Common LISP

An short introduction on most interesting parts

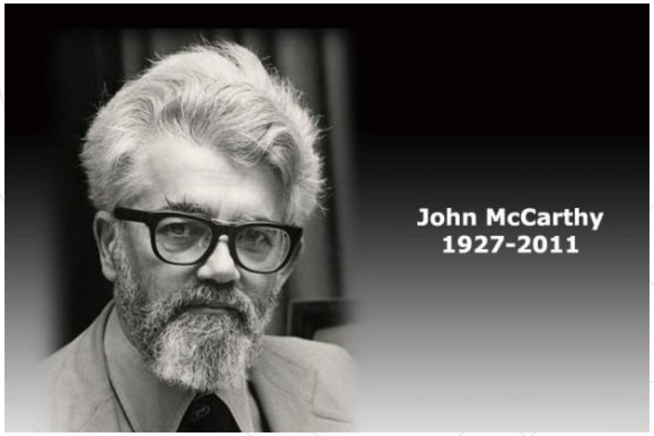
On Principles of Programming Languages

Chun Tian (binghe)

CONTENTS

The History of LISP

- LISP 1 on IBM 704: 1958-1959
- Lisp 1.5 to PDP-6 Lisp: 1960-1965
- MacLisp and InterLisp (late 1960's - early 1980's)
- The Early 1970's: various Lisp systems
- Lisp Machines: 1973-1980
- Scheme: 1975-1980
- Prelude to Common Lisp (CLTL1): 1980-1984
- Standards Development (CLTL2): 1984-1990
- Modern commercial platforms: 1987-1999
- Modern open source platforms: 1999-2009
- QuickLisp and 3rd-party libraries: 2009-2012



FORTRAN	= **FOR**mula **TRAN**slator
LISP	= **LIS**t **P**rocess

LISP Features

- Fully interactive user interface.
- Dynamic linking.
- Built-in dynamic storage management.
- Built-in facilities for manipulating lists and symbolic names.
- Standard internal representation of programs as list data.
- Powerful macro facility.
- Extensive runtime environment.
- Minimal syntax (as compared with ALGOL-like languages).
- Typeless variables.

Common Lisp Features

- Fully lexically scoped variables.
- A rich set of numberical data types.
- A rich set of numerical primitives.
- A string data type.
- Arrays and vectors.
- Bit and field manipulation.
- Dynamic non-local exits, and user-controllable error handling.
- Built-in hash facility.
- User-defined data types.
- Stream-based I/O with a relatively implementation-independent interface.
- Formatting and pretty-printing utilities.

EXAMPLE: A FACTORIAL FUNCTION

Example: a factorial function in C

```
int fact (int n) {
  int sofar = 1;
  while (n>0) sofar *= n--;
  return sofar;
}
```

Common Lisp (version 1)

```
(defun fact (n)
  (let ((sofar 1))
    (loop while (> n 0) do
      (setq sofar (* sofar n))
      (decf n))
    sofar))
```

Common Lisp (version 2)

```
(defun fact (x)
  (if (<= x 0)
      1
    (* x (fact (- x 1)))))
```

Common Lisp (version 3)

```
(defun fact (x)
  (labels ((iter (n acc)
             (if (<= n 0)
                 acc
               (iter (- n 1) (* n acc)))))
    (iter x 1)))
```

Common Lisp (version 4)

```
(defun fact (x)
  (declare (optimize speed))
  (labels ((iter (n acc)
             (declare (type fixnum n acc))
             (if (<= n 0)
                 acc
               (iter (the fixnum (- n 1))
                     (the fixnum (* n acc))))))
    (iter x 1)))
```

Multi-Paradigms

- Imperative Programming (for, if, goto, ...)
- Functional Programming (lambda)
- Object-oriented programming (CLOS & MOP)
- Generic programming (macro)
- Aspect-oriented programming (AspectL)
- Rule-based and Logic progamming (Knowledge-Works & Common Prolog)

