Last update: February 16, 2010

INTRODUCTION TO LISP

DANA NAU

Outline

◇ I assume you know enough about computer languages that you can learn new ones quickly, so I'll go pretty fast

◇ If I go too fast, **please say so** and I'll slow down

Assignment:

1. Get a TerpConnect account if you don't already have one

2. Start reading one or more of the following (you'll need to figure out which parts correspond to my lecture)
- *ANSI Common Lisp* - available at the bookstore
- *Common Lisp the Language, 2nd edition* (URL on the class page)
- *Allegro Documentation* (URL on the class page)

3. Read Norvig's tutorial on Lisp programming style (URL on the class page)

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?*

What is LISP?

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

The 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
- Programs made entirely of functional expressions that return values
- A *symbol* type
- Built-in extensibility
- The whole language always available – programs can construct and execute other programs on the fly

Most of these features have gradually been added to other languages

LISP's influence on other languages

It seems to me that there have been two really clean, consistent models of programming so far: the C model and the Lisp model. These two seem points of high ground, with swampy lowlands between them. As computers have grown more powerful, the new languages being developed have been moving steadily toward the Lisp model. A popular recipe for new programming languages in the past 20 years has been to take the C model of computing and add to it, piecemeal, parts taken from the Lisp model, like runtime typing and garbage collection.

– Paul Graham, *The Roots of Lisp*, May 2001

We were after the C++ programmers. We managed to drag a lot of them about halfway to Lisp.

– Guy Steele, co-author of the Java spec

More quotes at `http://lispers.org/`

LISP applications

AI programs often need to combine symbolic and numeric reasoning.
Lisp is the best language I know for this.

- ♢ Writing SHOP (my group's AI planning system) took a few weeks in Lisp
- ♢ Writing JSHOP (Java version of SHOP) took several months

Lisp is less used outside of AI, but there are several well-known LISP applications:

- ♢ AutoCAD - computer-aided design system
- ♢ Emacs Lisp - Emacs's extension language
- ♢ ITA Software's airline fare shopping engine - used by Orbitz
- ♢ Parasolid - geometric modeling system
- ♢ Remote Agent software - deployed on NASA's Deep Space 1 (1998)
- ♢ Script-Fu plugins for GIMP (GNU Image Manipulation Program)
- ♢ Yahoo! Merchant Solutions - e-commerce software

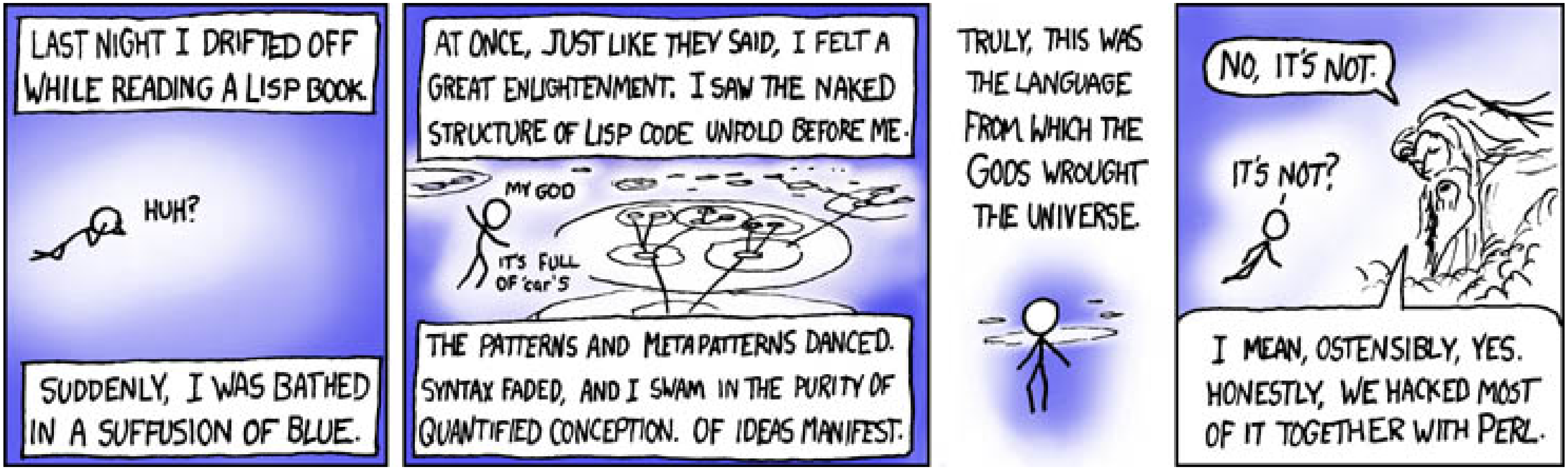
Why learn LISP?

