Introduction to LISP

Miros Hauskrecht
milos@cs.pitt.edu
5329 Sennott Square

LISP language

LISP: LISt Processing language
• An AI language developed in 1958 (J. McCarthy at MIT)
• Special focus on symbolic processing and symbol manipulation
  – Linked list structures
  – Also programs, functions are represented as lists
• At one point special LISP computers with basic LISP functions implemented directly on hardware were available (Symbolics Inc., 80s)

LISP today:
• Many AI programs now are written in C, C++, Java
  – List manipulation libraries are available
LISP language

LISP Competitors:
• Prolog, Python
• but LISP keeps its dominance among high level (AI) programming languages

Current LISP:
• Common Lisp
• Scheme

are the most widely-known general-purpose Lisp dialects

Common LISP:
• Interpreter and compiler
• CLOS: object oriented programming

LISP tutorial

Syntax:
• Prefix notation
  – Operator first, arguments follow
  – E.g. (+ 3 2) adds 3 and 2

A lot of parentheses
• These define lists and also programs
• Examples:
  – (a b c d) is a list of 4 elements (atoms) a,b,c,d
  – (defun factorial (num)
      (cond ((<= num 0) 1)
            (t (* (factorial (- num 1)) num)))
      ))
LISP tutorial: data types

Basic data types:

- **Symbols**
  - `a`
  - `john`
  - `34`

- **Lists**
  - `()`
  - `(a)`
  - `(a john 34)`
  - `(lambda (arg) (* arg arg))`

LISP tutorial

For each symbol lisp attempts to find its value

```lisp
> (setq a 10) ;; sets a value of symbol a to 10
10
> a ;; returns the value of a
10
```

Special symbols:

```lisp
> t ;; true
T
> nil ;; nil stands for false or
NIL
> () ;; an empty list
NIL
```
LISP tutorial

Lists represent function calls as well as basic data structures
>  (factorial 3)
  6
>  (+ 2 4)
  6

>  (setq a ‘(john peter 34)) ;; quote means: do not eval the argument
  (john peter 34)
>  (setq a ‘((john 1) (peter 2)))
  ((john 1) (peter 2))

LISP tutorial: lists

List representation:
• A singly linked list

>  (setq a ‘(john peter))
  (john peter)
>  (car a)
  john
>  (cdr a)
  (peter)
LISP tutorial: list

List building functions
> (cons ‘b nil) ;; quote means: do not eval the argument
  (b)
> (setq a (cons ‘b (cons ‘c nil))) ;; setq a is a shorthand for set ‘a
  (b c)
> (setq v (list ‘john 34 25))
  (john 34 25)
> (setq v (list a 34 25))
  ((b c) 34 25)
> (append ‘(1 2) ‘(2 3))
  (1 2 2 3)

LISP tutorial

List copying
> (setq foo (list 'a 'b 'c))
  (a b c)
> (setq bar (cons 'x (cdr foo)))
  (x b c)
> foo
  (a b c) ;; (cdr foo) makes a copy of the remaining list before cons
> bar
  (x b c)
• Car and cdr operations are nondestructive.
LISP tutorial: lists

\[(setq\ bar\ '(a\ b\ c))\]
(a\ b\ c)
\[(setq\ foo\ (cdr\ bar))\]
(b\ c)
\[(rplaca\ foo\ 'u)\ \;;\ replaces\ car\ component\ of\ foo\ (destructive\ op)\]
(u\ c)
\[foo\]
(u\ c)
\[bar\]
(a\ u\ c)
\[(rplacd\ foo\ '(v))\ \;;\ replaces\ cdr\ component\ of\ foo\ (destructive)\]
(u\ v)
\[bar\]
(a\ u\ v)

LISP tutorial

The same effect as with \texttt{rplaca} and \texttt{rplacd} can be achieved with \texttt{setf}
\[(setq\ bar\ '(a\ b\ c))\]
(a\ b\ c)
\[(setq\ foo\ (cdr\ bar))\]
(b\ c)
\[(setf\ (cadr\ bar)\ 'u)\]
(u)
\[bar\]
(a\ u\ c)
\[foo\]
(u\ c)
**LISP tutorial**

**Evaluation rules:**
- A symbol value is sought and substituted
- A quoted value is kept untouched

```lisp
> (setq a 12)
12
> (setq b (+ a 4))
16
> (setq b '(+ a 4))
(+ a 4)
> (eval b) ;; explicit evaluation call
16
```

**LISP tutorial: functions and predicates**

**Some useful functions and predicates:**
```
> (setq a '(1 2 3 4 5))
(1 2 3 4 5)
> (length a) ;; gives the list length of the argument
5
> (atom 'a) ;; checks if the argument is an atom
T
> (atom a)
NIL
> (listp 'a) ;; checks if the argument is a list
NIL
> (listp a)
T
```
LISP tutorial: function definition

Definition of a function

(defun <f-name> <parameter-list> <body>)

> (defun square (x)
   (* x x))
   SQUARE
> (square 2)
   4
> (square (square 2))
   16