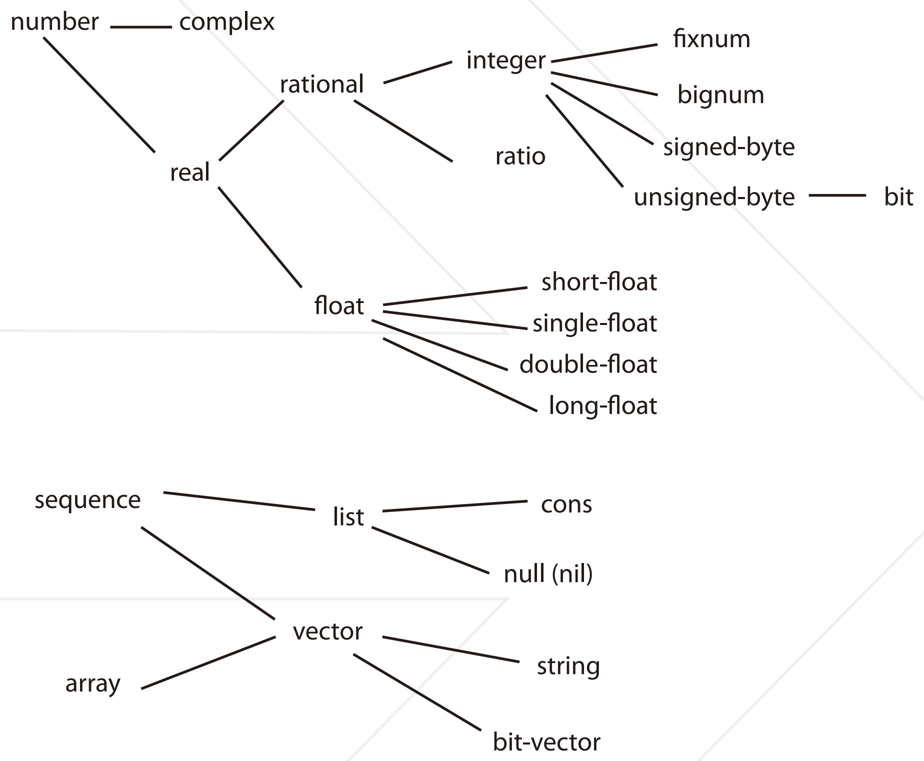
GOTO in C

```
int fact (int n) {
  int sofar = 1;
a:
  sofar *= n--;
  if (n>0) goto a;
  return sofar;
}
```

GOTO in Common Lisp

```
(defun fact (n)
  (let ((sofar 1))
    (tagbody
     a
      (setq sofar (* sofar n))
      (decf n)
      (if (> n 0) (go a)))
    sofar))
```

DATA TYPES IN COMMON LISP



data type = creation + operation

Data Types

- Numbers
- Characters
- Symbols
- Lists
- Arrays
- Hash tables
- Readtables
- Packages
- Pathnames
- Streams
- Random-states
- Structures
- Functions
- Conditions
- Classes
- Methods
- Generic Functions

Forms and Functions

- Forms: (+ 3 4)
- Self-Evaluating Forms: numbers, characters, strings and bit-vectors.
- Variables
- Special Forms
- Macros
- Function calls
- Named Functions: defun
- Lambda-Expressions

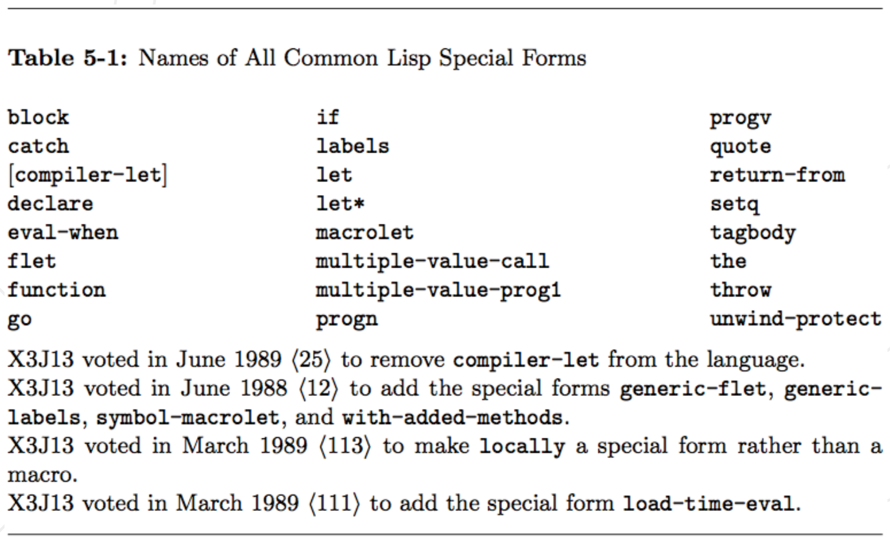
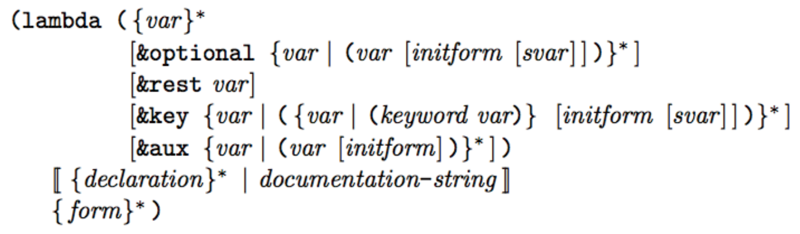


