CS Intro to AI

Welcome to LISP

William Regli
Geometric and Intelligent Computing Laboratory
Department of Mathematics and Computer Science
Drexel University
http://gicl.mcs.drexel.edu
Objectives

• “Crash” Intro to Lisp in < n slides
• Introduce all major concepts
  – Symbolic (functional) programming
  – Interaction
  – Major syntax and control structures
  – Functions and macros
  – Really cool stuff
Sources

• B. Webber and L. Spector Lisp materials from UMD-CP
• David Touretzky’s book
  – “A gentle intro to symbolic computation”
• Regli’s online Lisp Resources page
  – http://edge.mcs.drexel.edu/regli/Classes/Lisp
LISP= **LIST** Processing

- J. McCarthy in 1959
- Motivation: a computational implementation of the $\lambda$-Calculus
- Adept at creation of functions for the manipulation of symbolic data
What is the $\lambda$ Calculus?

• The Lambda-calculus is a universal model of computation, that is, any computation that can be expressed in a Turing machine can also be expressed in the lambda calculus.

peanuts $\rightarrow$ chocolate-covered peanuts
rasins $\rightarrow$ chocolate-covered rasins
ants $\rightarrow$ chocolate-covered ants

$\lambda x. \text{chocolate-covered } x$

$(\lambda x. \text{chocolate-covered } x) \text{peanuts} \rightarrow$
$\text{chocolate-covered peanuts}$

$\lambda y. \lambda x. \text{y-covered } x$

$(\lambda y. \lambda x. \text{y-covered } x) \text{caramel} \rightarrow$
$\lambda x. \text{caramel-covered } x$
Why learn Lisp?

• Because Regli says so
  – It’s good for you, like eating lots of fiber…
• Very high-level language
  – Lots of work in few lines of code
• Very few rules, simple syntax
• Data == Code, Code can change and adapt (e.g. Genetic Programming)
• Gives you a radically different way of thinking about programming a computer
Why Lisp?

- Much more extensible
- Users have total control over what goes on
  - You don’t like the syntax, change it!
- ...The second reason for choosing lisp is the way in which
- Lisp in oriented toward the manipulation of symbols

- Other important factors:
  - Built-in Support for Lists
  - Automatic Storage Management
  - Dynamic Typing
  - First-Class Functions
    - part of data structures,
    - formed at run-time,
    - arguments to functions, and returned by functions
  - Uniform Syntax
  - Interactive Environment
  - Extensibility
What is Functional/Symbolic Programming?

• Focuses on high-level semantic issues
• Subsumes object-oriented and logic programming
• Emphasizes data abstraction and the aesthetics of the design and implementation of a programming languages
• Note: Many present trends (.NET, ONE, Java, C#) are moving toward a functional programming paradigm
Symbolic Programming Examples

- Maple
- Mathematica
- Emacs
- CLIPS
- Prolog
- Lisp
- ML
Contrast to C/C++

- Data and Code are distinct
- Procedural languages
  - Create methods to act on data
  - Pass data around to other methods
- Compiled languages
  - Systems are rigid once built
- Emphasis at the level of bytes, registers, bits…
  - syntactic details, low-level binary representation, word length, ASCII codes, and pointer arithmetic…
  - Ever try to access data at the end of a null pointer?
  - Seg faults? Bus errors? Missing names?
What Java owes to Lisp…

• Object Orientation
  – In Lisp long before C++, called “Flavors”
• Memory Management
  – Garbage Collection
• Package encapsulation
• Interpreted (sort of)
• … many other things

Note: Gossling and Steele standardized Java…
Steele standardized Lisp….
The First Things To Grok

• S-expressions
  – Everything is a pointer, CONS cells
• The Read-Eval-Print Loop
  – interactivity
• Recursion
  – Recursion
    • Recursion
• Lambda functions
S-expressions

- Symbolic expressions
  - An atom is an S-expression
    - symbol
    - non-symbol
      (number, string, array)
    - Examples: X, JOHN, SETQ, 7.2, "John"
  - A list is an S-expression
    - \((S\text{-expr } S\text{-expr ...})\)
    - Examples: \((+ 1 2), (A (B C))\)