Several universities teach Scheme (a dialect of Lisp) in their introductory Computer Science classes

> LISP is worth learning for a different reason — 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.
>
> – Eric Raymond, *How to Become a Hacker*, 2001

More about Lisp and Enlightenment . . .

From http://xkcd.com/224



Common Lisp

- ♢ Lisp's uniform syntax makes it very easily extensible
  Just write new functions and include them when launching Lisp

- ♢ This led many groups to create their own Lisp dialects:
  BBN-Lisp, Franz Lisp, Interlisp-10, Interlisp-D, Le-Lisp, Lisp 1.5,
  Lisp/370, Lisp Machine Lisp, Maclisp, NIL, Scheme, T, ZetaLisp, . . .

  ⇒ problems with incompatibility

- ♢ Purpose of Common Lisp: to unify the main dialects
  Thus it 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 TerpConnect account

You'll be using **Allegro Common Lisp**. Here is how to launch it:

```
tap allegro81
alisp
```

To avoid having to type `tap allegro81` every time you login, put it into the `.cshrc.mine` file in your home directory

---

Running Common Lisp elsewhere:

◇ Allegro Common Lisp is installed on some of the CS Dept computers
e.g., the junkfood 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

```
sporty:~: 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
sporty:~:
```

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:	`foo`	`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:

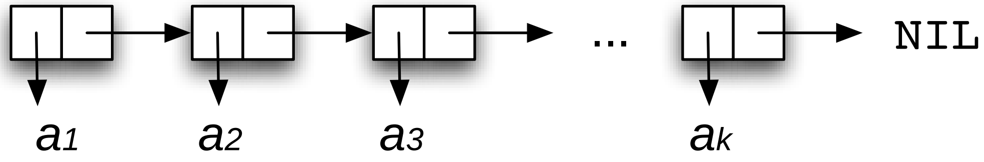
  `foo` = `FOO` = `Foo` = `FoO` = . . .

  `pi` = `Pi` = `PI` = `pI`

  `2e10` = `2E10`

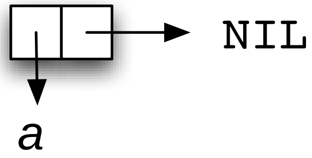
Lists

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



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

A list of one element: $(a) \Longrightarrow$



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

Examples:

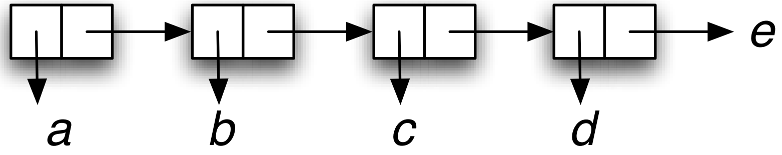
```
(235.4  (2e10  2/3)  "Hello, there!"  #(1 4.5 -7))

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

Dot notation

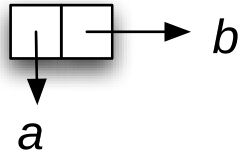
If the last pointer points to something other than `nil`, it's printed with a dot before it

$(a \;\; b \;\; c \;\; d \;\; . \;\; e) \Longrightarrow$



$(a \;\; b \;\; c \;\; d \;\; . \;\; \texttt{NIL}) \;=\; (a \;\; b \;\; c \;\; d)$

$(a \;\; . \;\; b) \Longrightarrow$



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)))))
```

This is a *very* bad code; its running time is exponential in n. My only purpose is to give an example that you can understand without knowing Lisp.

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

```
sporty:~: alisp
    ... several lines of printout ...
CL-USER(1): (load "fibonacci")
; Loading /homes/research/nau/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):
```

Compiling

- The code on the previous slide runs *interpretively* [*]
  Compiling makes your programs run faster, and may detect some errors
- To compile `fib` after it has been loaded, we can use `(compile 'fib)`
  Later I'll explain what the `'` is for
- That only compiles the code that's running in the current Lisp session.
  If you start up a new Lisp session and load `fibonacci.cl` again,
  it will run interpretively again.
- To compile the entire `fibonacci.cl` file, use `(compile-file "fibonacci")`
  This creates a binary file called `fibonacci.fasl`[**]

---

[*] A few Common Lisps will compile the code every time you load it. Allegro doesn't.

