Basic Introduction to Lisp

Based on Common Lisp

yujinglei_1222#yahoo.com.cn

gadmyth#gmail.com

Slide Context

- For Who?
  - Newbies for lisp
  - With a little knowledge about lisp
- For What?
  - Basic lisp knowledge
  - Not very deep topics
- How?
  - A lot of examples
  - Run codes in Lisp Environment

Basic Introduction to Lisp

- LISt Progressor (code ↔ data)

  (special-form data data data ….)

  (quote data data data data)

  (eval '(special-form data data data))

- Major Lisp Dialect:

  Scheme, Common Lisp, Clojure(based on JVM)

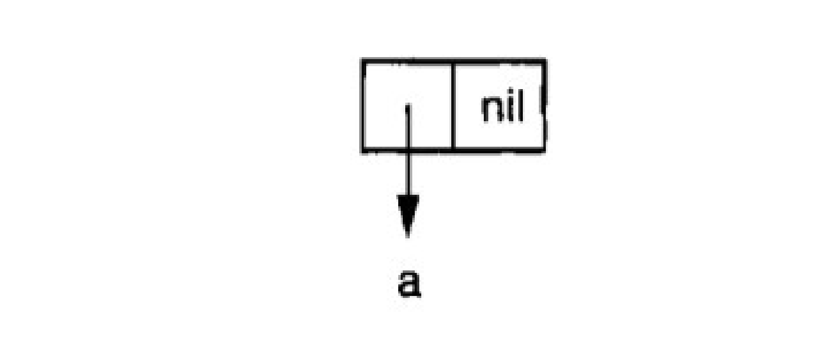
Quote & Eval

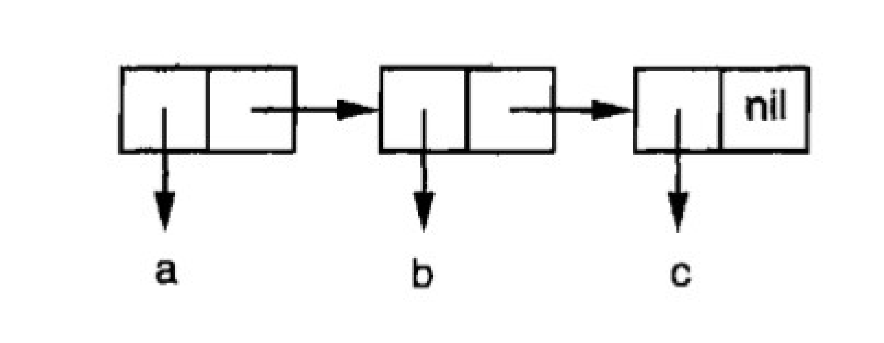
- cl-user> 2
  - ;;; return 2, 2 is evaluated as 2
- cl-user> a
  - ;;; a is a variable with value “table”, will return “table”
- cl-user> (quote a)
  - ;;; quote makes a variable avoid evaluating, return A, also 'a makes the same effect
- cl-user> (eval (quote a))
  - ;;; return “table”

What is list?

- List's Two Parts & List Constructor
  - car, cdr, cons
  - car → first, cdr → rest
  - (cons [car part] [cdr part])
  - (x . y) → (cons x y), '() → nil
  - '(a) → (cons a nil), '(a b c) → (cons a (cons (b (cons c nil)))