LISP tutorial

Definition of a function

(defun <f-name> <parameter-list> <body>)

<body> can be a sequence of function calls, the function returns the value of the last call in the sequence

> (defun foo (a)
   (setq b (+ a 1))
   (setq c (+ a 2))
   c)
   FOO
> (foo 2)
   4
LISP tutorial: conditionals

**Cond statement:** sequentially tests conditions, the call associated with the first true condition is executed.

```lisp
> (defun abs (a)
    (cond ((> a 0) a)
          (t   (- a))))
ABS
> (abs 2)
2
> (abs -3)
3
```

LISP tutorial

**if statement:**

```lisp
(if <test> <then> <else>)
```

```lisp
> (defun abs (a)
    (if (> a 0) a (- a)))
ABS
> (abs 2)
2
> (abs -3)
3
```
LISP tutorial: equality

4 equality predicates: =, equal, eq, eql

\[
> (= 2 4/2) \quad ;; \text{used for numerical values only}
T
\]

\[
> (setq a '(1 2 3 4))
(1 2 3 4)
\]

\[
> (setq b '(1 2 3 4))
(1 2 3 4)
\]

\[
> (setq c b)
(1 2 3 4)
\]

\[
> \text{(equal a b)} \quad ;; \text{equal is true if the two objects are isomorphic}
T
\]

\[
> \text{(equal c b)}
T
\]

LISP tutorial: equalities

\[
> \text{(eq a b)} \quad ;; \text{eq is true if the two arguments point to the same object}
NIL
\]

\[
> \text{(eq b c)}
T
\]
LISP tutorial: nil

Nil represents False and an empty list

```lisp
> (null nil) ;; tests if the argument is NIL
  T
> (null ( ))
  T
> (null `(a b))
  NIL
> (not `(a b))
  NIL
```

LISP tutorial: functions

Logical operators: and, or

```lisp
> (and NIL T)
  NIL
> (and T 2 3)
  3
> (or nil (= 5 4))
  NIL
> (or nil 5)
  5
```
Recursive function definitions are very common in LISP

> (defun factorial (num)
  (cond ((<= num 0) 1)
        (t (* (factorial (- num 1)) num))
    )
  FACTORIAL
> (factorial 4)
  24

> (defun check_lists (lis)
  (cond ((null lis) nil)
       (t (cons (listp (car lis)) (check_lists (cdr lis))))))
  CHECK_LISTS
> (check_lists (list ‘a ‘(1 2) 3 ‘(a b c) ‘(a)))
  (NIL T NIL T T)
LISP tutorial: local and global variables

> (setq a 12)
  12
> (defun foo (n)
   (setq a 14)
   (+ n 2))
   FOO
> a
  12
> (foo 3)
  5
> a
  14

LISP tutorial: local variables

Defining local variables with let

> (setq a 7) ;store a number as the value of a symbol
  7
> a ;take the value of a symbol
  7
> (let ((a 1)) a) ;binds the value of a symbol temporarily to 6
  1
> a ;the value is 7 again once the let is finished
  7
> b ;try to take the value of a symbol which has no value
Error: Attempt to take the value of the unbound symbol B
LISP tutorial: local variables

Defining local variables with let and let*

\[
\begin{align*}
> \text{(let ((a 5)) ; binds vars to values locally (b 4))} \\
& (+ a b)) \\
9 \\
> \text{(let* ((a 5) ; binds vars sequentially (b (+ a 2)))} \\
& (+ a b)) \\
12
\end{align*}
\]

LISP tutorial: functions revisited

Standard function – all parameters defined

\[
\begin{align*}
\text{(defun fact (x))} \\
& \text{(if (> x 0)} \\
& \quad (* x \text{(fact (- x 1))}) \\
& 1))
\end{align*}
\]

But it is possible to define functions:
- with variable number of parameters,
- optional parameters and
- keyword-based parameters
LISP tutorial: functions revisited

Functions with optional parameters

> (defun bar (x &optional y) (if y x 0))
BAR
> (defun baaz (&optional (x 3) (z 10)) (+ x z))
BAAZ
> (bar 5)
0
> (bar 5 t)
5
> (baaz)
13
> (baaz 5 6)
11
> (baaz 5)
15

LISP tutorial: functions revisited

Functions with variable number of parameters

> (defun foo (x &rest y) ;; all but the first parameters are put
  ;; into a list
  y)
FOO
> (foo 3)
NIL
> (foo 1 2 3)
(2 3)
> (foo 1 2 3 4 5)
(2 3 4 5)
LISP tutorial: functions revisited

Functions with ‘keyword’ parameters

> (defun foo (&key x y) (cons x y))
FOO
> (foo :x 5 :y '(3))
(5 3)
> (foo :y '(3) :x 5)
(5 3)
> (foo :y 3)
(NIL 3)
> (foo)
(NIL)

LISP tutorial: arrays

List is a basic structure; but arrays and structures are supported

> (setf a (make-array '(3 2))) ;; make a 3 by 2 array
#2a((NIL NIL) (NIL NIL) (NIL NIL))
> (aref a 1 1)
NIL
> (setf (aref a 1 1) 2)
2
> (aref a 1 1)
2
LISP tutorial: structures