Notes: case insensitive; \((A)\) is an abbrev for \((A . \text{NIL})\). Lists more commonly than dotted pairs;
\((\text{nil . nil}) = (()) . (() = (() . \text{nil}) = (() = /= \text{nil})

Lists in Lisp

- Everything in Lisp is a list

- Lists consist of
  - ATOMS
  - LISTS

(a b c 1 2 3)
(setf x 10)
(- x 10)
The CONS Cell

**CONS Cell:** A cell in memory (w/ left and right halves)

A  B

A  B

(CONS 'A 'B)  CAR = A  CDR = B

Note: No SIDE EFFECT (important later)

Note: combinations of CAR/CDR (up to 4), e.g.,
CADR = (CAR (CDR ..))

Also: FIRST, SECOND, ...
TENTH
(car,cadr,caddr, ...)

These are off in memory
Where are the pointers??

- EVERYTHING IS A POINTER!
- All atoms, lists, symbols etc get a location in memory
- Everything is accessed by a hidden pointer reference
- Issues to be aware of?
  - Which “A”? How to tell if things are equal?
The Read-Eval-Print Loop

• Sit in front of terminal
• Enter Lisp (interactive)
  – *top-level* read-eval-print loop
• EVAL = give me an expression to evaluate;
• Returns a value for an s-expr.
• Lisp reads the S-expr, evaluates the S-expr, then prints the result of the evaluation
Lisp Evaluation Rule

• Atoms:
  – If it is a numeric atom, the value is the number
  – Else it must be a variable symbol; so the value is the value of the variable

• Lists:
  – Assume the first item of the list is a function, and apply that function to the values of the remaining elements of the list
The quote ' symbol

- Examples:
  '2 -> 2
  2 -> 2
  'John -> John
  John -> ERROR
  – not a bound variable

- Quote: use ' to indicate “the S-expression stated” rather than the value of that S-expression.

- BLOCKs evaluation
Sample Lisp Session

;; Fire up Lisp ...
(+ 1 1)
=> 2
(* 23 13)
=> 299
(+ 1 (* 23 13))
=> 300
(+ (* 2 2) (− 10 9))
=> 5
(* 2 2 2 2)
=> 16

(sqrt (* 2 2 2 2))
=> 4
(/ 25 5)
=> 5
(/ 38 4)
=> 19/2
"Hello, world!"
=> "Hello, world!"
'x
=> X
x ; Can abort out of this
 (abort current
=> Error: Unbound variable X
 ; computation and return
to read-eval-print loop)
Sample Lisp Session

(setf x 5)
  ; Setf gives a value to a variable.
  ; It has a side
=> 5 ; effect and ALSO returns a value
(setf y (reverse '(+ a b)))
  ; Reverse reverses a list.
=> (B A +)
(setf l '(a b c d e))
  ; l's value will now be a list.
=> (A B C D E)
(length l)
  ; length --> gives length of a list
=> 5
(append l '(f))
  ; append --> merges to lists together
=> (A B C D E F)
  ; (Note use of quote; l not quoted)
(length l)
=> 5
  ; Note that the length of l has not changed.
(length (append '(pat Kim) (list '(John Q Public) 'Sandy)))
=> 4
  ; Symbolic computations can be nested and
  ; mixed with numeric computations.
;;; Note: "list" puts elts into a list;
  ; doesn't merge them like "append" does.
;;; Note: Order of evaluation critical.
  ; (Eval args first.)
;;; Note: Parens MUST match --> source of
  ; bizarre errors if not.
(car l) ; (first l)
=> A
(cdr l) ; (rest l)
=> (B C D E)
(first (rest l))
=> B
(cons 'q l) ; MAIN CONSTRUCTOR used in
  ; Lisp.
=> (Q A B C D E)
Lisp Data Structures