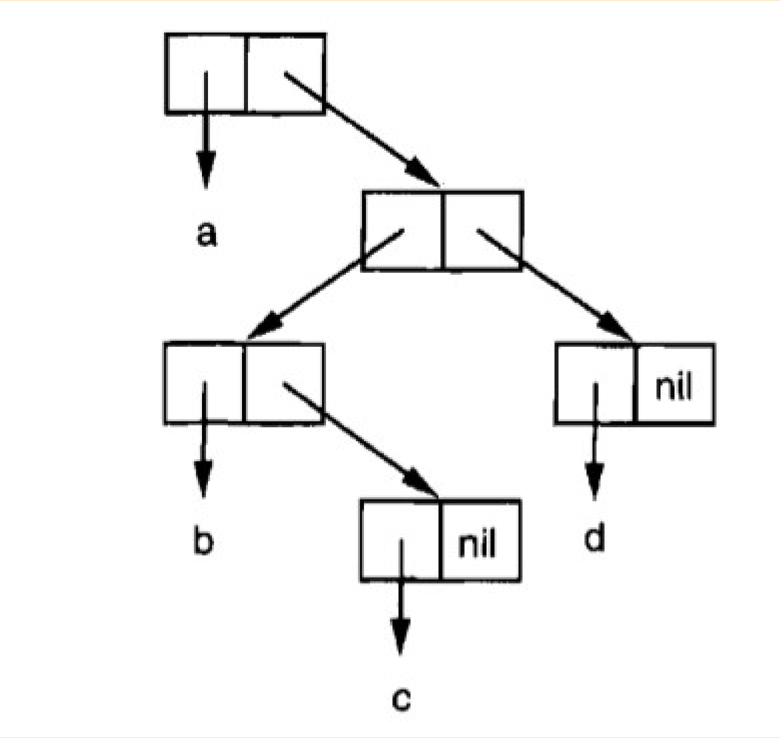
→ (list a b c)





What is List?

- Symbolic expression
  - data S-exp = Atom | Cons (S-exp, S-exp)
  - Atom: 1, 2, 3, t, nil → self evaluated
  - Cons: (1 . 2)
- S-expression forms AST
  - Normal exp: 1 + 2 * (7 – 3)
  - S-exp: (+ 1 (* 2 (- 7 3)))



Lisp Basic

- Variable
  - (defparameter   *fruit*   '(”apple” ”banana” ”orange”))
  - (setf a ”aaaaaaaaaa”)
  - (defvar *host* ”127.0.0.1”)
  - (defconstant QWERTY 0)
- Quote & back-quote – make list, define data
  - '(1 2 3 4)
  - `(1 2 3 ,a)   ;;; a is 1024
  - `(1 2 3 ,@rest)   ;;; rest is '(4 5 6 7)

Function

- Function definition
    - Defun, Lambda
    - Labels & flet
- Function parameters
    - Varying numbers of arguments
    - Optional arguments
    - Keywords arguments
- Multiple returning value
- Function as Data

Define a Function

-
  ```
  (defun a ()
  (format t ”~a, Hello, world!” a)) ;;; (setf a ”aaaaaaa”)
  ```

-
  ```
  (let ((name ”Ted”))
   (flet/label ((hello (n)
                  (format t ”Hello, ~a~%” n)))
       (hello name)))
  ```

Functon Keywords

- (defun foo

  (&key ((:apple a)) ((:box b) 0) ((:cat c) 0 c-supplied-p))

  (format t "apple:~a, box: ~a, cat: ~a, cat is setted? ~a~%" a b c c-supplied-p))
- ((:keyword alias) default-value [is-setted?])
- Check the answer
  - (foo)
  - (foo :apple "ack")
  - (foo :box 1001 :apple "ack")
  - (foo :box 1001 :charlie 'yes :apple "ack")

