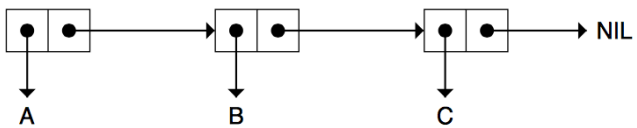
CS3101 Programming Languages (Lisp)

Lecture 1

Bob Coyne (coyne@columbia.edu)



Columbia University

March 10, 2017

Course Information

- When: 7 Weeks (10:10am-Noon on Fridays)
- Where: 417 Mathematics
- Who (instructor): Bob Coyne (coyne@cs.columbia.edu)
- Who (TA): Ben Kuykendall (brk2117@columbia.edu)
- Why: Lisp is fun (and powerful)!

Introduction to Lisp – Overview

- Class participation: 10%
- 5 homeworks (each due before the next class): 50%
  These will be small programming exercises to reinforce what's covered in class
- Project proposal: 10%
- Final project: 30% can be individuals or teams (up to 3 people)

Approximate Syllabus

March 10	**Background**: history, installing, resources. **Basics**: symbols, evaluation, data types, lists, conditionals, functions, lambda forms, Emacs, REPL, ...
March 24	**More basics**: Backquote, vectors, sequences, file system, loop, format, packages, streams, debugger, compiling, ...
March 31	**Macros etc**: Macros, closures, reader macros, Error system, performance tuning
April 7	**Objects etc**: Type system, CLOS, Structs, FFI, OS hooks...
April 14	**External libraries**: Quicklisp, ASDF, cl-json, cl-html, cl-ppcre, drakma, ...
April 21	**AI**: Prolog in Lisp, knowledge representation, constraints, unification
April 28	**Lisp variants/offshoots** (Elisp, Clojure, Scheme, Mathematica, Parenscript)

Online Books and Language References

- Practical Common Lisp (Peter Seibel) – [excellent modern intro to Lisp]
`http://www.gigamonkeys.com/book/`
- Lisp Hyperspec – [very useful, nicely indexed language reference]
`http://www.lispworks.com/documentation/HyperSpec/Front/index.htm`
- Lisp Recipes (Edi Weitz) - [lots of practical info...access within Columbia network]
`http://link.springer.com/book/10.1007%2F978-1-4842-1176-2`
- On Lisp (Paul Graham) – [semi-advanced, but excellent]
`http://www.paulgraham.com/onlisp.html`
- Common Lisp – A Gentle Introduction to Symbolic Computation (Touretzky)
`https://www.cs.cmu.edu/~dst/LispBook/book.pdf`
- Common Lisp the Language (Guy Steele) – [The de-facto language specification]
`https://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html`
- Let over Lambda (Doug Hoyte) – [advanced on closures, etc]
`http://letoverlambda.com/`

Useful websites and online resources

- CLiki
`http://cliki.net`
- Planet Lisp (a Lisp "meta" blog)
`http://planet.lisp.org`
- Quicklisp
`https://www.quicklisp.org/beta/UNOFFICIAL/docs/`
- Common Lisp Cookbook
`http://cl-cookbook.sourceforge.net/index.html`
- Lisp tutorials
`http://lisp.plasticki.com/show?36`
- Peter Norvig on Python for Lisp programmers
`http://norvig.com/python-lisp.html`

Lisp Implementations

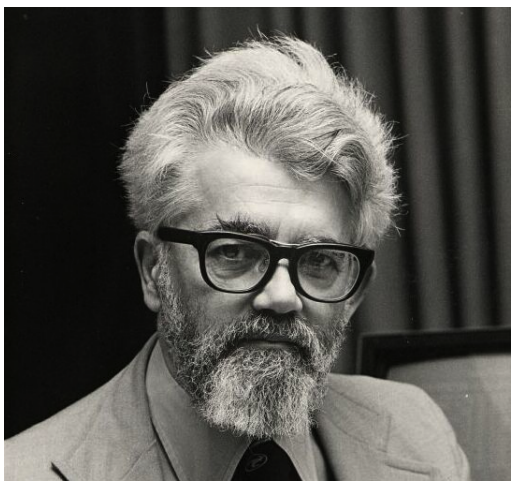
In class, I'll be using SBCL and Lispworks

- SBCL: `http://www.sbcl.org/platform-table.html`
- Lispworks: `http://www.lispworks.com/downloads/`
- Allegro CL: `http://franz.com/downloads/clp/survey`
- Others: ABCL, CMUCL, CLISP, OpenMCL, ECL, SCL

`https://common-lisp.net/~dlw/LispSurvey.html`

John McCarthy

John McCarthy was one of the founders of the discipline of artificial intelligence. He coined the term "artificial intelligence" (AI), developed the Lisp programming language family, significantly influenced the design of the ALGOL programming language, popularized timesharing, and was very influential in the early development of AI. McCarthy received many accolades and honors, such as the Turing Award for his contributions to the topic of AI, the United States National Medal of Science, and the Kyoto Prize. (from wikipedia)