[**] Other Common Lisps may use different file-naming conventions.

Loading

◇ `(compile-file "fibonacci")` doesn't load the file
You need to do that separately

◇ The next time you do `(load "fibonacci")`, it will load `fibonacci.fasl` instead of `fibonacci.cl`

◇ In Allegro Common Lisp, `(load "fibonacci")` does the following:[*]
load `fibonacci.fasl` if it exists
else load `fibonacci.cl` if it exists
else load `fibonacci.lisp` if it exists
else error

◇ Use `(load "fibonacci.cl")` to specify the exact file,
`(load "foo/fibonacci")` to specify a file in a subdirectory,
etc.

---

[*] Details (e.g., file suffixes) may vary in other Common Lisps.

Style

- Read Norvig's tutorial on Lisp programming style
  There's a link on the class page.
- Examples of comments, variables, and indenting:

```
;;; 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))
```

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
- Parenthesis matching
- Automatic indentation
- Automatic color coding of different parts of the program

But if you don't already know emacs,
- Don't bother learning it just for this class
- Steep learning curve

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

Development Environments

If you use Eclipse, there are two Lisp plugins for it:

- ♢ Cusp
- ♢ Dandelion

I don't use Eclipse, so I don't know much about them

If you use Emacs, there are two macro packages you can use:

- ♢ The one that comes with Allegro Common Lisp
- ♢ SLIME

These can run Common Lisp in an Emacs buffer, and do various other things

The class home page has links to all of these

Lisp functions

Next, I'll summarize some basic Common Lisp functions

- ◇ I may leave out some details
- ◇ There are many more functions than the ones I'll discuss
- ◇ For more information, see the list of sources at the start of this lecture

Numeric functions

`+`, `*`, `/`	plus, times, divide	`(/ (* 2 3 4) (+ 3 1))` $\Longrightarrow$ 6
`-`	minus	`(- (- 3) 2)` $\Longrightarrow$ –5
`sqrt`	square root	`(sqrt 9)` $\Rightarrow$ 3
`exp`, `expt`	exponentiation	`(exp 2)` $\Longrightarrow e^2$, `(expt 3 4)` $\Longrightarrow$ 81
`log`	logarithm	`(log` x`)` $\Longrightarrow \ln x$, `(log` x b`)` $\Longrightarrow \log_b x$
`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 funcs (radians)	`(sin (/ pi 2))` $\Longrightarrow$ 1.0

Special Forms

- These are used for side-effects.
- Unlike functions, they don't necessarily evaluate all args

`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` but also works on arrays, structures, . . .	`(setf foo #(1 2 3 4))` ⟹ `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))` ⟹ `a`
`second`, …, `tenth`	like `first`	`(third '(a b c d))` ⟹ `c`
`rest`, `cdr`	all but 1st	`(rest '(a b c d))` ⟹ `(b c d)`
`nth`	nth element, n starts at 0	`(nth 2 '(a b c d))` ⟹ `c`
`length`	#of elements	`(length '((a b) c (d e)))` ⟹ `3`
`cons`	inverse of `car` & `cdr`	`(cons 'a '(b c d))` ⟹ `(a b c d)` `(cons '(a b) 'c)` ⟹ `((a b) . c)`
`list`	make a list	`(list (+ 2 3) '(b c) 'd 'e)` ⟹ `(5 (b c) d e)`
`append`	append lists	`(append '(a) '(b c) '(d))`⟹ `(a b c d)` `(append '(a) '(b c) 'd)`⟹ `(a b c . d)`
`reverse`	reverse a list	`(reverse '((a b) c d))` ⟹ `(d c (a b))`

Predicates

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

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

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

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

More special forms: conditionals

`if`	if-then-else	`(if` $test$ $expr_1$ `[`$expr_2$`])` if $test$ is non-`NIL` then return $expr_1$ else return $expr_2$ (or `NIL`)

`cond`	extended if-then-else	`(cond (`$test_1$ $expr_{11}$ $expr_{12}$ … `)`
		`(`$test_2$ $expr_{21}$ $expr_{22}$ … `)`
		… `)`