Multiple Values & Binding

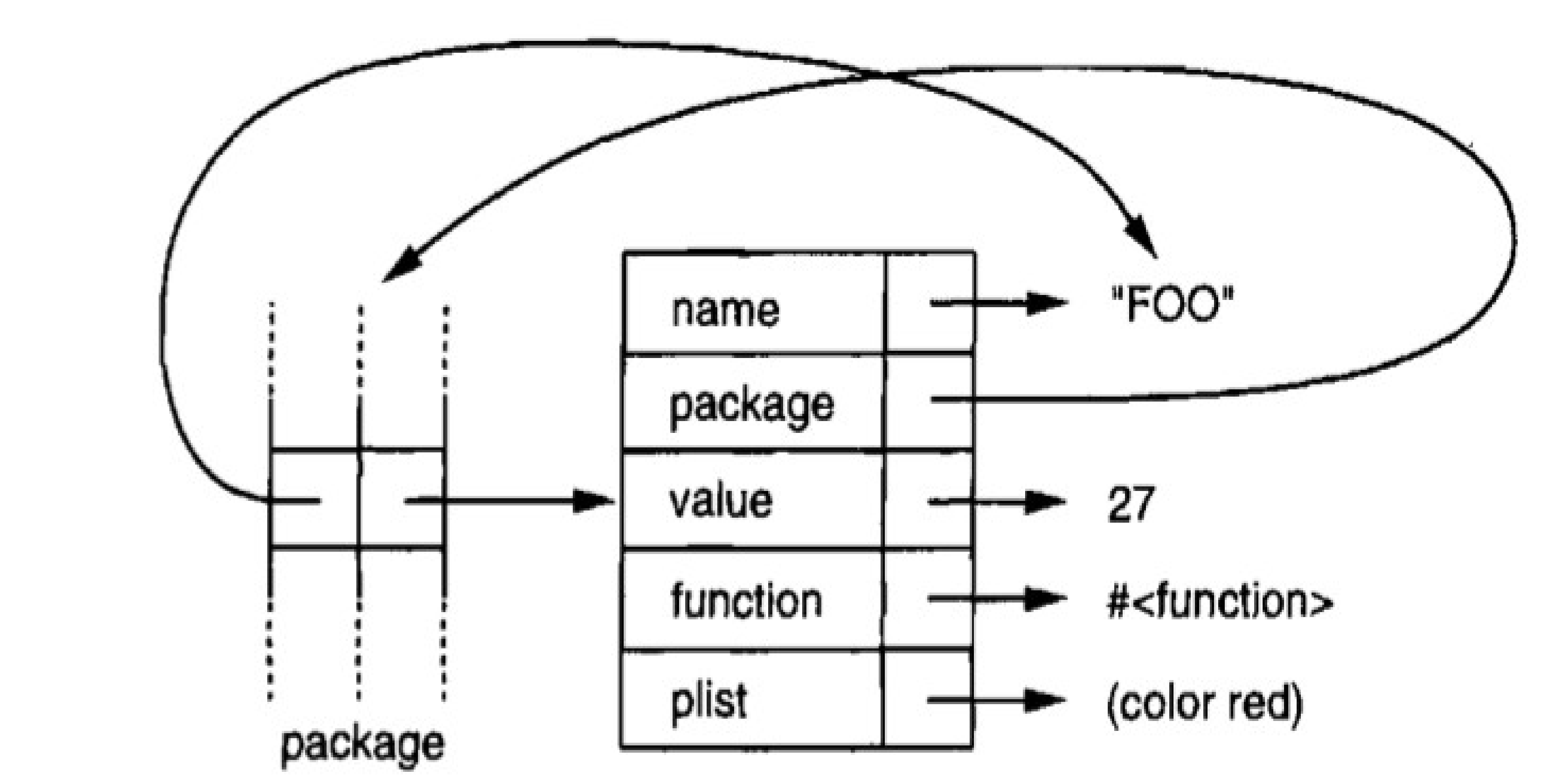
- Input: Return Multiple Values

```
(values 1 2 3) ;; will return 1, 2 and 3

(floor 1.23) ;; will return 1 and 0.23
```

- Output: Binding Multiple Values

```
(multiple-value-bind (f r) (floor 1.23)

   (format t "~a r ~a~%" f r))
```

Symbol



Symbol

- Example
  - (defparameter makiyo "Beatiful girl")
  - (format t "~a~%" (symbol-name 'makiyo))
  - (format t "~a~%" (symbol-package 'makiyo))
  - (defun makiyo ()

    (format t "Hi, my name is makiyo, I'm a beautiful girl!"))
  - (format t "~a~%" (symbol-value 'makiyo))
  - (format t "~a~%" (function makiyo)) ;;; you can use #'

Function as Data

- Funcall, Apply

```
(defun f (a) (+ 1 a))

(funcall (function f) 4)

(defun g (a b) (+ 1 a) b)

(apply #'g (list 3 4)

(apply #'g 3 '(4))
```

- High order functions
  - map[car], reduce, filter, find, remove-if
  - Complement,