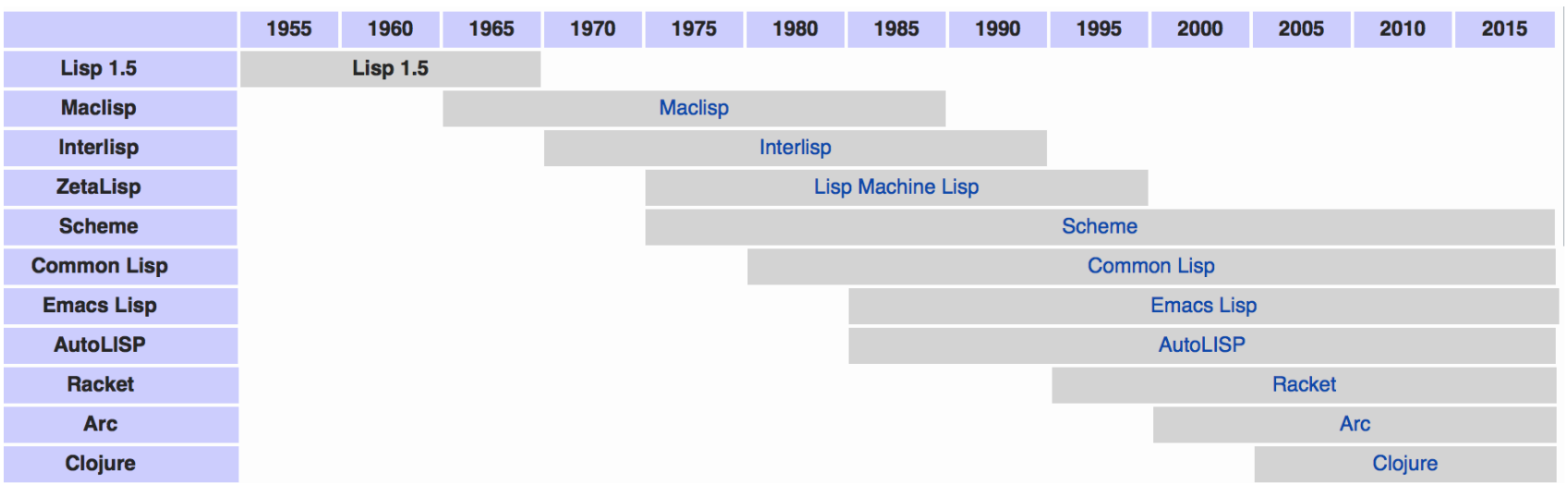


Lisp history and dialects

- Lisp (LISt Processing) is the second oldest language (1958) still in common use. Fortran was created one year earlier. Thanks to its simple syntax (lists) and macros (to transform those lists), Lisp has been called a "programmable programming language."

- Highly influential: Introduced if-then-else construct; garbage collection; read-eval-print loop; dynamic typing; incremental compilation; homoiconicity and meta-programming, closures (via Scheme)...

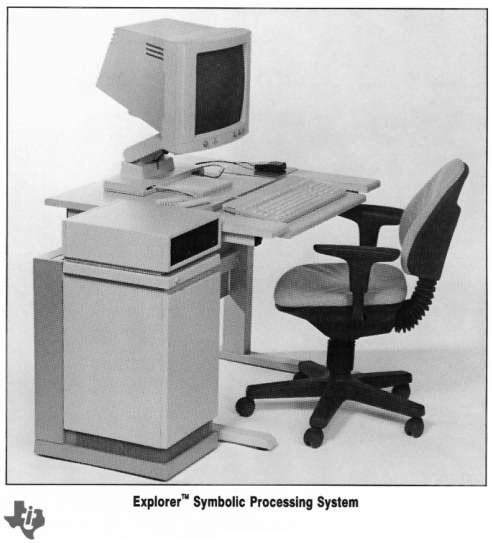
- Multi-paradigm: functional, procedural, reflective, meta



Lisp Machines – MIT AI Lab (circa 1975)

Commercial Lisp Machines circa 1980-1990

TI Explorer



LMI Lambda

Symbolics

Symbolics Keyboard

Intro and history

Lists, s-expressions, and cons cells

Data types

Evaluation

Misc operations

Variables and value binding

Flow of control

Defining functions

Evaluation and the Read

Emacs

Lists as programs and data – key idea of Lisp

Arithmetic expressions:

```
(+ (* 3 4) (* 4 5))
```

Conditional expressions:

```
(when (> x 4) (print "bigger than 4"))
```

Flat and nested lists

```
("columbia" "yale" "dartmouth" "penn" "cornell")
(0 (1 2 3) (3 4 5) (6 (7 8)))
```

Association lists and property lists

```
(("new jersey" :capital "trenton")
 ("alabama" :capital "montgomery")
 ("new york" :capital "albany") ...)
```

Variable assignment:

```
(setq x '((1 "one") (2 "two") (3 "three")))
```

Function definition:

```
(defun cube (x) (* x x x))
```

Minimal syntax

No special syntax for operators and statements – just functional application to arguments in lists

(FUNCTION arg0 arg1 arg3 ....)

For example:

(+ 2 3 4 5) ⇒ 14

Including nested:

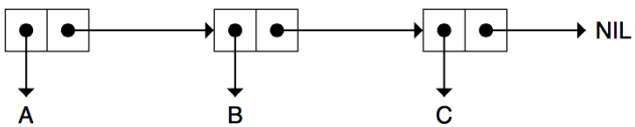
(+ 2 (* 3 4) 5) ⇒ 19

S-Expressions

Lisp programs and data consist of **s-expressions** (Symbolic expressions)

An s-expression can be an **atom** or **list** (including nested lists)

- An atom can be a symbol, string, array, structure, number...
- A list is sequence of **cons cells** linking s-expressions together



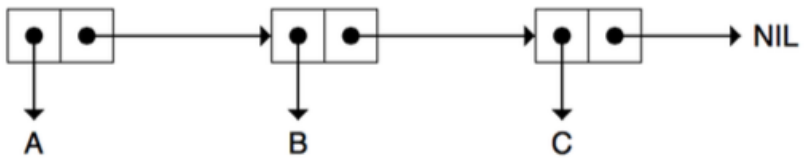
Examples

- (aaa 12 foo 44.0) is a flat list
- (nil "hello" (99.0 1/2) foo) is a nested list
- nil, "hello", 99.0, 1/2, and foo are all atoms.