`case`	like C's "switch". The v_{ij} args aren't evaluated; `otherwise` is optional and is like C's `default`	`(case` x `((`v_{11} v_{12} …`)` $expr_{11}$ $expr_{12}$ …`)`
		`((`v_{21} v_{22} …`)` $expr_{21}$ $expr_{22}$ …`)`
		…
		`(otherwise` $expr1$ $expr_2$ …`)`

`ecase`	like `case`, but signals a *continuable* error if there's no match	`(ecase` x `((`v_{11} v_{12} …`)` $expr_{11}$ $expr_{12}$ …`)`
		`((`v_{21} v_{22} …`)` $expr_{21}$ $expr_{22}$ …`)`
		…`)`

Special forms for sequential execution

(`progn` $e_1\ e_2\ \ldots\ e_n$) evaluates $e_1, e_2, \ldots, e_n$, and returns the value of e_n

(`prog1` $e_1\ e_2\ \ldots\ e_n$) evaluates $e_1, e_2, \ldots, e_n$, and returns the value of e_1

`let` and `let*` are like `progn` but let you declare local variables

`let` assigns initial values
in parallel

```
(let (a b c)
   (setq a 1)
   (setq b 2)
   (setq b 3)
   (let ((a (+ b 5))
         (b (+ a 5)))
      (list a b c))
⟹ (7 6 3)
```

`let*` assigns initial values
sequentially

```
 (let* (($x_1$ $v_1$) (($x_2$ $v_2$) ($x_3$ $v_3$))
    $e_1$ $e_2$ ... $e_n$)
=
 (let (($x_1$ $v_1$))
    (let (($x_2$ $v_2$))
       (let (($x_3$ $v_3$))
          $e_1$ $e_2$ ... $e_n$)))
```

Formatted output

`(format` *destination control-string args*`)` is like `printf` in C

`(setq x "foo")`

`(format t "~%~s is ~s" 'x x)` ⟹ go to new line and print `X is "foo"`

`(format t "~%~a is ~a" 'x x)` ⟹ go to new line and print `X is foo`

destination is where to send the output

- name of stream ⇒ send it to the stream, then return `NIL`
- `t` ⇒ send to standard output, then return `NIL`
- `nil` ⇒ send output to a string, and return the string

control-string is like a `printf` control string in C

- **~** is like `% in` C
- `~%` is a newline like `\n` in C, `~2%` is 2 newlines, `~3%` is 3 newlines, etc.
- `~&` is like `~%` but is ignored if you're already at the start of a line
- `~s` matches any Lisp expression, and prints it with escape characters
- `~a` matches any Lisp expression, and prints it without escape characters
- `~2s` uses field size ≥ 2, `~3a` uses field size ≥ 3, etc.
- **many** more options – some useful, some you'll never use

Basic I/O

`(read [`*stream*`])`	read a single Lisp expression, and return it unevaluated	`(read)`
		`(+ foo 5)`
		$\Longrightarrow$ `(+ FOO 5)`

`(terpri [`*stream*`])`	is like `(format` *stream* `"~%")`	
`(prin1` *expr* `[`*stream*`])`	is like `(format` *stream* `"~s"` *expr*`)` but returns *expr*	
`(princ` *expr* `[`*stream*`])`	is like `(format` *stream* `"~a"` *expr*`)` but returns *expr*	
`(print` *expr* `[`*stream*`])`	is like `(format` *stream* `"~%~s"` *expr*`)` but returns *expr*	
`(pprint` *expr* `[`*stream*`])`	"pretty" `print` – does fancy indenting to improve readability, and returns no value	

The *stream* argument is optional

for `read`, it defaults to `*standard-input*`

for the other functions, it defaults to `*standard-output*`

Macros

Macros expand inline into other pieces of Lisp code

Example:

`push` and `pop` use lists as stacks

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

Various other built-in macros

e.g., see next page

Lisp also lets you define your own macros

It gets complicated

I won't discuss it

I/O Macros

(`with-open-file` (*stream filename* [*options*]) e_1 e_2 ... e_n)

(`with-input-from-string` (*stream string* [*options*]) e_1 e_2 ... e_n)

(`with-output-to-string` (*stream string* [*options*]) e_1 e_2 ... e_n)

Like (`progn` e_1 e_2 ... e_n), but binds *stream* to the file or string

```
(with-open-file (*standard-output* "foo.txt" :direction :output)
   (format t "2 + 3 = ~s" (+ 2 3))
   14)
```