- Best way to understand Lisp: Develop an intuitive understanding of its data structures (remember: pgms and data stored the same way)
  - forget conventional pgming langs w/ which you are familiar (else confusion).
  - Three notations for Lisp data structures:
    - paren/list (printed)
    - dot (printed only if req; list notation is abbrev. for this);
    - pictorial or graphic or box/cell notation (never printed; shows what's going on inside)
CAR and CDR

• The first element of a list (be it an atom or another list) is the car of the list

(car '(alphabet soup)) => alphabet
(car '(((pickles beer) alphabet (soup)))
  =>(pickles beer)
(car '(((miro (braque picasso))) leger))
  =>((miro (braque picasso)))

• Everything except the first element of a list is the cdr of the list:

(cdr '(alphabet soup))
  =>(soup)
(cdr '(((pickles beer) alphabet (soup)))
  => (alphabet (soup))
(cdr '(((miro (braque picasso))) leger))
  =>(leger)
In addition to CAR/CDR

- first
- rest
- third
- second
- nth

- caar
- cadr
- ...

Building Lists with CONS

cons builds lists

(cons 'woof '(bow wow))
=> (woof bow wow)
(cons '(howl) '(at the moon))
=> ((howl) at the moon)

Create any list:

(cons 'naked-atom '())
=> (NAKED-ATOM)
(cons 'naked-atom nil)
=> (NAKED-ATOM)

car, cdr, and cons are all non-destructive:

(setf x ' (the rain in spain))
=> (THE RAIN IN SPAIN)
(car x)
=> THE
(cdr x)
=> (RAIN IN SPAIN)
(cons 'darn x)
=> (DARN THE RAIN IN SPAIN)
x
=> (THE RAIN IN SPAIN)
Other Commands for Building Lists: Append and List

• Append

  – **append** &rest lists => result
    
    (append '(a b c) '(d e f) () '(g)) => (A B C D E F G)

• List

  – **list** &rest objects => list
    
    (list 3 4 'a (car '(b . c)) (+ 6 -2)) => (3 4 A B 4)
Append and List

**Difference Between:** CONS, LIST, APPEND

- **CONS 'A 'B**
  - Adds one mem cell
  - \((A . B)\)
  - \((\text{CONS 'A NIL}) \rightarrow (A)\)

- **LIST 'A 'B**
  - Adds one mem cell for each elt
  - \((A B)\)
  - \((\text{LIST 'A}) \rightarrow (A)\)

- **APPEND '(A) '(B)**
  - All args must be lists!
  - Last ptr of each elt points to next elt
  - \((A B)\)
Append and List

Suppose:
\(X = (A)\)
\(Y = (B)\)

What is \((\text{CONS } X \ Y)\)？
What is \((\text{LIST } X \ Y)\)？

\[(\text{CONS } X \ Y) = ((A) \ B)\]
\[(\text{LIST } X \ Y) = ((A) \ (B))\]
Assignment: setf

- Generalized assignment
- Basically: modifies internal pointers

(setf A '4))

(setf (car A) 4)

Can use an accessor such as CAR, CDR, FIRST, etc.
Under the hood…

- Lisp doesn't need type declaration because everything is a pointer (i.e., the values of symbols have types associated w/ them; atoms, lists, integers, etc.)
How does this effect equality?

• Common lisp has a variety of Equality Predicates
  – equalp is the most general (see CLtL2 p. 103).
  – and, or, and not allow the combination of predicates into complex predicates.
• EQ and EQUAL ---most commonly used.
• EQ --- args eval to exact same Lisp obj
• EQUAL --- compares any two s-expressions (lists, strings, etc.)
• = used for numbers (must be same number)
• EQL: EQ or =
• EQUALP --- same as EQUAL but disregards case
• Others: tree-equal, char-equal, string-equal, string=, char=
# Equality in LISP