Lisp symbols are generally made upper case when they are **read**
(print (list 1 2 3)) automatically becomes (PRINT (LIST 1 2 3))

Cons Cells – used to represent lists

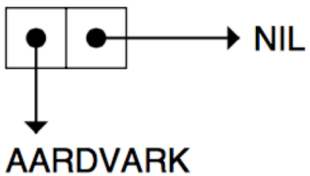
(a b c)



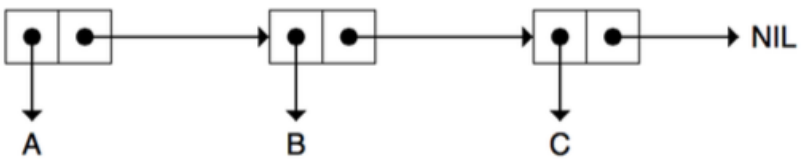
- A cons cell is a pair of pointers – First part is the **car** and second part is the **cdr**
- The car and cdr point to any s-expression
- For a **proper list**, the CDR of last CONS cell points to **nil**
- Can represent: flat lists, nested lists (trees), even circular lists and other structures.
- Dots omitted when cdr is a list: '(a . (b . (c . nil))) ⇒ (a b c)

Cons Cells – used to represent lists

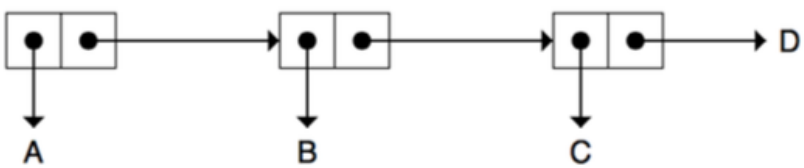
(aardvark . nil) or (aardvark)



(a b c)



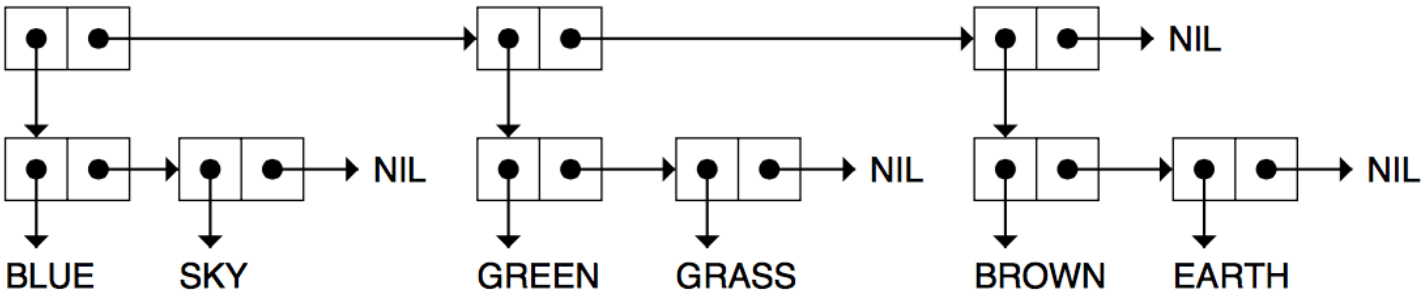
(a b c . d)



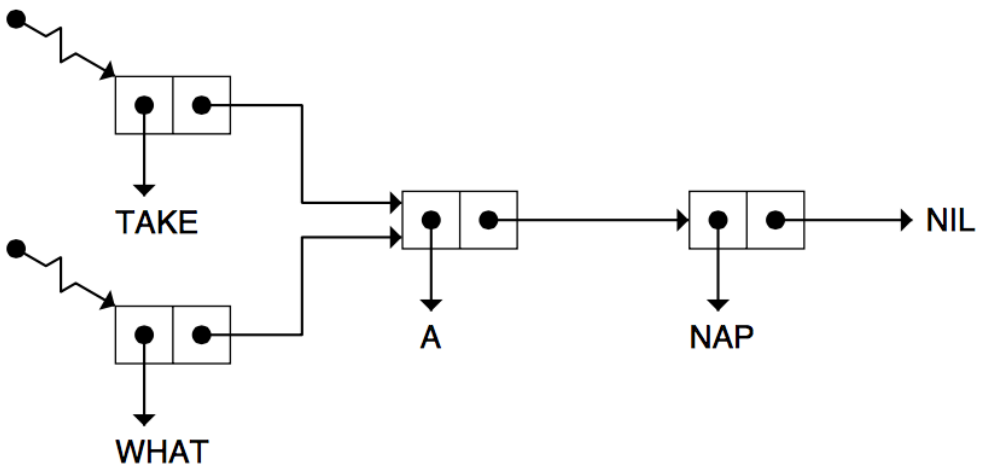
Diagrams from `https://www.cs.cmu.edu/~dst/LispBook/book.pdf`

Cons Cells for NESTED or SHARED lists

((BLUE SKY) (GREEN GRASS) (BROWN EARTH))

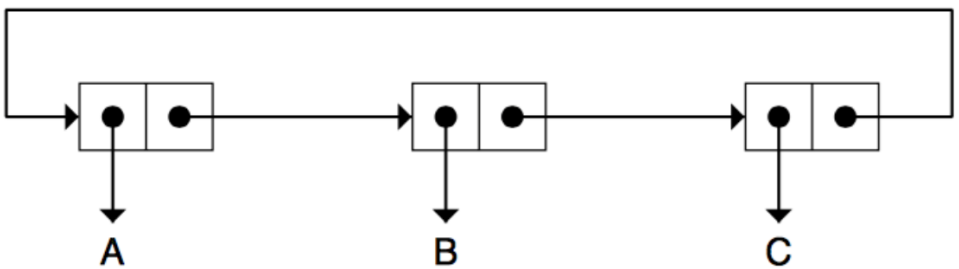


Two lists (TAKE A NAP) and (WHAT A NAP) sharing same (A NAP)

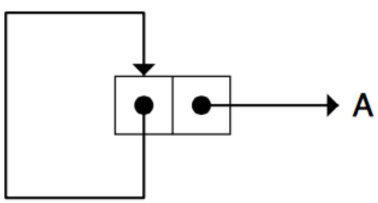


Circular Lists are possible

(setq cl '(a b c)) ⇒ (a b c)
(setf (cdddr cl) cl) ⇒ (a b c a b c a b c a b c a b c a b c a b c a b c a ...)
Print length controlled by *PRINT-LENGTH*

