

Common Lisp: A Brief Tutorial

Elham S.Khorasani

Introduction

- The most commonly used programming language for Artificial Intelligence
- Highly interactive
- Originally developed in 1958 (was run interpreted)
- **ANSI Common Lisp** developed late 1994 (code is compiled before run)
- Is extensible and dynamic
- Automatic Garbage collection like java
- Typed **Values** instead of typed **variables**
- You will be using **Allegro Common Lisp (ACL 8.1)**

Syntax

- Prefix notation
 - Operator first, arguments follow:
(operator argument1 argument2 argumentn)
 - Example:
 - >(+ 2 3 4)
 - >(print "lisp commands are evaluated immediately")
- Data Types:
 - ❖ Basic Data types: self-evaluated
 - ❖ Numbers: >34.5
 - ❖ Strings : > "Hello"
 - ❖ Characters: > #\a
 - ❖ Lists :
 - ❖ > (list 1 2 3 4)
- Back quote: prevent the evaluation, treats the argument as a symbol
 - ❖ >'a
 - a
 - ❖ >'(a b c)
 - (a b c)

Lisp Variables

- **Special** variables:
 - More like global variables in C/C++/Java
 - Declared using the keyword **defvar**

```
>(defvar counter 0)
```
- Use **setq** command to set a value for a variable:

```
>(setq counter 1)
```
- **Lexical** variables:
 - More like a Local variables in C/C++/Java
 - Have a scope
 - Declared using the special keyword **let**

Try:

```
>(let ((x 5)
      (y 6)) x)
```



Returns 5

```
>(let ((x 5)
      (y 6))
  >x)
```



Error

Lisp Functions

- Declared using the keyword “defun” :

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

- Example

```
>(defun square (x)  
  (* x x))
```

```
>(defun foo (n)  
  (let ((b (+ n 2)))  
    b))
```

- Calling the function: (<fname> <arguments>)

```
>(square 6)
```

```
36
```

```
>(foo 5)
```

```
7
```

Conditional Statements (IF)

- “If” statement

(if test expr1 [expr2])

- Example: factorial function

```
>(defun fact(n)
  (if (> n 1)
      (* n (fact (- n 1)))
      1
  ))
>(fact 5)
120
```

Conditional Statements (Cond)

- “Cond” statement:
 - sequentially tests conditions, the call associated with the first true condition is executed

```
(cond ((test1) expression1)
      ((test2) expression 2) )
```

- Example: Absolute function

```
>(defun absolute (a)
  (cond
    ((> a 0) a)
    (t (- a)) ))
>(absolute 4)
4
>(absolute -4)
4
```

Some Predicates

`numberp`, `integerp`,
`stringp`, `characterp`
`evenp`, `oddp`

test whether arg is
a number, integer,
string, character, etc.

```
(numberp 5.78)  $\implies$  T  
(integerp 5.78)  $\implies$  NIL  
(characterp \#a)  $\implies$  T
```

`listp`, `atom`,
`null`, `consp`

test whether arg is a list,
atom, empty/nonempty list

```
(listp nil)  $\implies$  T  
(consp nil)  $\implies$  NIL
```

`<`, `<=`, `=`, `>=`, `>`

numeric comparisons

arg must be a number

`string<`, `string<=`, ...

string comparisons

arg must be string or char

`eql`, `equal`

equality tests; they
work differently on
lists and strings

```
(setq x '(a))  
(eql x x)  $\implies$  T  
(eql x '(a))  $\implies$  NIL  
(equal x '(a))  $\implies$  T
```

`and`, `or`, `not`

logical predicates; `not`
and `null` are identical

```
(not (evenp 8))  $\implies$  NIL  
(and 3 'foo T)  $\implies$  T
```

Optional Arguments & Default Values

```
> (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
```

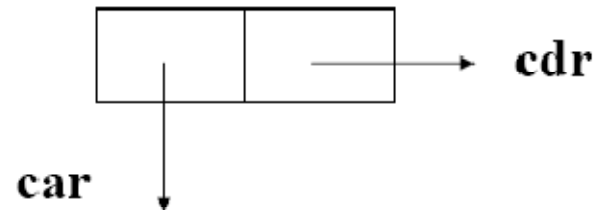
```
> (baaz 5)
```

```
15
```

Lists

List representation:

- A singly linked list



- > (setq a '(john peter))
 (john peter)
- > (car a)
 john
- > (cdr a)
 (peter)