$\Longrightarrow$ *creates file* `foo.txt`, *puts* `2 + 3 = 5` *into it, closes it, and returns* `14`

```
(with-input-from-string (*standard-input* "(+ 2 3)")
  (eval (read)))
```

$\Longrightarrow$ `5`

◇ *stream* is *dynamically* scoped:
its binding is used during execution of everything called by $e_1, \ldots, e_n$

◇ `with-open-file` closes *filename* automatically when finished

Operators

Lisp operator: a function, special form, or macro

Some differences among functions, special forms, and macros:

- ◇ Lisp evaluates all of a function's args before calling the function
  Not so for special forms and macros
- ◇ You can pass functions as arguments to other functions
  You can't pass special forms and macros (at least, not in the same way)
- ◇ If your code contains a Lisp macro, and if an error occurs while executing it, the debugging messages will probably refer to the code that the macro expanded into, rather than the macro itself

Loops

(`dotimes` (*i* *num* [*value*]) *expressions*)

executes *expressions* with $i = 0, \ldots, num - 1$, then returns *value* or `NIL`

(`dolist` (*x* *list* [*value*]) *expressions*)

executes *expressions* with $x =$ each element of *list*,
then returns *value* or `NIL`

(`return` *value*) returns *value* from the middle of a loop

```
(setq result nil)
(dotimes (foo 5 (reverse result))
    (push foo result))                          ⟹   (0 1 2 3 4)
```

```
(setq result nil)
(dolist (foo '(a 1 b 2 "stop here" 3 z 33))
    (if (stringp foo) (return result))
    (push foo result))                          ⟹   (2 B 1 A)
```

More loops

(`do` ((i_1 start_1 incr_1) . . . (i_n start_n incr_n))
    (*termination-test* [*expressions to evaluate at termination*])
  $\mathit{expression}_1$
  . . .
  $\mathit{expression}_n$)

Somewhat like C's "for", but the iteration variables are local, and are set simultaneously. To set them sequentially, replace `do` with `do*`

Unfortunately, the syntax is a bit painful

```
(setq c 0)
(do ((a 1 (+ a 1))                            ; a = 1, 2, 3, ...
     (b '(1 10 3 2) (cdr b)))                 ; take successive cdrs
    ((null b) c)                              ; if b is empty, return c
  (setq c (+ c (expt (car b) a))))            ; add x^a to c
```

$\Longrightarrow$ compute $1^1 + 10^2 + 3^3 + 2^4 = 144$

More loops

`(loop [`*loop clauses*`])`

iteration macro with a huge number of options

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

But simple cases are easier to understand than `do` is:

```
(loop for a from 1 by 1
      for b in '(1 10 3 2)
      sum (expt b a))
```

```
(setq c 0)
(do ((a 1 (+ a 1))
     (b '(1 10 3 2) (cdr b)))
    ((null b) c)
  (setq c (+ c (expt (car b) a))))
```

$\Longrightarrow$ compute $1^1 + 10^2 + 3^3 + 2^4 = 144$

More loops

`(loop [`*loop clauses*`])`

some of the possible loop clauses:

```
initially expressions    ; do these before looping starts
for variable from bottom to top
while condition
do expressions
if expression do expressions else expressions end
sum expression           ; add up all the values of expression
count expression         ; count how many times expression is non-NIL
collect expression       ; collect the values into a list
maximize expression      ; keep the highest value
minimize expression      ; keep the smallest value
return expressions       ; exit the loop and return this value
finally expressions      ; execute when the loop ends
```

For info and examples, see the links for `loop` on the class page

More loops

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

```
(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)))
```

⟹ `((A 1) (B 2) (C 3) (D 4))`

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

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

⟹ `(A 1 B 2 C 3 D 4)`

Write your own Lisp interpreter!