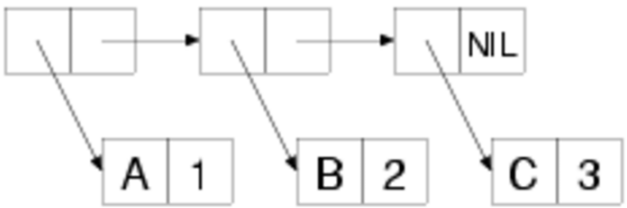


(setq cl '(a . a)) ⇒ (a . a)
(setf (car cl) cl) ⇒ ((((((# . a) . a) . a) . a) . a) . a)
Print depth controlled by *PRINT-LEVEL*



Alists – association lists

Alists provide a flexible lightweight way to store and retrieve data associations



((a . 1) (b . 2) (c . 3))

```
(defparameter *elephant-alist*
   '((:height . 10)
     (:color . gray)
     (:mammal? . t)
     (:locations . (africa asia))))

(cdr (assoc :color *elephant-alist*))
   => gray
```

Plists – property lists

Plists provide another flexible lightweight way to store and retrieve data associations

(a 1 b 2 c 3)

```
(defparameter *elephant-plist*
   '(:height 10
     :color  gray
     :mammal? t
     :locations (africa asia)))

(getf *elephant-plist* :locations)
  => (AFRICA ASIA)
```

Intro and history

Lists, s-expressions, and cons cells

Data types

Evaluation

Misc operations

Variables and value binding

Flow of control

Defining functions

Evaluation and the Read-Eval-Print-Loop (REPL)

Emacs

Basic data types – each with their own test predicate

Symbols: T, NIL, foo-1, system-error, *database*, foo.bar, sb-unix:unix-fstat
(symbolp 'foo-1) → T

Numbers: 7, 33.33, #C(0.0 1.0), 34324234234234234
(numberp 33.0) → T

Strings: "artificial intelligence", "Lisp", "Barack Obama"
(stringp 'foo) → NIL

Characters: #\a, #\space
(characterp #\NEWLINE) → T

Lists: ((1 2 3) 4 5 (6 7 (8 9)))
NIL is the empty list (), as well as being a symbol representing FALSE
(listp '(a d)) → T

More: vectors, structs, hash tables, arrays, CLOS objects, ...
(vectorp 343) → nil ... etc

Symbols

Symbols have an associated name, value, function, property list.

Value: (setq foo '(a b c d))
Function: (defun foo (x) (* x x))
Propertes: (setf (get 'foo :size) 22) (setf (get 'foo :length) 44)

Symbol-name, **symbol-package**, **symbol-value**, **symbol-function**, **symbol-plist** (or **get**) can retrieve the above. E.g. (symbol-name 'foo) → "FOO"

```
(describe 'foo)
FOO is a SYMBOL
NAME          "FOO"
VALUE         (A B C D)
FUNCTION      #<Function FOO 2009E892>
PLIST         (:LENGTH 44 :SIZE 22)
PACKAGE       #<The WORDSEYE package, 3462/4096 internal, 180/256 external>
```

Note: T, NIL, and keyword symbols (e.g. :FOO) evaluate to themselves

What is NIL?

NIL represents both FALSE and the empty list ().

```
;; it's an atom
(atom nil) => T

;; it's a symbol
(symbolp nil) => T

;; it's also a list
(listp nil) => T

;; it's not a cons (unlike other lists)
(consp nil) => NIL

;; it evaluates to itself
nil => NIL

;; it's a constant (value can't be changed)
(constantp nil) => T
```

Symbols

E.g. *database*, house, add, my-name, T, NIL

Symbols exist in a package (namespace)

- foobar ⇒ in current package (*package*)
- cl-user is default user package
- math:matrix ⇒ external in the MATH package
- math::matrix ⇒ internal in the MATH package
- :depth ⇒ special **keyword** package
- #:G1067 ⇒ an uninterned symbol

e.g. *package* → #<package "common-lisp-user">

Function names are symbols too (e.g. append, +, print etc)

Numeric Types

Type	Example value	Example function
Integer	7	(+ 2 5)
Single-float	0.33333334	(/ 1 3.0)
Double-float	0.3333333333333333D0	(/ 1 3.0d0)
Rational	1/3	(/ 3 9)
Complex	#C(0.0 1.0)	(sqrt -1.0)
Bignum	321466960818222453181624576	(expt 4234324 4)

Ways of checking types

(type-of 3.0) → single-float
(typep 3.0 'number) → T
(typep 3.0 'float) → T
(integerp 3.0) → NIL
(numberp 3.0) → T

Intro and history

Lists, s-expressions, and cons cells

Data types

Evaluation

Misc operations

Variables and value binding

Flow of control

Defining functions

Evaluation and the Read-Eval-Print-Loop (REPL)

Emacs

Evaluating s-expressions

Numbers evaluate to themselves
  34234 ⇒ 34234

T, NIL, and keyword symbols evaluate to themselves
  NIL ⇒ NIL, T ⇒ T, :foo ⇒ :foo

Quoted symbols or lists are not evaluated
  '(+ 23 11) ⇒ (+ 23 11)
  'foo ⇒ foo

Lists evaluate by applying first element to the rest
  (+ 23 11) ⇒ 34

Symbols evaluate to their values
  (setq aaa '(1 2 3 4))
    aaa ⇒ (1 2 3 4)
  (eval '(+ 23 11)) ⇒ 34
  bbb ⇒ Error: The variable BBB is unbound. [debugger]

Evaluating lists

Lists are evaluated by applying first element (a **function**, **special operator**, or **macro**) to the rest (the arguments) in order to return a value or values.

- **Functions** evaluate all their arguments.
- **Special operators** can choose which arguments to evaluate
- **Macros** transform the whole form to another, controlling what is evaluated.
- **Single quote** (eg '(a b c)) prevents an argument from being evaluated.