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>=</th>
<th>eq</th>
<th>eql</th>
<th>equal</th>
<th>equalp</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘x’</td>
<td>‘y’</td>
<td>Err</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>‘0’</td>
<td>‘0’</td>
<td>T</td>
<td>?</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>‘(x)’</td>
<td>‘(y)’</td>
<td>Err</td>
<td>Nil</td>
<td>Nil</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>”xy”</td>
<td>”xy”</td>
<td>Err</td>
<td>Nil</td>
<td>Nil</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>”Xy”</td>
<td>”xY”</td>
<td>Err</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
<td>T</td>
</tr>
<tr>
<td>‘0’</td>
<td>‘0.0’</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
<td>T</td>
</tr>
<tr>
<td>‘0’</td>
<td>‘1’</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
</tr>
</tbody>
</table>
Equality: what goes on in memory

- In reality, both pointers to A point to the SAME symbol (i.e., the same exact mem location)
- EQ: tests for memory location exactly, therefore, atoms only!
- LISP automatically makes sure that all refs to some symbol actually point to the UNIQUE point in mem that the symbol happens to be stored physically. This way, any info stored w/ it (e.g., its value) is accessible from every ref.
- Anytime LISP sees new symbol, it adds the symbol to its storehouse of known atoms
Equality Example

(equalp 'foot 'foot) => T
(equalp 'nose 'ear) => NIL
(equalp (+ 22 33 44) (* 33 3)) => T

(setq long-list '(1 2 3 4 can I show you out the door?)) => (1 2 3 4 CAN I SHOW YOU OUT THE DOOR?)
(setq lucky-number 23) => 23

(or (equalp lucky-number (car long-list))
  (equalp (* (car long-list) 2)
  (car (cdr long-list))))
=> T

(and (equalp lucky-number (car long-list))
  (equalp (* (car long-list) 2)
  (car (cdr long-list))))
=> NIL
Recursion

• Example: length in terms of itself
  – the empty list—length 0
  – other lists have a length which is one more than the length of the rest of the list.

(defun length (list)
  (cond ((null list) 0)
        (BASE)
        (t (1+ (length (cdr list))))))

• RECURSION: the leap of faith
Tips for Recursive Programming

• Think in terms of “Function Composition”
  – Build large answers from small ones

• Always ask what the “base case” is
  – Much like proof by induction

• Collect answers as recursive calls unwind
  – Pass answers “back up”… no need for vars, this is functional programming! Just return the result….

• Break down tasks into simpler versions of the same tasks then

• Specify a way to combine simpler versions of to solve the original problem.
Recursion Example

(defun lat? (l)
"Returns t if the argument is a list of atoms, nil otherwise"
(cond ((null l) t)
((atom (car l)) (lat? (cdr l)))
(t nil)))
=> LAT?
(lat? '(remember the alamo))
=> T
(lat? '(did you remember (to floss?)))
=> NIL
(lat? long-list)
=> T

(lat? 12)
=>
> Error: Can't take CAR of 12.
> While executing: LAT?
> Type Command-. to abort.
See the Restarts menu item for further choices.
1 >
(lat? '(remember the alamo))
=> T
(lat? '(did you remember (to floss?)))
=> NIL
(lat? long-list)
=> T
(lat? 12)
=> NIL
Recursion Example

(defun factorial (n)
  "returns the factorial of n"
  (cond ((<= n 1) 1)
        (t (* n (factorial (- n 1)))))
  => FACTORIAL
  (factorial 5)
  => 120
  (trace factorial)
  => NIL
  (factorial 5)
  Calling (FACTORIAL 5)
    Calling (FACTORIAL 4)
      Calling (FACTORIAL 3)
        Calling (FACTORIAL 2)
          Calling (FACTORIAL 1)
            FACTORIAL returned 1
          FACTORIAL returned 2
        FACTORIAL returned 6
      FACTORIAL returned 24
    FACTORIAL returned 120
  => 120

(defun member? (a lat)
  "Returns t if a occurs in lat, nil otherwise"
  (cond ((null lat) nil)
        (t (or (equalp (car lat) a)
               (member? a (cdr lat))))))
  => MEMBER?
  (setq five-colleges
        '(amherst umass hampshire
                    smith mount-holyoke))
  => (AMHERST UMASS HAMPSHIRE
         SMITH MOUNT-HOLYOKE)
  (member? 'hampshire five-colleges)
  => T
  (member? 'oberlin five-colleges)
  => NIL
**lambda**