Table 5-1: Names of All Common Lisp Special Forms

`block`	`if`	`progv`
`catch`	`labels`	`quote`
`[compiler-let]`	`let`	`return-from`
`declare`	`let*`	`setq`
`eval-when`	`macrolet`	`tagbody`
`flet`	`multiple-value-call`	`the`
`function`	`multiple-value-prog1`	`throw`
`go`	`progn`	`unwind-protect`

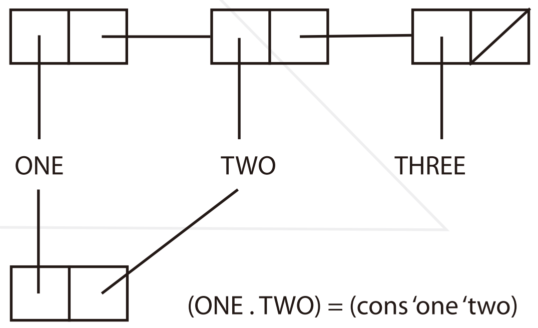
X3J13 voted in June 1989 ⟨25⟩ to remove **`compiler-let`** from the language.
X3J13 voted in June 1988 ⟨12⟩ to add the special forms **`generic-flet`**, **`generic-labels`**, **`symbol-macrolet`**, and **`with-added-methods`**.
X3J13 voted in March 1989 ⟨113⟩ to make **`locally`** a special form rather than a macro.
X3J13 voted in March 1989 ⟨111⟩ to add the special form **`load-time-eval`**.



```
(lambda ({var}*
              [&optional {var | (var [initform [svar]])}*]
              [&rest var]
              [&key {var | ({var | (keyword var)} [initform [svar]])}*]
              [&aux {var | (var [initform])}*])
   ⟦ {declaration}* | documentation-string ⟧
   {form}* )
```

```
((lambda (a b) (+ a b)) 1 2) => 3
```

(ONE TWO THREE)



(ONE . TWO) = (cons 'one 'two)

```
(defun good-reverse (lst)
  (labels ((rev (lst acc)
             (if (null lst)
                 acc
               (rev (cdr list)
                    (cons (car lst) acc)))))
    (rev lst nil)))

> (setq lst '(a b c))
(A B C)
> (good-reverse lst)
(C B A)
> lst
(A B C)
```

Functions on Lists

For the sake of example, assume we have the following assignments:

```
(setf x '(a b c))
(setf y '(1 2 3))
```

The most important functions on lists are summarized here. The more complicated ones are explained more thoroughly when they are used.