Some List Operators

<code>first, car</code>	1st element	<code>(first '(a b c d)) ⇒ a</code>
<code>second, ..., tenth</code>	like <code>first</code>	<code>(third '(a b c d)) ⇒ c</code>
<code>rest, cdr</code>	all but 1st	<code>(rest '(a b c d)) ⇒ (b c d)</code>
<code>nth</code>	<i>n</i> th element, <i>n</i> starts at 0	<code>(nth 2 '(a b c d)) ⇒ c</code>
<code>length</code>	#of elements	<code>(length '((a b) c (d e))) ⇒ 3</code>
<code>cons</code>	inverse of <code>car</code> & <code>cdr</code>	<code>(cons 'a '(b c d)) ⇒ (a b c d)</code> <code>(cons '(a b) 'c) ⇒ ((a b) . c)</code>
<code>list</code>	make a list	<code>(list '(a) '(b c) (+ 2 3))</code> <code>⇒ ((a) (b c) 5)</code>
<code>append</code>	append lists	<code>(append '(a) '(b c) '(d)) ⇒ (a b c d)</code> <code>(append '(a) '(b c) 'd) ⇒ (a b c . d)</code>
<code>reverse</code>	reverse a list	<code>(reverse '((a b) c d)) ⇒ (d c (a b))</code>

Arrays

- Creating an array

```
>(setf arr (make-array '(3 2)))
```

```
#2A((NIL NIL) (NIL NIL) (NIL NIL))
```

```
>(setf weekdays #("Mon" "Tue" "Wed" "Thu" "Fri" "Sat" "Sun"))
```

- Accessing the elements of an array

```
>(aref weekdays 1)
```

```
“Tue”
```

```
>(aref arr 1 1)
```

```
Nil
```

```
>(setf (aref arr 1 1) 5.5)
```

```
>a
```

```
#2A((NIL NIL) (NIL 5.5) (NIL NIL))
```

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
```

Iterations

- Many ways to define iterations:
- Commands:
 - Loop
 - While
 - Dolist
 - Dotimes
 - Do

Iterations (Cont.)

- (LOOP FOR i FROM start TO end....)

```
>(setf arr #( 3 4 5 7 8 9))  
>(loop for i from 2 to 5 do  
      (incf (aref arr i)))  
  
>arr  
#(3 4 6 8 9 10)
```

- (LOOP FOR element IN list...)

```
>(let ((flag nil)  
      (people '(bob ann todd)))  
  (loop for person in people do  
    (if (eq person 'ann)  
        (setq flag t)))  
  flag)
```

Iterations (Cont.)

- **While**

```
>(loop while (> x 0) do  
  (print x)  
  (decf x))
```

- **Dolist**

```
>(dolist (x (list 1 2 3 4)) (print x))
```

- **Dotimes**

```
>(dotimes (i 4) (print i))
```

- **Do**

```
>(do ((x 1 (+ x 1)) ;; Variable, initial value, next cycle update  
      (y 1 (* y 2))) ;; The same  
      ((> x 5)) ;; End condition  
      (print (list x y))) ;; body of do a sequence of operations
```

Higher order functions

- Functions can have other functions as their arguments
 - Calling functional parameter inside Defun: **“funcall”**
 - Pass function as an argument to another function: use **#’myfunction**
- Example: A generic min function that finds the minimum of elements in an array of any type:

```
>(defun generic-min (arr lt)
  (let ((m (aref arr 0)))
    (loop for i from 1 to (1- (length arr)) do
      (if (funcall lt (aref arr i) m)
          (setq m (aref arr i)) ))
    m ))
>(setf weekdays #("Mon" "Tue" "Wed" "Thu" "Fri" "Sat" "Sun"))
>(generic-min weekdays #’String<)
“Fri”
```

Practice

- A pseudo code version of insertion sort is given below, assuming an array/vector of numbers is being sorted.
- Write the lisp program of this pseudo code .
- Use the idea in the pervious example to make the code you wrote generic, that means to sort array of any type. (You need to pass a functional parameter for comparing the elements of the array).

- INSERTION-SORT(A)
- FOR i <- 2 to length[A] DO
- key <- A[i]
- j <- i-1
- WHILE j>0 AND key<A[j] DO
- A[j+1] <- A[j]
- j <- j-1
- A[j+1] <- key
- RETURN A