- As close to the heart of computing as you’ll ever get
- Lambda is the fundamental function building block
  - Every function is composed of lambdas, except lambda
- Format:
  `(lambda (parameters ...) body ...)`
lambda

• Programmers who are used to other langs sometimes fail to see the point of lambda expressions. Why are they useful?
  – It's a pain to think up names and to clutter up a program with lots of functions that are only used very locally;
  – MORE IMPORTANT: We can create new functions at run time
Lambda Examples

(funcall #'(lambda (x) (+ x x)) 2)
  => 4

(apply #'(lambda (x) (+ x x)) '(2))
  => 4 ;(wasteful)

(mapcar #'(lambda (x) (+ x x)) '(1 2 3 4 5))
  => '(2 4 6 8 10)
Conditionals

- **IF**: special form
  
  \[
  \text{(if} \ (= \ x \ 21) \ \text{(print} \ \text{'blackjack})\text{)}
  \]

- **WHEN**: macro; same form:
  
  \[
  \text{(when} \ (= \ x \ 21) \ \text{(print} \ \text{'blackjack})\text{)}
  \]

- **What if more than 2-way fork? Use COND!**
  
  – Powerful multiway branching construct; Arbitrary number of args (called clauses) Note: general form can have more than one action.
COND Examples

```
(cond (x 'b) (y 'c) (t 'd))
What if x='a?
(retuns b)
What if x = nil, y = t?
(then returns c)
What if x = nil y = nil?
(then returns d)
```

Can also use OR and AND as conditional control constructs (as we talked about earlier):
(and nil t t t), (or t nil nil nil)

```
(cond (x (setf x 1) (+ x 2))
(y (setf y 2) (+ y 2))
(t (setf x 0) (setf y 0)))
What if x = t? (then returns 3) What is x? (x = 1)
What if x = nil, y = t? (then returns 4) What are x/y? (nil/2)
What if x = nil y = nil? (then returns 0) What are x/y? (0/0)
Note: could also do the following:
(cond (x (setf x 1) (+ x 2)) ((setf y 2)))
If x is nil, then it'll execute the last statement and return it.
IF vs COND

• JM(H)O: always use COND
• The IF special form is a special case of COND: IF testform thenform [elseform]
  – Evaluates testform. If the result is true, evaluates thenform and returns the result; if the result is nil, evaluates elseform and returns the result.
• Note that the thenform and the elseform are both restricted to being single forms. In contrast, you can specify any number of forms in a cond clause:

```lisp
(setq temperature 8)
=> 8
(cond
  ((< temperature 32)
   (print '(yowch -- it is cold!))
   print '(i will play god and change that!))
   setq temperature 78))
(t
   (print '(well i guess it is not so bad))
   (print '(where do you think we are? hawaii?)))
(yowch -- IT IS COLD!)
(I WILL PLAY GOD AND CHANGE THAT!)
=> 78
```
format

- The command for formatting output
- Truly amazing set of options
- Learn it by reading the man pages in the hyperspec
Variable scoping in Lisp

• Lisp uses lexical binding
  – cannot access vars not declared locally
    (unless special global), even if defined inside of calling fn.

• **Let** is the most common way of introducing vars that are not parameters of fns; resist temptation to use a var w/o introducing it (can actually do this in Lisp).

• **Let** introduces a new local variable and binds it to a value. General form:

  \[(\text{let ( (var value) ...) body-containing-vars})\]

• Equiv to:

  \[((\text{lambda (var ...)} body-containing-vars) value)\]
Variable scoping in Lisp

- **Let** used to initialize LOCALS; **Setf** used to initialize GLOBALS (or use **defparameter, defconstant, defvar**).
- Need to know symbols (name for var or fn) are NOT vars (depends on context). Symbol value does NOT change when fn is called w/ a symbol as a local var:
  - `(symbol-value 'x)` returns global val of x
  - x returns global val of x
- **IMPORTANT:** Try not to overload programs w/ globals; use locals where possible.
Let and let*

- The LET special form is used to create local variables.
  \[
  \text{(let \((\text{foo} \ 2) \ (\text{bar} \ 3))\)} \newline
  \text{ (+ \text{foo} \ \text{bar})}) \newline
  \Rightarrow \ 5
  \]

- LET (\{variable \* (variable value) \}*) \{declaration\}* \{form\}* [Special Form]

- LET creates a binding for each variable (in parallel) and evaluates forms in the resulting environment. Returns the value of the last form.

- Note that the initializations in a LET are performed “in parallel” If you want sequential initialization behavior, use LET*:

  \[
  \text{(setq laadeedaa \ 'hihohiho)} \newline
  \Rightarrow \ \text{HIHOHIHO}
  \]
  \[
  \text{(let ((laadeedaa \ 'feefifofum))} \newline
  \text{ (list laadeedaa laadeedaa \ laadeedaa))} \newline
  \Rightarrow \ (\text{FEFIFOFOFUM FEFIFOFOFUM \ FEFIFOFOFUM})
  \]
  laadeedaa
  \Rightarrow \ \text{HIHOHIHO}

  \[
  \text{(setq laadeedaa \ 'hihohiho)} \newline
  \Rightarrow \ \text{HIHOHIHO}
  \]
  \[
  \text{(let* ((foo \ 2) (bar (+ foo 3)))} \newline
  \text{ (+ foo bar))} \newline
  \Rightarrow \ 7
  \]
The time macro
PROG1, PROG2, PROGN
(if (< temperature 32)
  (progn (print '(yowch -- it is cold!))
    (print '(i will play god and change that!))
    (setq temperature 78))
  (progn (print '(well i guess it is not so bad))
    (print '(where do you think we are? hawaii?))))

=> (WHERE DO YOU THINK WE ARE? HAWAII?)
Defining Functions w/ Defun

• Define Function = \Defun\“
  (defun function-name (parameter ...) function-body)

• Function name = symbol;
  parameters are usually symbols
  (defun first-name (name)
   (first name))
  (first-name '(john q public))
  => JOHN
• Here's a function that takes one argument and returns the argument plus 20:

    (defun add-20 (n)
      "returns n + 20"
      (+ n 20))

=> ADD-20

    (add-20 15)

=> 35

• Note: embedded docs!

    (documentation 'add-20
      'function)

=> returns n + 20

    (describe 'add-20)

=> ADD-20 is a symbol. Its home package is USER. Its global function definition is

    #<Interpreted-Function
      ADD-20 FB0C26> ... Its source code is (NAMED-LAMBDA ADD-20 (N) (BLOCK
      ADD-20 (+ N 20))) ... It has this function documentation: "returns n + 20" ...
Defun --- some examples

(defun dumb-function ()
  '(you might as well use a
    variable for something
    like this))
=> DUMB-FUNCTION