Single quote is shorthand for the special operator QUOTE

Examples

(+ 23 11) $\Rightarrow$ 34
(quote hello) $\Rightarrow$ hello
(append '(a b) '(c d e)) $\Rightarrow$ (a b c d e)

Note: multiple values can be returned:
(floor 3.4 4) $\rightarrow$ 3 .6

Special Operators

A **special form** is a form with special syntax, special evaluation rules, or both, possibly manipulating the evaluation environment, control flow, or both. A **special operator** has access to the current lexical environment and the current dynamic environment. Each special operator defines the manner in which its subexpressions are treated – which are forms, which are special syntax, etc. Lisp contains 25 special operators:

```
block       let*                  return-from
catch       load-time-value       setq
eval-when   locally               symbol-macrolet
flet        macrolet              tagbody
function    multiple-value-call   the
go          multiple-value-prog1  throw
if          progn                 unwind-protect
labels      progv
let         quote
```

Special Operators – Examples

(**quote** (a b c))
→ return argument without evaluating it. Same as '(a b c)

(**if** (< x 5) (print "big") (print "small"))
→ Evaluate first term and then, depending on value, evaluates (and returns) ONE of remaining arguments

(**progn** (setq x 33) (print (+ x 4)))
→ Evaluate all its arguments in order, returning final value

(**setq** seconds-in-day (* 24 60 60))
→ Assigns a value to a variable. The value arg is evaluated, but not the variable

(**let** ((i 10) (j 10))
  (print "adding 2 and 3")
  (print (+ i j)))
→ Binds lexical variables and then evaluates all remaining arguments (the body) in order and returns final value

Macros

Macros allow arbitrary evaluation of their arguments. A macro works by transforming the whole form (operator and arguments) into a new form. This new macroexpanded form is substituted into the calling code in place of the original form. In creating the new form, it can arbitrarily evaluate and transform its arguments.

For example, DEFUN is defined by Lisp itself as a built-in macro that associates a function definition with a name.

Another of a built-in macro is INCF, which increments a variable: (macroexpand '(incf x)) $\rightarrow$

```
(LET* ((#:G1057 1)
       (#:NEW1056 (+ X #:G1057)))
  (SETQ X #:NEW1056))
```

Intro and history

Lists, s-expressions, and cons cells

Data types

Evaluation

Misc functions

Variables and value binding

Flow of control

Defining functions

Evaluation and the Read-Eval-Print-Loop (REPL)

Emacs

Equality

EQ (identity), **EQL** (like EQ, but for numbers and characters), **EQUAL** (like EQL but for equivalent lists), **EQUALP** (like EQUAL but ignoring case), = (for numbers only)

(setq foo '(c d)) → (c d)
(setq bar '(c d)) → (c d)
(setq baz foo) → (c d)
(setq bar2 (cons 33 foo)) → (33 c d)
(setq baz2 'foo) → foo

(eq foo bar) → NIL	Different objects
(equal foo bar) → T	Equivalent lists
(eq foo baz) → T	Same object
(equal foo baz) → T	Same object, no need to check element equality
(eql (car bar2) 33) → T	Equivalent numbers are EQL
(eq (car bar2) 33) → ??	Probably T, but don't count on it!
(equal "foo" "FOO") → NIL	EQUAL is case sensitive
(equalp "foo" "FOO") → T	EQUALP is case insensitive
(equal 'foo "FOO") → NIL	FOO is symbol. "FOO" is string.
(equal 33 (car bar2)) → T	Equivalent numbers
(= pi foo) → ERROR	= requires numbers, else signals an error
(eq baz2 foo) → NIL	FOO points to a list, baz2 points to a symbol
(eq (cdr bar2) foo) → T	Same object

Some list operations

(**cons** 3 '(4 5 6)) → (3 4 5 6)
(**car** '(4 5 6)) → 4
(**cdr** '(4 5 6)) → (5 6) ; same as **rest**
(**nth** 2 '(a b c d e f g)) → c ; also can use **first, second, third, etc**
(**nthcdr** 2 '(a b c d e f g)) → (c d e f g)
(**length** '(a b c d e f g)) → 7
(**reverse** '(a b c d e f g)) → (g f e d c b a)
(**position** 'c '(a b c d e f g)) → 2

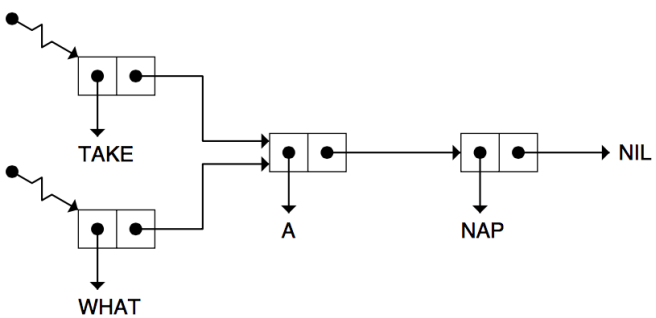
(**append** '(a b c) '(d e) '(1 2)) → (a b c d e 1 2)
(**list** 1 2 3 (a b)) → (1 2 3 (a b))

(setq *list* '(1 2 3))
(**push** :foo *list*)
  list → (:foo 1 2 3)
(**pop** *list*)
  list → (1 2 3)

Some **destructive** list operations

;;; **nreverse** is like **reverse** but modifies the lists to avoid copying.
(setq *list* '(a b c d e f g))
(**nreverse** *list*) → (g f e d c b a)
  list → (g f e d c b a)

;;; **nconc** is like **append** but modifies some of the lists to avoid copying.
(setq *lst1* '(a nap) *lst2* '(take) *lst3* '(what))
(**nconc** *lst2* *lst1*) → (take a nap)
(**nconc** *lst3* *lst1*) → (what a nap)
  lst2 → (take a nap)
  lst3 → (what a nap)
  (eq (cdr *lst2*) (cdr *lst3*)) → T



Some numeric operations

Many of these take arbitrary number of arguments. Some return multiple values.

(+ 2 1 1 1) → 5
(* 2 3) → 6
(/ 2 3) → 2/3
(/ 2 3.0) → 0.6666667

(< 2 2) → NIL
(> 2 -4 -9) → T
(> 2 'two) → ERROR
(<= 2 2) → T
(>= 2 3.0) → NIL

(float 2) → 2.0
(ceiling 2.2) → 3 -0.8
(floor 2.2) → 2 0.2
(rem 11 3) → 2
(mod 11 3) → 2

(min 3 -1 88) → -1
(max 3 -1 88) → 88

Logical connectives

AND and OR can "short-circuit" evaluation when current condition determines result.