You can use `loop` or `do` to write your own simple Lisp interpreter:

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

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

Interacting with Allegro Common Lisp

- ♢ Allegro Common Lisp has a command-line interface
  Maybe also a GUI, depending on what OS you're using
  - check the documentation

- ♢ When it prompts you for input, you can type
  any Common Lisp expression or any Allegro command

- ♢ Allegro command syntax is `:command arg1 arg2 ...`
  - `: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**
  They won't work inside Lisp programs
  They're only available interactively, at Allegro's input prompt

- ♢ Which 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

- To get Allegro to print out *all* of a long list list (rather than just the first 10 elements), type `(setf tpl:*print-length* nil)`

- 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

```
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):
```

- 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
- `:reset` exits the debugger completely, back to the top level of Lisp
- Type `:help` for a list of other commands

Starting at Lisp's top level, do `(trace fib)`, then `(fib "foo")`

```
 0[1]: (FIB "foo")
Error: `"foo"' 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
(< "foo" 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 the value of *expr* is *func*,
then $(\texttt{funcall}\ \mathit{expr}\ e_1\ e_2\ \ldots\ e_n) = (\mathit{func}\ e_1\ e_2\ \ldots\ e_n)$

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

and $(\texttt{apply}\ \mathit{expr}\ (\texttt{list}\ e_1\ e_2\ \ldots\ e_n)) = (\mathit{func}\ e_1\ e_2\ \ldots\ e_n)$

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

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

◇ If *func* takes n args, you can do `(mapcar` $expr\ list_1\ list_2\ \ldots\ list_n$`)`
This takes *func*'s i'th arg from the i'th list

`(mapcar #'list '(a b c) '(1 2 3))` $\Longrightarrow$ `((A 1) (B 2) (C 3))`

◇ `mapcan` is like `mapcar` but *concatenates* the results (which must be lists)
`(mapcan #'list '(a b c) '(1 2 3))` $\Longrightarrow$ `(A 1 B 2 C 3)`

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

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

More functions

```
(coerce #(a b c) 'list)  ==> (a b c)
(coerce '(#\a #\b #\c) 'string)  ==> "abc"
(coerce 1 'float)  ==> 1.0

(member 3 '(1 2 3 4 5))  ==> (3 4 5)
(member 6 '(1 2 3 4 5))  ==> NIL
(member-if #'numberp '(a b 1 c d))  ==> (1 C D)
(member-if-not #'atom '(a b (c d) e))  ==> ((C D) E)

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

(`copy-list` *expr*) returns a new list whose elements are the ones in *expr*

(`copy-tree` *expr*) is like `copy-list`, but recursively copies all the way down

Keyword arguments

```
(member '(1 2) '((a 1) (b 2) (c 3)))                ==> NIL
(member '(1 2) '((a 1) (b 2) (c 3)) :test #'equal) ==> ((B 2) (C 3))
(member      b '((a 1) (b 2) (c 3)) :key #'first)  ==> ((B 2) (C 3))

(member '(B) '(((A) 1) ((B) 2) ((C) 3)) :key #'first :test #'equal)
 ==> (((B) 2) ((C) 3))

(subsetp '((a) b) '(x (a) y b z))  ==> NIL
(subsetp '((a) b) '(x (a) y b z) :test #'equal)  ==> T
```

In a list of the form `(` ... x ... `)`,

◇ `:key` f changes what part of x you apply the test to
instead of `(eql` z x`)`, use `(eql` z `(`f x`))`

◇ `:test` p and `:test-not` p change what the test function is
instead of `(eql` z x`)`, use `(`p z x`)` or `(not (`p z x`))`

`:test`, `:test-not`, and `:key` can be used in almost any built-in function in which they would have a sensible meaning

Defining functions with optional arguments

This function takes two positional arguments and one keyword argument:

```
(defun my-member (item list &key (test #'eql))
  (cond ((null list) nil)
        ((funcall test item (car list)) list)
        (t (my-member item (cdr list) :test test))))
```

This function requires at least one argument:

```
(defun tformat (control-string &rest args)
      (apply #'format t control-string args))))
```

This function takes any number of arguments:

```
(defun count-args (&rest args)
  (length args))
```

Functions of sequences

Some functions work on any *sequence* (list, character string, or vector)
In these functions, sequences are indexed starting at 0

`(elt` *seq n*`)` returns the n'th element of *seq*

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

`(subseq` *seq num1* `[`*num2*`])` returns the subsequence that starts at *num1* and ends just before *num2*

```
(subseq '(a b c d e f) 2 4) ==> (C D)
(subseq #(a b c d e f) 2) ==> #(C D E F)
(subseq "abcdef" 2 5) ==> "cde"
```

`(copy-seq` *seq*`)` returns a copy of *seq*

```
(setq a "abc"))
(equal a (copy-seq a)) ==> T
(eq a (copy-seq a)) ==> F
```