(dumb-function)
=> (YOU MIGHT AS WELL USE A
    VARIABLE FOR SOMETHING
    LIKE THIS)
(defun rand8 ()
  (random 8))
=> RAND8
(rand8)
=> 0
(rand8)
=> 6
(rand8)

(defun even-or-odd? (n)
"returns 'even if n is even,
  'odd if n is odd, and
  'neither if n is not an in
(cond ((not (integerp n))
  'neither)
  ((evenp n) 'even)
  ((oddp n) 'odd)))
=> EVEN-OR-ODD?
(even-or-odd? 24)
=> EVEN
(even-or-odd? 25)
=> ODD
(even-or-odd? '(swahili))
=> NEITHER
Defun --- some examples

(defun my-sum (n1 n2)  
"silly substitute for +"  
(+ n1 n2))  
=> MY-SUM  
(my-sum 99 100)  
=> 199  
(defun +store (n1 n2)  
"Returns the sum of n1 and n2, and also sets *last-sum* to the"  
(setq *last-sum* (+ n1 n2)))  
=> +STORE  
(+store 99 100)  
=> 199  
*last-sum*  
=> 199

(defun funny-arg-lister (arg1 arg2 arg3)  
(cons (cons 'arg1 (cons arg1 nil))  
(cons (cons 'arg2 (cons arg2 nil))  
(cons (cons 'arg3 (cons arg3 nil))  
nil))))  
=> FUNNY-ARG-LISTER  
(funny-arg-lister 'a 'b '(x y z))  
=> ((ARG1 A) (ARG2 B) (ARG3 (X Y Z)))
(setq shoes 'nikes)
=> NIKES
(defun run (shoes)
  (print (append '(i think i will
take a trot in my)
(list shoes)))
(setq shoes (append '(old beat-up)
(list shoes)))
(print (append '(i think i will
take a trot in my) shoes)))
=> RUN
(run shoes)
(I THINK I WILL TAKE A TROT IN MY
NIKES)
(I THINK I WILL TAKE A TROT IN MY
OLD BEAT-UP NIKES)
=> (I THINK I WILL TAKE A TROT IN
MY OLD BEAT-UP NIKES)
Predicates

• Recall “Predicate Logic”
• Predicates are T/NIL (aka F) functions
• Examples
  - (atom x), (null x), (not x), (listp x),
    (consp x), (numberp x), (stringp x),
    (arrayp x), (vectorp x), (characterp x),
    (member x y :test <test>)
Lisp Comments

• ; everything after is ignored
• #|  
  everything in between is ignored  
  |#
#'

- (sharp-quote): maps name of function to function itself. Equivalent to FUNCTION.
- Use this when using mapcar, apply, funcall, lambda expressions keywords (e.g., :TEST).
- Note: only functions may be quoted (not macros or special forms).
- More efficient; tells compiler: go to compiled code, not through symbol to find function defn.
- Analogous to quote, but for functions
MAP’ing functions

• Mapcar: Expects an n-ary fn as 2st arg, followed by n lists. It applies the fn to the arg list obtained by collecting the first elt of each list.
  – (mapcar #'1+ '(5 10 7))
    (6 11 8)
  – (mapcar (function 1+) '(5 10 7))
    (6 11 8)
  – (mapcar #'cons '(a b c) '(1 2 3))
    ((A . 1) (B . 2) (C . 3))
  – (mapcar #'print '(a b c))
    (A B C)
• Note: last case also has side effect of printing A, B, and C.
• Avoid consing up a whole new list by using Mapc:
  – (mapc #'print '(a b c)) ! (A B C)
• [This prints A, B, and C, but then returns the second arg, NOT a new list.]
Apply, Funcall, Eval

- Following forms equivalent:
  - (+ 1 2 3 4)
  - (funcall #'+ 1 2 3 4)
  - (apply #'+ '(1 2 3 4))
  - (apply #'+ 1 2 '(3 4))
  - (eval '(+ 1 2 3 4))
- Funcall: great to use if we don't know function name in advance
  - (funcall <fn> arg1 arg2 ...)
- Applies to arguments, not in a list.
Apply, Funcall, Eval

• But what if we don't know the no. of args in advance?
• Apply: same idea, but don't need to know no. of args
  – (apply <fn> arg1 arg2 ... arglist)
• Applies to arguments; last arg MUST be a list
• Eval: in general we can use funcall and apply.
Argument Lists
&optional
&rest
Keyword Arguments: &keyword
structures
Property Lists
Hash tables
sort
arrays
Macros
When should you use macros?
Compilation
DEFVAR, DEFPARAMETER, DEFCONSTANT
Lisp Memory
Note: On Lisp Performance
Taking Lists Apart: CAR, CDR
Putting Lists Together: CONS, LIST, APPEND
Dotted Pair Notation
The TRACE facility
MEMBER
Lexical Closures
Characters and Strings
Backquote/Comma
Gensym
Alists, Assoc
Lisp: Syntactic Details w/ Examples
Looping
Making Lists
MAP functions
I/O: Command Line
I/O: Files