`(first x)`	+a	first element of a list
`(second x)`	⇒ b	second element of a list
`(third x)`	⇒ c	third element of a list
`(nth 0 x)`	+a	nth element of a list, 0-based
`(rest x)`	⇒ (b c)	all but the first element
`(car x)`	⇒ a	another name for the first element of a list
`(cdr x)`	⇒ (b c)	another name for all but the first element
`(last x)`	⇒ (c)	last cons cell in a list
`(length x)`	⇒ 3	number of elements in a list
`(reverse x)`	⇒ (c b a)	puts list in reverse order
`(cons 0 y)`	⇒ (0 1 2 3)	add to front of list
`(append x y)`	⇒ (a b c 1 2 3)	append together elements
`(list x y)`	⇒ ((a b c) (1 2 3))	makeanewlist
`(list* 1 2 x)`	⇒ (1 2 a b c)	append last argument to others
`(null nil)`	⇒ T	predicate is true of the empty list
`(null x)`	+nil	...and false for everything else
`(listp x)`	⇒ T	predicate is true of any list, including nil
`(listp 3)`	+nil	...and is false for nonlists
`(consp x)`	⇒ **t**	predicate is true of non-nil lists
`(consp nil)`	⇒ nil	...and false for atoms, including nil
`(equal x x)`	⇒ **t**	true for lists that look the same
`(equal x y)`	⇒ nil	...and false for lists that look different
`(sort y #'>)`	⇒ (3 2 1)	sort a list according to a comparison function
`(subseq x 1 2)`	⇒ (B)	subsequence with given start and end points

Map/Reduce: a sample use

Target: $a_1^2+a_2^2+a_3^2+...+a_n^2$=?

```
(defun square-sum (seq)
  (labels ((square (x) (* x x)))
    (reduce #'+
            (map 'list #'square seq))))

> (setq a '(1 2 3 4))
(1 2 3 4)

> (square-sum a)
30

> (setq a #(1 2 3 4))
#(1 2 3 4)

> (square-sum a)
30
```

Another version

```
(defun square-sum (seq)
  (reduce #'+
    (map 'list #'(lambda (x) (* x x)) seq)))
```

Question: $a_1^2+a_2^2+a_3^2+...+a_n^2$=?

The CLOS Approaches

- The Layered Approach
- The Generic Function Approach
- The Multiple Inheritance Approach
- The Method Combination Approach
- First-Class Objects

The first level of the Object System provides a programmatic interface to object-oriented programming. This level is designed to meet the needs of most serious users and to provide a syntax that is crisp and understandable. The second level provides a functional interface into the heart of the Object System. This level is intended for the programmer who is writing very complex software or a programming environment. The first level is written in terms of this second level. The third level provides the tools for the programmer who is writing his own object-oriented language. It allows access to the primitive objects and operators of the Object System. It is this level on which the implementation of the Object System itself is based.

The Common Lisp Object System is based on generic functions rather than on message-passing. This choice is made for two reasons: 1) there are some problems with message-passing in operations of more than one argument; 2) the concept of generic functions is a generalization of the concept of ordinary Lisp functions.

Generic functions are first-class objects in the Object System. They can be used in the same ways that ordinary functions can be used in Common Lisp. A generic function is a true function that can be passed as an argument, used as the first argument to funcall and apply, and stored in the function cell of a symbol. Ordinary functions and generic functions are called with identical syntax.

COMMON LISP OBJECT SYSTEM (CLOS), PART II

Multimethods VS. Method Overloading

```
public class A {
  public void foo(A a) { System.out.println("A/A"); }
  public void foo(B b) { System.out.println("A/B"); }
}
public class B extends A {
  public void foo(A a) { System.out.println("B/A"); }
  public void foo(B b) { System.out.println("B/B"); }
}

public class Main {
  public static void main(String[] argv) {
    A obj = argv[0].equals("A") ? new A() : new B();
    obj.foo(obj);
  }
}

$ java com.gigamonkeys.Main A
A/A

$ java com.gigamonkeys.Main B
B/A
```

CLOS Demo Code

```
(defclass a () ())
(defclass b (a) ())

(defgeneric foo (a b))

(defmethod foo ((a a) (a a)) "A/A")
(defmethod foo ((a a) (b b)) "A/B")
(defmethod foo ((b b) (a a)) "B/A")
(defmethod foo ((b b) (b b)) "B/B")

> (defvar a (make-instance 'a))
> (defvar b (make-instance 'b))

> (foo a a)
"A/A"
> (foo b b)
"B/B"
> (foo a b)
"A/B"
> (foo b a)
"B/A"
```

COMMON LISP CONDITION SYSTEM

Java-style exception handling

```
try {
  doStuff();
  doMoreStuff();
} catch (SomeException se) {
  recover(se);
}
```

or this in Python:

```
try:
  doStuff()
  doMoreStuff()
except SomeException, se:
  recover(se)
```

