

Lisp

CS161 Discussion 1

Shirley Chen

09/27/2019

CLISP on SEASnet

```
ssh -X lnxsrv.seas.ucla.edu -l yourseasaccountname  
clisp
```

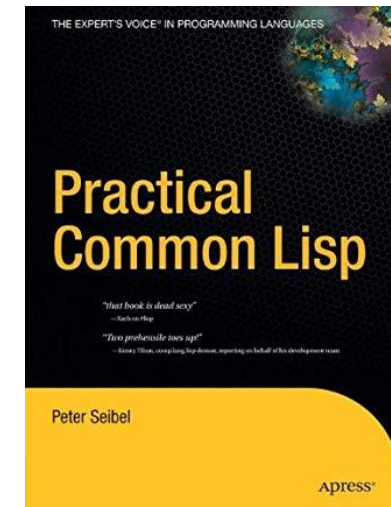
Windows: use putty

Background

- What is Lisp?
 - Originally specified in 1958 by John McCarthy , Lisp is the second-oldest high-level programming language
- Why do we use it in this class?
- Common Lisp
 - The modern, multi-paradigm, high-performance, compiled, ANSI-standardized, most prominent descendant of the long-running family of Lisp programming languages.
 - Object oriented programming and fast prototyping capabilities

Useful links

- CLISP
 - CLISP implements the language described in the ANSI Common Lisp standard with many extensions.
 - <https://clisp.sourceforge.io/>
 - You can also access it in SEASnet
- Try Lisp online:
 - <https://jscl-project.github.io/>
- Portacle (if you don't have seasnet account yet)
 - All-in-one IDE (Windows, Mac OS X, Linux)
 - <https://portacle.github.io/>
- ***Practical Common Lisp* (Book)**
 - <http://www.gigamonkeys.com/book/>



Syntax

Two fundamental pieces

- ATOM
- S-EXPRESSION.

Atom

	comment
30	; => 30
"Hello!"	; string
t	; denoting true any non-NIL value is true!
nil	; false; 0; the empty list: ()
:A	; symbol
A	; Error. Not defined

Atom

99999999999999999999999999999999	; integer
#b111	; binary => 7
#x111	; hexadecimal => 273
3.14159s0	; single
3.14159d0	; double
1/2	; ratios
#C(1 2)	; complex numbers

s-expression: super simple, super elegant

(f x y z ...)
function arguments

(+ 1 2 3 4) ; 1+2+3+4 => 10

Use **quote** or ' to prevent it from being evaluated

'(+ 1 2) ; => (+ 1 2)

(quote (+ 1 2)) ; => (+ 1 2)

'(1 2 3) ; list (1 2 3)

Basic arithmetic operations

- $(+ \ 1 \ 1)$; $\Rightarrow 2$
- $(- \ 8 \ 1)$; $\Rightarrow 7$
- $(* \ 10 \ 2)$; $\Rightarrow 20$
- $(\text{expt} \ 2 \ 3)$; $\Rightarrow 8$
- $(\text{mod} \ 5 \ 2)$; $\Rightarrow 1$
- $(/ \ 35 \ 5)$; $\Rightarrow 7$
- $(/ \ 1 \ 3)$; $\Rightarrow 1/3$
- $(+ \ \#C(1 \ 2) \ \#C(6 \ -4))$; $\Rightarrow \#C(7 \ -2)$

Booleans and Equality

(not nil) ; => T

(and 0 t) ; => T

(or 0 nil) ; => 0

(and 1 ()) ; => 0

empty list

Booleans and Equality

compare numbers

```
(= 3 3.0)           ; => T  
(= 2 1)             ; => NIL
```

compare object identity

```
(eq 3 3)             ; => T  
(eq 3 3.0)           ; => NIL  
(eq (list 3) (list 3)) ; => NIL
```

compare lists, strings

```
(equal (list 'a 'b) (list 'a 'b)) ; => T  
(equal (list 'a 'b) (list 'b 'a)) ; => NIL
```

Strings

(concatenate ^{type} 'string "Hello," "world!") ; => "Hello,world!"

(format nil "Hello, ~a" "Alice") ; returns "Hello, Alice"

(format t "Hello, ~a" "Alice") ; returns nil. formatted string goes to standard output

(print "hello") ; value is returned and printed to std out

(+ 1 (print 2)) ; prints 2. returns 3.