High Order Functions

- (find 20 '((a 10) (b 20) (c 30) (d 40)) :key #'cadr)
- (remove-if-not #'alpha-char-p
  #("foo" "bar" "1baz") :key #'(lambda (x) (elt x 0)))

Recursion in Lisp

- S-exp makes recursion easy
  - Count the number of a list

```
(defun len! (lst)
  (if (null lst)
      0 ;;; nil list is zero length
      (+ 1 (len! (cdr lst)))))
```

Tail Recursion

- If the lisp is soooooo long → stack overflow
- Tail recursion

```
(defun len! (lst)
   (labels ((len-! (li len) ;;; define the inner fn with a counter
        (if (null li)
           len ;;; returns the length
           (len-! (cdr li) (+ 1 len))))) ;;; recursive part
     (len-! lst 0)))
```

Quick Sort

```
(defun merge-list (left mid right)
  (append left (list mid) right))
(defun quick-sort (lst)
  (cond ((null lst) nil)
      (t (let* ((mid (car lst))
                 (rest (cdr lst))
                 (left (remove-if #'(lambda (a) (> a mid)) rest))
                 (right (remove-if #'(lambda (a) (<= a mid)) rest)))
             (merge-list (quick-sort left) mid (quick-sort right))))))
```

*** Make this to be tail recursion ***

```
quicksort [] = []
quicksort (s:xs) = quicksort [x|x <- xs,x < s] ++ [s] ++ quicksort [x|x <- xs,x >= s]
```

Macro

- Macro expansion time/Compile time → runtime
- Back-quote generating code

make program as data: list, `

program → data

- (list 1 2 3 4) → '(1 2 3 4)
- `(1 2 3 4) → '(1 2 3 4)
- `(format t ”hello, world”) → '(format t ”hello, world”)

Macro

- (defun fn (people)

  (hello people))
- (defun fn (people)

  (format t “hello, ~a~%” people))
- (hello people) → (format t “hello, ~a~%” people)
- (defmacro hello (p)

  `(format t ”hello, ~a~%” ,p))

Define your Macro

- (if [condition]

  ([condition is true, do one thing])

  ([condition is false, do one thing else]))
- Do more things when the condition is true
  - Use **progn** wrapping out
  - Define your own 'when' macro
- You can also define unless
  - Do more thing when the condition is false

Macro

- How to Use Progn:

-
  ```
  (if (= 3 (- 4 1))
   (progn
     (format t ”Yes, they are equal”)
     (expt 3 5))) ;;; will return 243
  ```
- Your When likes as this:

```
(when [condition]
    ([do 1])
    ([do 2])
    ([do...])
     ……))
```

Macro

- Implement 'When'

```
(defmacro when1 (condition &body body)
    `(if ,condition
        (progn
            ,@body)))
```

- Check Macro
  - expand it and see what it is
  - macroexpand & macroexpand-1

Pratical Use Of Macro

- DSL (Domain Special Language)
  - Define your own grammar
  - Generate html
  - AOP (Aspect Oriented Program)
- AOP example (Macro is tricky)
  - Print the name of every function

```
(defmacro defunction (name params &body body)
 `(defun ,name ,params
     (format t "** fn-name: ~a **~%" ',name)
     ,@body))
```

Macro

- Too many)))))))))))))))))))).......
  - Clojure: ->, -?>
  - We “decrease” the number of ')' with macro
    - Easy to read
  - (defmacro -> (data &body body)

```
 `(reduce #'(lambda (val code)
                    (apply (car code) val (cdr code)))
                    ',body
                     :initial-value ,data))
```

Macro

-
  ```
  (-> 1
    (+ 3)
    (+ 5)
    (* 6))  ;; will return 54
  ```

-
  ```
  (* (+ (+ 1 3) 5) 6) ;; will return 54
  ```

To be Continued...

- Collections (list, vector, string, array, hashtable)
- Struct define
- CLOS (MOP, OOP)
- Format tricky, Loop tricky
- CPS
- Read Macro
- Pattern Match in Common Lisp
- REPL, Compling, Running, Evaluating...
- ……

Q & A

Thank you!