More functions of sequences

`(find` *item seq*`)`	find *item* in *seq* and return it, else return `nil`
`(position` *item seq*`)`	return *item*'s position in *seq*, else return `nil`
`(remove` *item seq*`)`	remove top-level occurrences of *item*
`(substitute` *new old seq*`)`	replace top-level occurrences of *old* with *new*

Optional keyword arguments (you can use several of them at once):

`:key` *key*	use `(eql (`*key item*`)` *x*`))` instead of `(eql` *item x*`)`
`:test-if` *pred*	use `(`*pred item x*`)` instead of `(eql` *item x*`)`
`:test-if-not` *pred*	use `(not (`*pred item x*`))` instead of `(eql` *item x*`)`
`:from-end t`	search leftward rather than rightward
`:start` *num*	start searching at position *num* (instead of position 0)
`:end` *num*	end searching just before position *num*
`:count` *num*	in `remove` and `substitute`, only change *num* occurrences of *item*, rather than all of them

Examples

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

```
(position #\d "abcde") ==> 3
(position #\d #(#\a #\b #\c #\d #\e) ==> 3
```

```
(remove 'a '((a 1) (a 2) (a 3) (a 4)) :key #'car) ==> NIL
```

```
(remove 'a '((a 1) (a 2) (a 3) (a 4)) :key #'car :start 1)
    ==> ((A 1))
```

```
(remove 'a '((a 1) (a 2) (a 3) (a 4)) :key #'car :start 1 :end 3)
   ==> ((A 1) (A 4))
```

More functions of sequences

With these functions, you can use the same keyword arguments as before except for `:test-if` and `:test-if-not`

`(find-if` *pred seq*`)`

`(find-if-not` *pred seq*`)`

find item that satisfies *pred*

`(position-if` *pred seq*`)`

`(position-if-not` *pred seq*`)`

find position of item that satisfies *pred*

`(remove-if` *pred seq*`)`

`(remove-if-not` *pred seq*`)`

remove items that satisfy *pred*

`(substitute-if` *new pred seq*`)`

`(substitute-if` *new pred seq*`)`

substitute *new* for items that satisfy *pred*

Examples

```
(defun almost-equal (Num1 Num2)
   (<= (abs (- Num1 Num2)) 0.1))

(defun almost-pi (Num)
   (almost-equal num pi))

(find pi #(2.9 3.0 3.1 3.2 3.3) :test #'Almost-Equal) ==> 3.1

(find-if  #'almost-pi #(2.9 3.0 3.1 3.2 3.3)) ==> 3.1
```

Tree functions

`copy-tree`

like `copy-list` but copies an entire tree structure

`subst`, `subst-if`, and `subst-if-not`

like `substitute`, `substitute-if`, and `substitute-if-not`,
but they look through the entire tree structure

```
(substitute 'tempest 'hurricane
        '(shakespeare wrote (the hurricane)))
   ==> (SHAKESPEARE WROTE (THE HURRICANE))

(subst 'tempest 'hurricane
        '(shakespeare wrote (the hurricane)))
   ==> (SHAKESPEARE WROTE (THE TEMPEST))
```

`subst` recognizes the keyword arguments `:test`, `:test-not`, and `:key`

`subst-if` and `subst-if-not` recognize `:key` but not the others

Destructive versus nondestructive functions

- ♢ The functions on the previous pages are *nondestructive*
  They don't modify their arguments

- ♢ There are destructive versions of the same functions
  Rather than making copies, these redirect pointers (like you'd do in C)
  - `delete` is the destructive version of `remove`,
  - `nconc` is the destructive version of `append`,
  - `nreverse` is the destructive version of `reverse`, etc.

  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)
```

Don't do destructive modifications unless (i) there's a very good reason to do them and (ii) you're very sure you know what you're doing

Defstruct

```
(defstruct employee name id dept phone)

(setq x (make-employee :name "Dana Nau"))
    ==> #S(EMPLOYEE :NAME "Dana Nau" :ID NIL :DEPT NIL :PHONE NIL)