(**and** form1 form2 form3 ...) → Tests if all forms are not NIL. Returns last value or NIL
(**or** form1 form2 form3 ...) → Tests if any form is not NIL. Returns that value or NIL
(**not** form) → returns T if form's value is NIL, else returns NIL

```
(defparameter *max*  99)
(defparameter *min*  0)
(defvar *current*  22)
(if (and (numberp *current*)
         (<= *current* min)
         (>= *current* min))
    (print :ok)
    (print :bad-or-out-of-range))
==> :OK
```

Printing and formatting strings

(**format** destination control-string arguments*)
destination: T for standard output, NIL to produce a string, else a STREAM object
control-string: See reference for details. Here are the most basic format directives:

```
~a for lisp object (from arguments*).
~s like ~a but strings get printed in quotes.
~% outputs a NEWLINE
  * (format t "Today is ~a" "sunday") prints: Today is sunday
  * (format t "Today is ~s" "sunday") prints: Today is "sunday"
```

(**print** string &optional stream)
  * prints string to *standard-output* or a STREAM object (optional argument).
princ is like print but omits surrounding quotes.
  * (print "hello") prints: "hello"
  * (princ "hello") prints: hello

Multiple outputs to the same string:

  * (**with-output-to-string** (s) (princ "2+3 is " s) (princ (+ 2 3) s)) → "2+3 is 5"

Intro and history

Lists, s-expressions, and cons cells

Data types

Evaluation

Misc operations

Variables and values

Flow of control

Defining functions

Evaluation and the Read-Eval-Print-Loop (REPL)

Emacs

Variable types

Lexical variables – When a lexical variable is bound, its value can be accessed by name only within a certain textual (lexical) block. Its binding is said to have lexical scope. Function parameter variables and variables bound by LET (unless declared special) are lexically scoped and cannot be referenced by name from the outside.

Special variables (aka dynamic variables) have indefinite scope and can be accessed from anywhere. A special variable value must be declared as such.

Global variables are special variables that have a top-level value.

```
(defvar *something*) ;; declared special but globally unbound
(defparameter *current-year* 2017) ;; declared special and given a global value
```

Constants (and NIL, T, and keyword symbols) have indefinite scope but fixed value

```
(defconstant +days-per-week+ 7) ;; by convention, constants use plus signs
(let ((+days-per-week+ 8))
  (print +days-per-week+))
Error: Cannot bind +DAYS-PER-WEEK+ -- it is a constant.
```

Lexical variable bindings

```
(progn
  (let ((x 3))
     (format t "value of X is ~a ~%" x)
     (let ((x 5))
       (format t "value of X is ~a ~%" x))
     (format t "value of X is ~a ~%" x))
  (format t "value of X is ~a ~%" x))
value of X is 3
value of X is 5
value of X is 3
```

Error: The variable X is unbound.

...Because the last reference to X is outside the lexical scope of the LET

Special variable bindings and dynamic extent

```
;;; By convention, special variables use asterisks.
(defvar *animal-name* "donkey")
(defun print-animal-name ()
  (format t "*animal-name* bound to ~s ~%" *animal-name*))

(progn
  (print-animal-name)
  (let ((*animal-name* "horse"))
     (print-animal-name)))
*animal-name* bound to: "donkey"
*animal-name* bound to: "horse"
```

Dynamic variable binding experiment

```
(defvar *yy* 3)
(defun print-yy-value () (print *yy*))
(defun set-yy () (setq *yy* :blah) (print *yy*))
(defun bind-set-yy () (let (*yy*) (setq *yy* :bloo) (print *yy*)))

(let ((*yy* 44))
    (setq *yy* 22)
    (print-yy-value)  => 22
    (set-yy)          => :blah
    (print-yy-value)  => :blah (value changed within dynamic extent)
    (bind-set-yy)     => :bloo (change value within a new dynamic extent)
    (print-yy-value)  => :blah (Outer value NOT CHANGED)
    )
```

Intro and history

Lists, s-expressions, and cons cells

Data types

Evaluation

Misc operations

Variables and value binding

Flow of control

Defining functions

Evaluation and the Read-Eval-Print-Loop (REPL)

Emacs

Basic Flow of Control

PROGN, IF, COND, CASE, ECASE, WHEN, UNLESS: All return a value and can be nested.

(**progn** form*)
→ executes all forms unconditionally. Returns value of last

(**cond** ((test form*) (test form*) (test form*) (t form*) ...))
→ T is optional catchall

(**if** test then-form else-form)

(**when** test form*)

(**unless** test form*)

(**case** test-var (val form*) (val form*) (t val-n))
→ T is optional catchall

(**case** test-var ((val*) form*) (val* form*) (t form*))
→ can test for list membership

ecase like case but signals error if no match

Iteration: do, dotimes, dolist, loop

```
(DOTIMES (x 10)
  (print x))

(LOOP for x below 10
   do (print x))

(DO ((x 0 (+ x 1)))
    ((= x 10))
  (print x))
```

```
(DOLIST (x '(a b c))
  (print x))

(LOOP for x in '(a b c)
   do (print x))

(let ((lst '(a b c)))
  (DO ((x (car lst) (car lst)))
      ((not lst))
    (setq lst (cdr lst))
    (print x)))
```

Iteration

```
(DO ((x 1 (+ x 1))  ; initial bindings and successive assignments
     (y 1 (* y 2)))
    ((> x 5) y)         ; termination condition and return value
  ;; body (executed each time through the loop)
  (format t "y=~a.." y))

y=1..y=2..y=4..y=8..y=16..
32
-----

(LOOP for x from 1
   for y = 1 then (* y 2)
   until (> x 5)
   finally (return y)
   do (format t "y=~a.." y))

y=1..y=2..y=4..y=8..y=16..
32
```

Example Loop vs Do vs Dotimes

```
(loop for i from 0 to 10 by 2
      collect (* i i))
=> (0 4 16 36 64 100)

(do ((result nil)
     (i 0 (+ i 2)))
    ((>= i 11) (nreverse result))
  (push (* i i) result))
=> (0 4 16 36 64 100)

(let (result)
  (dotimes (i 11)
    (when (evenp i)
      (push (* i i) result)))
  (nreverse result))
=> (0 4 16 36 64 100)
```

