A Brief Introduction to Common Lisp

David Gu

Schloer Consulting Group

david_guru@gty.org.in

A Brief History

- Originally specified in 1958, Lisp is the second-oldest high-level programming language in widespread use today; only Fortran is older (by one year).
- Lisp stands for LISt Processing (while Fortran stands for *FORmula TRANslator*).
- 1958 ~ 1980: Various dialects and systems, *MacLisp*, *InterLisp*, and *Lisp Machines*.

A Brief History

- 1975 ~ 1980: **Scheme**, the first dialect choosing lexical scope, developed in MIT.
- 1980 ~ 1990: **Common Lisp**, an industry-level language, published in ANSI standard.
- 1990 ~ Now: Various implementation for Common Lisp and Scheme; More 3rd party libraries; A new ~~successful~~ dialect Clojure.

First Impression

```c
// C Code
int factorial(int n){
  if(n==0)
    return 1;
  else
    return n * factorial(n-1);
}
```

```lisp
;; Common Lisp Code
(defun factorial (n)
  (if (= n 0)
      1
      (* n
         (factorial (- n 1)))))
```

First Impression: Evolution

```
// C code, using tail recursion
int factorial_helper(int result, int count){
  if(count == 0) return result;
  else
    return factorial_helper(result*count, count-1);
}
int factorial(int n){
  return factorial_helper(1, n);
}
```

First Impression: Evolution

```lisp
;; Common Lisp code, using tail recursion
(defun factorial (n)
  (declare (optimize (speed 3)))
  (labels ((iter (result count)
             (if (= count 0)
                 result
                 (iter (* result count) (1- count)))))
    (iter 1 n)))
```

First Impression: Final 'Product'

```
;;;; can even use a function called ‘disassemble’
;;;; to check the assemble code.
(defun factorial (n)
  (declare (optimize (speed 3)))
  (labels ((iter (result count)
                 (declare (type fixnum result count))
                 (if (= count 0)
                   result
                   (iter (the fixnum (* result count))
                         (the fixnum (- count 1))))))
    (iter 1 n)))
```

Multi-Paradigms: Functional

Lambda(λ)

```
((lambda (x y) (+ x y)) 1 2)
=> 3
```

Map

```
(map 'list #'(lambda (x) (1+ x)) (list 0 1 2 3))
=> (1 2 3 4)
```

Multi-Paradigms: Functional

Filter

```
=> (1 3 5)
```

Fold(foldr in ML)

```
(reduce #'+ (list 1 2 3 4 5))
=> 15
```

Curried Function, Lazy Evaluation, and more...

Multi-Paradigms: Imperative

- Common Lisp does provide imperative operators like:
  `(setf x 10) ⇔ x := 10`
- And for functional functions, Common Lisp also provides their 'destructive' version:
  - map ⇔ map-into
  - filter (remove-if-not) ⇔ delete-if-not
- And even more: goto, for, while...

Multi-Paradigms: OOP

- Common Lisp has its own implementation for object oriented programming, which is called CLOS.
- Unlike Java or C++, Common Lisp uses an approach called Generic Function instead of Message Passing.
- Basically, all the methods belongs to generic function instead of a specific class.

Multi-Paradigms: OOP

For example, if there's a human class and there's a method called `speak`, then I create an instance of human called 'me':

- In message passing style: `me.speak("Hello")`
- In generic function style: `speak(me, "Hello")`

Multi-Paradigms: OOP

```
;;; CLOS Example
(defclass human ()
  ((name :initform "Anonymous"
         :initarg :name
         :accessor name)))

(defclass man (human) ())

(defclass woman (human) ())

(defgeneric say-love (human)
  (:documentation "A human says: 'I love you!'"))

(defmethod say-love ((obj human))
  (format t "~a says: 'I love you!'~%" (name obj)))

(defmethod say-love :after ((obj man))
  (format t "In ~a's mind: Hmm... Why should I say that?~

(defvar remeo (make-instance 'man :name "Romeo"))

(defvar juliet (make-instance 'woman :name "Juliet"))
```

Multi-Paradigms: OOP

A 'men' will have different behavior.

```
CL-USER> (say-love juliet)
Juliet says: 'I love you!'
NIL
CL-USER> (say-love remeo)
Romeo says: 'I love you!'
In Romeo's mind: Hmm... Why should I say that?
NIL
```

It’s called method combination.

What Makes Lisp Special?

1. S-Expression

2. Macro

S-Expression

In Common Lisp, a s-exp would be like:

s-exp : (op s-exp1 s-exp2 ...)

op: a function | a macro | a special form

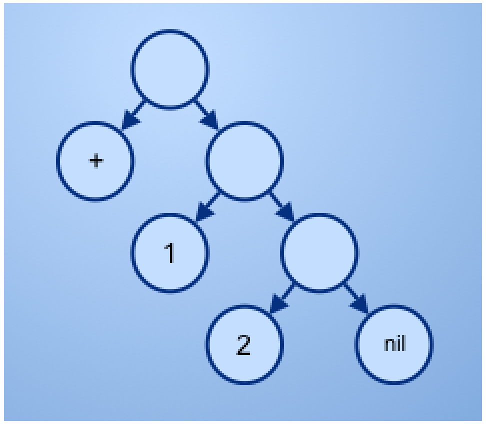
And interestingly, a s-exp could be ‘made’ like this:

```
(cons 1 2) => (1 . 2)
(cons 1 (cons 2 nil)) => (1 . (2 . nil)) => (1 2)
(cons '+ (cons 1 (cons 2 nil))) => (+ 1 2)
(eval (cons '+ (cons 1 (cons 2 nil)))) => 3
```

S-Expression

A s-exp looks like a linked list but actually a tree.

```
(car (list '+ 1 2)) => '+
(cdr (list '+ 1 2)) => (1 2)
```



Writing s-expressions is actually writing the **Abstract Syntax Tree(AST)**.