(setf (employee-dept x) "Computer Science")

x  ==>  #S(EMPLOYEE :NAME "Dana Nau"
                    :ID NIL
                    :DEPT "Computer Science"
                    :PHONE NIL)
```

Many more options (see the book)

default initial values, inheritance, print functions, . . .

For object-oriented programming, use `defclass` rather than `defstruct`

Lisp has a huge set of features

Many more features that I didn't cover. Here are a few of them:

- ◇ `random` - return a random number in a given range
- ◇ `make-hash-table` - return a hash table
- ◇ `cerror` - continuable error: error message with options for fixing the error
- ◇ `values`, `multiple-value-setq` - functions that return multiple values
- ◇ `return-from`, `catch`, `throw`, `unwind-protect` - non-local returns
- ◇ packages - separate namespaces to avoid naming conflicts
- ◇ object-oriented programming
- ◇ how to write macros

. . .

Seven ways to copy a list

(Adapted from the Lisp FAQ; link on the class page)

Let's define a function `cc-list` that does the same thing as `copy-list`

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

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

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

2nd implementation goes through the list twice: first to build up the list in reverse order, and then to reverse it. It has linear running time.

Seven ways to copy a list (continued)

```
3.  (defun cc-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)))))

4.  (defun cc-list (list)
      (mapcar #'identity list))

5. (defun cc-list (list)
     (loop for x in list
        collect x))
```

3rd, 4th, and 5th implementations: efficiency usually similar to the 2nd one, depending on the Lisp implementation.

The 4th and 5th implementations are the easiest to understand.

Seven ways to copy a list (continued)

```
6.  (defun cc-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))))))
```

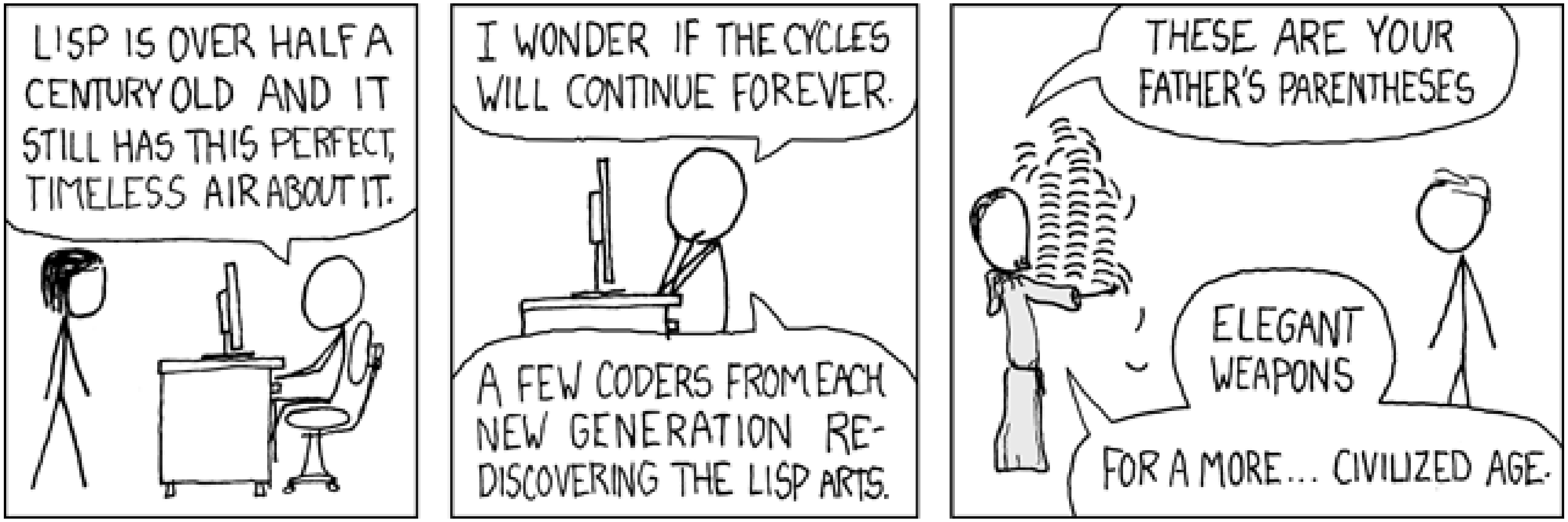
6th implementation iterates down the list only once, keeps a pointer to the tail of the list, destructively modifies the tail to point to the next element. Same speed as 2nd, 3rd, 4th, 5th implementations, or slightly slower.

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

7th implementation: recursively copies dotted lists, and runs in linear time, but isn't tail-recursive ⇒ compiler can't remove the recursion

Conclusion

From http://xkcd.com/297



If you don't understand this cartoon, go to the following URL and search for "a more civilized age":
http://www.imdb.com/title/tt0076759/quotes