
(defun copy-list (list)
  (let ((result nil))
    (dolist (item list
              (nreverse result))
      (push item result))))
```

1st implementation uses `APPEND` to put elements onto the end of the list. Traverses the entire partial list each time ⇒ quadratic running time.

2nd implementation improves on this by iterating down the list twice: once to build up the list in reverse order, and the second time to reverse it.

Copy-list examples 3 and 4

```
3.  (defun copy-list (list)
      (mapcar #'identity list))

4.  (defun copy-list (list)
      (let ((result (make-list (length list))))
        (do ((original list (cdr original))
             (new result (cdr new)))
            ((null original) result)
            (setf (car new) (car original)))))
```

3rd and 4th implementations:
efficiency usually similar to the 2nd one
depends on the Lisp implementation

The 3rd implementation is the easiest to understand

Copy-list example 5

```
5.  (defun copy-list (list)
      (when list
        (let* ((result (list (car list)))
               (tail-ptr result))
          (dolist (item (cdr list) result)
            (setf (cdr tail-ptr) (list item))
            (setf tail-ptr (cdr tail-ptr))))))
```

5th implementation iterates down the list only once

Runs at the same speed as the 2nd implementation, or slightly slower.

Keeps a pointer to the tail of the list, destructively modifies the tail to point to the next element

Copy-list examples 6 and 7

```
6.  (defun copy-list (list)
      (loop for item in list collect item))

7.  (defun copy-list (list)
      (if (consp list)
          (cons (car list)
                (copy-list (cdr list)))
        list))
```

6th implementation: uses the `loop` macro, usually expands into code similar to the 3rd.

7th implementation: copies dotted lists, and runs in linear time, but isn't tail-recursive.