S-Expression: Benefits

- There will be no *lexical analysis* because all the codes already are the AST.
- There will be no need to worry about the *Operator Precedence*.
- Very convenient to represent data structures like **trees** and **graphs**.

Macro:
Why Lisp is Called Programmable Programming Language?

- Unlike functions, macros will be expanded first before evaluated;
- After expanding finished, the whole s-expression generated by macro will be evaluated;
- So a macro is actually a function that transforms arguments to s-expressions.

Macro: A Simple Example

In this case, it acts like a inline function.

```
CL-USER> (defun add (x y) (+ x y))
ADD
CL-USER> (defmacro add-1 (x y) `(+ ,x ,y))
ADD-1
CL-USER> (add 10 20)
30
CL-USER> (add-1 10 20)
30
CL-USER> (macroexpand '(add-1 10 "string"))
(+ 10 "string")
T
```

Macro:
A Real (but a little silly) Example

Macros can do something that a function can never do.

```
(defmacro delay (thing)
  ;; return a thunk
  `(lambda () ,thing))

(defun force (thunk) (funcall thunk))

(defmacro my-if (test then else)
  `(block nil
     (and ,test
          (return ,then))
     ,else))

(defun factorial (n)
  (my-if (= n 0) ;; test
         1 ;; then
         (* n (factorial (- n 1))))) ;; else
```

Macro: A Real Example

Macros can do something that a function can never do.

```
CL-USER> (delay (loop))
#<FUNCTION (LAMBDA ()) {1003AF4EBB}>
CL-USER> (macroexpand '(delay (loop)))
#'(LAMBDA () (LOOP))
T
CL-USER> (force (delay (+ 1 2)))
3
CL-USER> (factorial 20)
2432902008176640000
CL-USER> (macroexpand
          '(my-if (= n 0) ; test
            1 ; then
            (* n (factorial (- n 1))))) ; else
(BLOCK NIL (AND (= N 0) (RETURN 1)) (* N (FACTORIAL (- N
T
```

Macro: Define New Syntax

```
;;;; define new syntax by macro
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))

(defmacro for (var start end &body body)
  (let ((gend (gensym)))
    `(do ((,var ,start (1+ ,var))
          (,gend ,end))
         ((> ,var ,gend))
       ,@body)))
```

```
CL-USER> (let ((x 5))
           (while (> x 0)
             (format t "~d " x)
             (decf x)))
5 4 3 2 1
NIL
CL-USER> (for i 1 5 (format t "~d " i))
1 2 3 4 5
NIL
```

Macro: Even Let Evaluation Happens During 'Compiling' Time

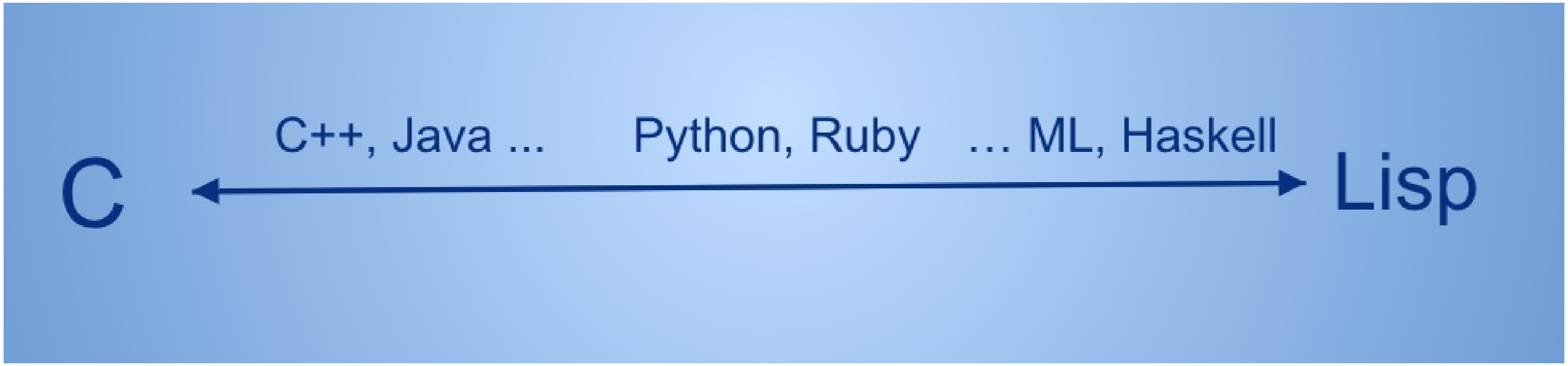
As mentioned before, a macro will be expanded before evaluated,

even before compiled.

```
CL-USER> (defmacro avg (&rest numbers)
           `(/ (+ ,@numbers) ,(length numbers)))
AVG
CL-USER> (avg 1 2 3 4 5 6 7 8 9 10)
11/2
CL-USER> (macroexpand '(avg 1 2 3 4 5 6 7 8 9 10))
(/ (+ 1 2 3 4 5 6 7 8 9 10) 10)
T
```

'Counting how many numbers' happens before run time.

Lisp: An ‘Edge’ of Programming Languages



Thanks for Watching!

> We toast the Lisp programmer who pens his thoughts within nests of parentheses.
> -- Alan J. Perlis
> <SICP>'s foreword