Intro and history

Lists, s-expressions, and cons cells

Data types

Evaluation

Misc operations

Variables and value binding

Flow of control

Defining functions

Evaluation and the Read-Eval-Print-Loop (REPL)

Emacs

Defining Functions

Lisp programs consist of functions that can be defined with **defun**.

```
(defun name (parameter*)
  "optional documentation string"
  body-form*)
```

Parameter* can optionally include non-fixed argments **&optional**, **&keyword**, **&rest** in that order. Optional and keyword args can also specify defaults.

```
(defun foo (a b) (list a b))
  * (foo 2 3) => (2 3)

(defun foo (a b &optional c (d :hello)) (list a b c d))
  * (foo 1 2 3) => (1 2 3 :hello)

(defun foo (a b &key c) (list a b c d))
  * (foo 2 3 :c 88) => (2 3 88 nil)

(defun foo (a b &rest args) (list (list a b) args))
  * (foo 2 3 22 33 44 55) => ((2 3) (22 33 44 55))
```

Defining functions examples

```
(defun dist (x1 y1 x2 y2)
  (sqrt (+ (expt (- x1 x2) 2)
           (expt (- y1 y2) 2))))
```

* (dist 0 0 3 4) → 5.0

```
(defun palindrome-string (string)
   (concatenate 'string string (reverse string)))
```

* (palindrome-string "blog") → "bloggolb"

```
(defun palindrome-list (lst)
   (append lst (reverse lst)))
```

* (palindrome-list '(a b c)) → (A B C C B A)

Another simple example – using an Alist

```
(defparameter *colors*
   '((red :rgb (1 0 0) :example "cherry")
     (green :rgb (0 1 0) :example "leaf")
     (blue :rgb (0 0 1) :example "sky")
     (yellow :rgb (1 1 0) :example "sun")
     (white :rgb (1 1 1) :example "cloud")
     (black :rgb (0 0 0) :example "ink")))

(defun describe-color (color)
  (let ((entry (assoc color *colors*)))
     (if entry
         (format nil "~a ~a is the color of ~a"
             color (getf (cdr entry) :rgb) (getf (cdr entry) :example))
         (format nil "RGB unknown for ~a" color))))

 * (describe-color 'blue)
  "BLUE (0 0 1) is the color of sky"
```

Functions as first class objects

Many built-in Lisp functions take other functions as arguments.

To specify a function argument and not have it be evaluated, you can use **#'** (which is shorthand for the special operator **function**).

(sort sequence predicate), where predicate is a comparitor function that takes two arguments and returns NIL or non-NIL (eg T).

```
(sort '(22 88 11 99 -44) #'<)
(-44 11 22 88 99)
```

```
(sort '(22 88 11 99 -44) #'>)
(99 88 22 11 -44)
```

```
(sort '(22 88 11 99 -44) (function >))
(99 88 22 11 -44)
```

Functions as first class objects

Use **funcall** or **apply** to call a function object.

```
(APPLY #'* '(2 3 4 5))
120

(defun plot (fn min max step)
  (loop for i from min to max by step do
       (loop repeat (FUNCALL fn i) do (format t "*"))
       (format t "~%")))

(plot #'exp 0 4 .5)  or (plot (function exp) 0 4 .5)
*
**
***
*****
********
*************
*********************
***********************************
*****************************************************
```

Lambda functions

Lisp functions don't need names. Create an anonymous function with **lambda**.

```
(defun collect (sequence test)
  (loop for i in sequence
      when (funcall test i)
      collect i))

(collect '(11 22 0 44 21 -7 9)
         (lambda (x) (zerop (mod x 11))))
(11 22 0 44)

(collect '(-4 11 aa 22 "foo" 0 44 21 -7 9)
         (lambda (x) (and (numberp x) (plusp x))))
(11 22 44 21 9)
```

Mapping functions

Mapcar and other mapping functions make it very easy to apply an arbitrary function to elements of a list.

```
(defun range (min max &optional (incr 1))
  (loop for i from min to max by incr
        collect i))

(range 1 10 2)
=> (1 3 5 7 9)

(MAPCAR #'sqrt (range 1 5))
=> (1.0 1.4142135 1.7320508 2.0 2.236068)

(MAPCAR (lambda (x) (cons x (if (oddp x) "ODD" "even")))
        (range 1 5))
=> ((1 . "ODD") (2 . "even") (3 . "ODD")
    (4 . "even") (5 . "ODD"))
```

Naming Conventions

Intra-word separators	remove-item	dash vs underscore, camelCase
Special variables	*PRINT-LENGTH*	surround with asterisks
Constants	+FASL-FILE-VERSION+	surround with + signs
Internal low-level function	%MEMORY-BARRIER	with percent
Predicates	EVENP, SYMBOLP	often end with “-p” or “p”
“Destructive” functions	nreverse, nconc	start with “n”
Dotted lists	LIST*	trailing * on function

Comments

```
Comments begin with a semi-colon (with three conventions).

;;; triple at beginning of line
(defun foo (x)
  ;; double above indented text
  (print "hello")
  (print "goodbye")) ; single at end of line

#|
 comment regions between these characters
|#

;;; comment an s-expression using a reader macro
#+skip(defun foo (x) (print "this is all commented out"))
```

Intro and history

Lists, s-expressions, and cons cells

Data types

Evaluation

Misc operations

Variables and value binding

Flow of control

Defining functions

Evaluation and the Read-Eval-Print-Loop (REPL)

Emacs

Evaluation and the Read-Eval-Print-Loop (REPL)

When an s-expression is typed to the REPL it is first **read**. This converts the character form into an s-expression. I.e. something within parentheses will be converted to a LIST. Sequences of characters within double quotes will be converted into strings. Etc.