In Common Lisp you'd write this:

```
(handler-case
    (progn
      (do-stuff)
      (do-more-stuff))
  (some-exception (se)
    (recover se)))
```

Restart

```
(defun foo (n)
  (+ 100 (restart-case (/ 1 n)
           (use-value (x) x))))

> (foo 1)
101

> (foo 0)

Error: Division-by-zero caused by / of (1 0).
  1 (continue) Return a value to use.
  2 Supply new arguments to use.
  3 USE-VALUE
  4 (abort) Return to level 0.
  5 Return to top loop level 0.

> (handler-bind ((division-by-zero
                   #'(lambda (ignore) (use-value 0))))
    (foo 0))
100
```

MAINTENANCE OF LISP IMAGE

- Redefine variables: defvar v.s. defparameter
- Redefine functions
- Redefine macros
- Delete unused objects
- Change the classes of instances
- Upgrade instaces for redefined class

A example

Suppose a class definition of triangle:

```
(defclass triangle (shape)
  ((a :reader side-a :initarg :side-a)
   (b :reader side-b :initarg :side-b)
   (c :reader side-c :initarg :side-c)))
```

was changed to:

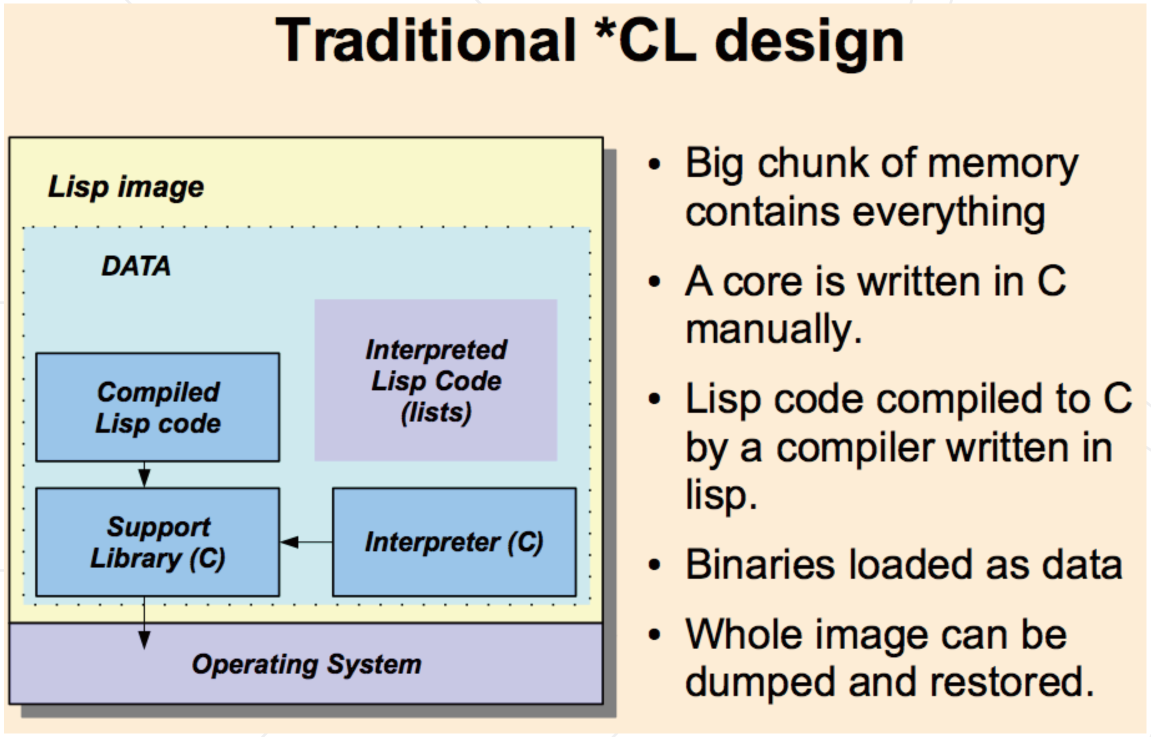
```
(defclass triangle (shape)
  ((a :reader side-a :initarg :side-a)
   (b :reader side-b :initarg :side-b)
   (angle-C :reader angle-C :initarg :angle-C)))
```

```
(defmethod update-instance-for-redefined-class :after
           ((instance triangle)
            added-slots discarded-slots
            plist &rest initargs)
  (declare (ignore initargs))
  (if (and (member 'c discarded-slots)
           (member 'angle-C added-slots))
      (setf (slot-value instance 'angle-C)
            (three-sides-to-angle
              (getf plist 'c)
              (side-a instance)
              (side-b instance)))))

(defmethod side-c ((tri triangle))
  (third-side (side-a tri)
              (side-b tri)
              (angle-C tri)))

;; Algorithm is: c^2 = a^2 + b^2 - 2ab(cos C)
(defun third-side (a b angle-C)
  (sqrt (- (+ (expt a 2)
              (expt b 2))
           (* 2 a b (cos angle-C)))))
```