>(defstruct weather
    temperature
    rain
    pressure)
WEATHER
> (setf a (make-weather)) ;; make a structure
#s(WEATHER :TEMPERATURE NIL :RAIN NIL :PRESSURE NIL)
> (setf a (make-weather :temperature 35))
#s(WEATHER :TEMPERATURE 35 :RAIN NIL :PRESSURE NIL)
> (weather-temperature a) ;; access a field
35
> (weather-rain a)
NIL
> (setf (weather-rain a) T) ;; set the value of a field
T
> (weather-rain a)
T

LISP tutorial: iterations

Many ways to define iterations

Commands:
• loop
• dolist
• dotimes
• do, do*

Also we can write compactly the code for repeated application of function to elements of the list:
• mapc, mapcar
LISP tutorial: iterations

Iterations: loop

> (setq a 4)
4
> (loop (setq a (+ a 1))
    (when (> a 7) (return a))) ;; return exists the loop
8
> (loop (setq a (- a 1))
    (when (< a 3) (return)))
NIL

LISP tutorial: iterations

Iterations: dolist

> (dolist (x '(1 2 3 4)) (print x))
1
2
3
4
NIL ;; NIL is returned by dolist
>
LISP tutorial: iterations

Iterations: dotimes

> (dotimes (i 4) (print i)) ;; starts from 0 and continues till limit 4
  0
  1
  2
  3
  4
NIL ;; returns NIL

LISP tutorial: iterations

Iterations: do

> (do ((x 1 (+ x 1)) ;; variable, initial value, next cycle update
       (y 1 (* y 2))) ;; the same
     ((> x 5) y) ;; end condition, value do returns
     (print (list x y))) ;; body of do – a sequence of operations
     (print 'next))
(1 1)
NEXT
(2 2)
NEXT
(3 4)
NEXT
(4 8)
NEXT
(5 16)
NEXT
32
LISP tutorial: iterations

Iterations: do *

> (do* ((x 1 (+ x 1)) ;; variable, initial value, next cycle update
    (y 1 (* x 2)))  ;; <<< --- update based on x
    ((> x 5) y) ;; end condition, value do returns
    (print (list x y))) ;; body of do – a sequence of operations
    (print ‘next))

(1 1)
NEXT
(2 4)
NEXT
(3 6)
NEXT
(4 8)
NEXT
(5 10)
NEXT
12

LISP tutorial: mapcar

Repeated application of a function to elements of the list

> (mapcar #’oddp ‘(1 2 3 4 5)) ;; named function
(T NIL T NIL T)
> (mapcar #’(lambda(x) (* x x)) ‘(1 2 3 4 5)) ;;temp function
(1 4 9 16 25)
Evals and function calls

- A piece of code can be built, manipulated as data
- What if we want to execute it?

\[
\begin{align*}
\text{> (setq b ‘(+ a 4))} \\
\text{(+ a 4)} \\
\text{> (eval b) ;; explicit evaluation call} \\
\text{16} \\
\text{> (funcall #’+ 2 4) ;; calls a function with args} \\
\text{6} \\
\text{> (apply #’+ 2 ‘(5 6)) ;; calls a function with args} \\
\text{13 (last args as a list)}
\end{align*}
\]

LISP tutorial: input/output

You can input/output data to:
- standard input/output,
- string or
- file

A number of functions supported by the Lisp:
- (read) ;; reads the input from the standard input
- (print ‘a) ;; prints to the standard output
- (scanf …) (printf …) (format …) for formatted input and output
- (open ..) (close ..) for opening and closing the files
- (load ..) read and executes the file
LISP tutorial: program calls

Assume you have your lisp code ready in the .lisp file
This is how you load it
(load "~/private/lsp/file-to-load.lisp")
… and you can call another load from it as well

Running LISP for CS Students

• Remotely login via ssh to elements.cs.pitt.edu
• LISP is installed in the following directory:
  /usr/local/contrib/cmucl-19d/
• You can run lisp from linux by typing /usr/local/contrib/cmucl-19d/bin/lisp
  – You may want to provide a path to the lisp directory so that the executable is seen from anywhere
  – To do this, edit your .cshrc.custom file under your home directory and add the following line:
    set path = ($path /usr/local/contrib/cmucl-19d/bin)
• Use the command (quit) to quit LISP
Running LISP for Non-CS Students

- Remotely login via ssh to unixs.cis.pitt.edu
- LISP is installed in the following directory: /usr/pitt/franz-lisp/
- You can run lisp from unix by typing: /usr/pitt/franz-lisp/mlisp
  - You may want to provide a path to the lisp directory so that the executable is seen from anywhere
  - To do this, edit your .cshrc file under your home directory and add the following line:

    ```
    set path = ($path /usr/pitt/franz-lisp)
    ```
    - If .cshrc is read-only, then add write permission with the command: chmod u+w .cshrc
- Use the command (exit) to quit LISP