The REPL then **evaluates** the s-expression. If the s-expression is a:
List: the first element is applied to the rest of the arguments
Symbol: the symbol's value is used
Other atom: the atom evaluates to itself

The evaluation step (above) returns a value or values that the REPL then **prints**. If the evaluation resulted in an error, then the interactive debugger is invoked.

This process is repeated (**looped**) for each successive input.

Using the REPL

Useful things to do in the REPL:

(**apropos** STRING-OR-SYMBOL)
  Find names of matching functions

(**describe** OBJECT)
  describe the given object

debugging variables
  *, **, *** $\rightarrow$ automatically set to last 3 returned values in REPL
  +, ++, +++ $\rightarrow$ automatically set to last 3 inputs to REPL

(**trace** function-name)
(**untrace** function-name)
  trace or untrace the given function

Trace example

```
CL-USER> (defun fact (n)
           (if (= n 1)
               1
               (* n (fact (- n 1)))))
FACT

CL-USER> (trace fact)
(FACT)

CL-USER> (fact 4)
  0: (FACT 4)
    1: (FACT 3)
      2: (FACT 2)
        3: (FACT 1)
        3: FACT returned 1
      2: FACT returned 2
    1: FACT returned 6
  0: FACT returned 24
24
```

Debugger

```
(defun fact (x)
  (if (= x 1)
      nil ; ** OOPS! **
    (* x (fact (- x 1))))) ==>
FACT

(fact 4) ==>

Error: In * of (1 NIL) arguments should be of type NUMBER.
  1 (continue) Return a value to use.
  2 Supply a new second argument.
  3 (abort) Return to level 0.
  4 Return to top loop level 0.

Type :b for backtrace or :c <option number> to proceed.
Type :bug-form "<subject>" for a bug report template or :? for other options.

WORDSEYE 58 : 1 >
```

Debugger

```
WORDSEYE 56 > (fact 4)

Error: In * of (1 NIL) arguments should be of type NUMBER.
  1 (continue) Return a value to use.
  2 Supply a new second argument.
  3 (abort) Return to level 0.
  4 Return to top loop level 0.

Type :b for backtrace or :c <option number> to proceed.
Type :bug-form "<subject>" for a bug report template or :? for other options.

WORDSEYE 58 : 1 > :b
Call to ERROR
Call to *
Interpreted call to FACT
Interpreted call to FACT
Interpreted call to FACT
Interpreted call to FACT
Call to EVAL
...
WORDSEYE 59 : 1 >
```

Intro and history

Lists, s-expressions, and cons cells

Data types

Evaluation

Misc operations

Variables and value binding

Flow of control

Defining functions

Evaluation and the Read-Eval-Print-Loop (REPL)

Emacs

Getting started

Make sure you have first installed Lispworks or SBCL/Emacs/Slime
(See install.pdf in courseworks2)

To write lisp code you generally have a buffer or buffers with your code and use emacs commands to compile it. You switch to the REPL to run/test as you work.

To create a buffer, type c-x c-f to emacs (or the Lispworks editor). Give your file a name that ends in ".lisp" (eg "my-code.lisp"). This will tell the editor that you are editing lisp vs random text. You can save your file with c-x c-s or write it to a new location with c-x c-w.

Put a package declaration at the top of the file. For now, just use (in-package :cl-user).

Then add your code below. Remember to keep it formatted/indented (using c-m-q). You can use c-c c-c (in GNU Emacs) or c-sh-c (in the Lispworks editor) to compile the Lisp form under the cursor, which will often point out errors.

See Emacs Cheat Sheet (next slide) and online documentation for more.

Emacs Cheat Sheet

Tour: `https://www.gnu.org/software/emacs/tour/`

Emacs Help
- Tutorial: c-h t
- Index of help commands: c-h ?
- Search for command: c-h a
- Describe command: c-h w
- Describe command: m-x apropos
- Describe key binding: c-h c

General
- Abort command (eg search): c-g
- Search: c-s (forward), c-r (rev)
- Search&repl: m-% [type values]
- Mark region: c-SPACE and move
- Exit: c-x c-c
- Load E-lisp file: m-x load-file
- Eval E-lisp expr: m-:

Panes
- Two panes (horiz): c-x
- Two panes (vert): c-x 2
- Single pane (selected one): c-x 0
- Switch focus to other pane: c-x o

Files and buffers
- Select buffer: c-x b [type name]
- List buffers: c-x c-b
- Open file: c-x c-f [type name]
- Save: c-x c-s
- Save as: c-x c-w [type name]
- Kill buffer: c-x k [select buffer]
- Create shell buffer: m-x shell

Lisp
- Start Lisp: m-x slime
- Compile Lisp form: c-c c-c
- Eval Lisp form: c-m-x
- Format/indent s-expr: c-m-q
- Symbol complete c-m-i, c-c TAB
- Find Lisp definition: m-.
- TAB will indent or complete
- SPACE will show function arglists
- REPL buffer: *slime-repl sbcl*
- Debugger buffer: *sldb sbcl/0*
- Yank prev REPL cmd: m-p
- Yank next REPL cmd: c-p

Deleting/restoring
- Char: c-d
- Word: m-d
- S-expr: c-m-d
- Kill line (store): c-k
- Delete/store marked region: c-w
- Store marked region: m-w
- Yank (paste) stored text: c-y
- Undo: c-/ or c-_

Misc text
- Upper case: m-u
- Lowercase: m-l
- Open newline: c-o
- Transpose s-expressions: c-m-t

Move cursor

```
c-f Forward a character
c-b Backward a character
m-f Forward a word
m-b Backward a word
c-m-f Forward s-expression
c-m-b Backward s-expression
c-c c-p Move to prev REPL cmd
c-c c-n Move to next REPL cmd

c-n Next line
c-p Previous line
c-a Beginning of line
c-e End of line

c-a Beginning of s-expression
c-e End of s-expression
c-m-u   UP s-expression

m-v Backward a page
c-v forward a page
c-l Center on page
m-< File beginning
m-> File end
```

Useful extensions
- Swap current buffer: c-;
- Select REPL buffer: c-m-;
- Select shell buffer: c-,
- Comment region: c-=
- Uncomment region: c-&