Variables

- global (dynamically scoped) variable
- The variable name can use any character except: ()",',` ;#|\

```
(defparameter age 35)
```

```
age ; => 35
```

```
name ; error
```

```
(defparameter *city* "LA")
```

```
*city* ; => "LA"
```

Variables

```
(defparameter age 35) ; age => 35
```

```
(defparameter age 60) ; age => 60
```

```
(defvar newage 20) ; newage => 20
```

```
(defvar newage 60) ; newage => 20
```

defvar does not change the
value of the variable!

```
(setq newage 30) ; newage => 30
```

Local variable

```
(let ( (a 1) (b 2) ) ; binding
      (+ a b)         ; body
)
```

You will **NOT** be allowed to set global variables in your homework!

let only

Lists

- Linked-list data structures
- Made of CONS pairs

```
(cons 1 2)
```

```
; => '(1 2)
```

```
(cons 3 nil)
```

```
; => '(3)
```

```
(cons 1 (cons 2 (cons 3 nil)))
```

```
; => '(1 2 3)
```

```
(list 1 2 3)
```

```
; => '(1 2 3)
```

```
(cons 4 '(1 2 3))
```

```
; => '(4 1 2 3)
```

```
(cons '(4 5) '(1 2 3))
```

```
; => ?
```


Lists

```
(cons 1 (cons 2 (cons 3 nil))) ; => '(1 2 3)
(list 1 2 3)                   ; => '(1 2 3)
(cons 4 '(1 2 3))              ; => '(4 1 2 3)
(cons '(4 5) '(1 2 3))        ; => '((4 5) 1 2 3)
```

```
(append '(1 2) '(3 4))         ; => '(1 2 3 4)
(append 1 '(1 2))              ; ERROR!
(concatenate 'list '(1 2) '(3 4)) ; => '(1 2 3 4)
```

```
(car '(1 2 3 4))               ; => 1
(cdr '(1 2 3 4))               ; => '(2 3 4)
```

car and cdr should be used for list

Functions

- Define a function

```
(defun hello (name) (format nil "Hello, ~A" name))
```

- Call the function

```
(hello "Bob") ; => "Hello, Bob"
```

Control Flow

```
(if (equal *name* "bob") ; test expression
    "ok"                 ; then expression
    "no")                ; else expression
```

- Chains of tests: cond

```
(cond ((> *age* 20) "Older than 20")
      ((< *age* 20) "Younger than 20")
      (t "Exactly 20"))
```

```
(cond ((> *age* 20) "Older than 20")
      ((< *age* 20) "Younger than 20")) ; returns NIL when *age*=20
```

Recursion - factorial

```
(defun factorial (n)
  (if (< n 2)
      1
      (* n (factorial (- n 1)))
  )
)
```

; returns 1 when $n < 2$
; when $n \geq 2$

(factorial 5) ; => 120

Recursion – compute list length (top-level)

'((a b) (c (d 1)) e) => 3

```
(defun listlength (x)
  (if (not x)                ; base case: empty list
      0
      (+ (listlength (cdr x)) 1)
  )
)
```

'(1 2 3 4) -> '(2 3 4)

Recursion – compute list length (deep)

'((a b) (c (d 1)) e) => 6

```
(defun deeplength (x)
  (cond ((not x) 0) ; empty list. returns 0
        ((atom x) 1) ; atom. returns 1
        (t (+ (deeplength (car x)) ; else
                (deeplength (cdr x))
              )
        )
  )
)
```

Recursion – check if list contains an element

```
(defun contains (e x)
  (cond ((not x) nil)
        ((atom x) (equal e x))
        (t (or (contains e (car x)) (contains e (cdr x)))))
  )
)
```

```
(contains 'a '((b a) (1 e c)))
```

Recursion – check if list contains a number

Consider this case: '((a b) (c (d 1)) e)

```
(defun contains_number (x)
  (if (atom x)                ; NIL if x is a list
      (numberp x)             ; numberp: check if x is a number
      (or (contains_number (car x))
          (contains_number (cdr x)) ; recursively flatten
      )
  )
)
```


Recursion – find kth element (top-level)

```
(defun find_kth (k x)
  (if (= k 1)
      (car x)
      (find_kth (- k 1) (cdr x))
  )
)
```

How do we find kth element in the flattened list?

3, '((a b) (c (d 1)) e) => c

Recursion – delete kth element

```
(defun delete_kth (k x)
  (if (= k 1)
      (cdr x)
      (cons (car x)
             (delete_kth (- k 1) (cdr x))
            )
  )
)
```

Recursion

```
(defun x () (x))
```

This runs forever!

Iteration

```
(loop for x in '(1 2 3 4 5)
      for y in '(1 2 3 4 5)
      collect (+ x y)
)
```

; => (2 4 6 8 10)

```
(loop for x in '(1 2 3 4 5)
      for y in '(1 2 3 4)
      collect (+ x y)
)
```

; => ?

Iteration

```
(loop for x in '(1 2 3 4 5)
      for y in '(1 2 3 4 5)
      collect (+ x y)
)
```

; => (2 4 6 8 10)

```
(loop for x in '(1 2 3 4 5)
      for y in '(1 2 3 4)
      collect (+ x y)
)
```

; => (2 4 6 8)

Iteration

```
(loop for x from 1 to 5  
      for y = (* x 2)  
      collect y  
)
```

; => (2 4 6 8 10)

How do we compute factorial in loop?

Iteration

```
(setf fact 1)
(defun factorial(n)
  (loop for x from 2 to n
        do(setf fact (* x fact))
        )
  (print fact)
)
```

Recursion is much more natural