RUNTIME SYSTEM



Traditional *CL design

- Big chunk of memory contains everything
- A core is written in C manually.
- Lisp code compiled to C by a compiler written in lisp.
- Binaries loaded as data
- Whole image can be dumped and restored.

- Lisp application = System code + User code
- Lisp image can be dumped into executions
- Unused system code can be stripped out for saving application file size (optional)
- Building process of Common Lisp systems: Bootstrap (mostly)

Common Lisp Implementations (not all!)

NAME	COMPILER	SUPPORTING PLATFORMS	ARCHITECTURE	AVG. COST
CMUCL	Bytecode & Native	Linux, Mac OS X, Solaris, FreeBSD, NetBSD	*Intel x86, SPARC*	Free
SBCL	Native	Linux, Mac OS X, Solaris, FreeBSD, NetBSD, OpenBSD, Windows	*Intel x86, AMD64, PPC, SPARC, Alpha, MIPSbe, MIPEle*	Free
CLISP	Bytecode (old), Native	Windows and all UNIX-like systems	*Most architectures*	Free
ECL	Native via C, also byte-code	Linux, FreeBSD, NetBSD, OpenBSD, Solaris, Windows, Mac OS X, iOS	*Most architectures*	Free
ABCL	JVM Bytecode	All platforms with JDK	*JDK supported architectures*	Free
CLOZURE CL	Native	Mac OS X, Linux, FreeBSD, Solaris, Android, Windows	*Intel x86, AMD64, PPC, PPC64*	Free
MCL	Native	Mac OS 9, Mac OS X	*PPC*	Free since version 5.2
ALLEGRO CL	Native	Linux, Mac OS X, Windows, FreeBSD, Solaris	*Intel x86, AMD64, SPARC*	$599+
LISPWORKS	Native	Windows, Mac OS X, Linux, Solaris, AIX, HP-UX	*Intel x86, AMD64, PPC, PPC64, PA-RISC*	$1500 (pro), $5000 (ent)
LIQUID CL	Native	Solaris, UP-UX, AIX	*SPARC, PA-RISC, IBM RS/6000*	Unknown
SCIENEER CL	Native	Linux, Solaris, HP-UX	*Intel x86, AMD64, SPARC, IA64, PA-RISC*	$299+
CORMAN LISP	Native	Windows	*Intel x86*	$99
GCL	Native via C	Windows, Linux	*Most architectures*	Free
XCL	Native	Windows, Linux, FreeBSD	*Intel x86, AMD64*	Free

Books

- Common Lisp: The Language, 2nd Edition
- Practical Common Lisp
- On Lisp: Advanced Techniques for Common LISP
- Paradigms of Artificial Intelligence: Case study in Common Lisp
- COMMON LISP: An Interactive Approach
- Common LISP: a gentle introduction to symbolic computation
- Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS
- The Art of the Metaobject Protocol

Papers

- Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I, by John McCarthy
- History of LISP, by JohnMcCarthy
- The Evolution of Lisp, by Guy L. Steele Jr. etc.
- The Common Lisp Object System: An Overview, by Linda G. DeMichiel
- SBCL: a Sanely-Bootstrappable Common Lisp, by Christophe Rhodes

Links

- Common Lisp HyperSpec:

  http://www.lispworks.com/documentation/HyperSpec/Front/index.htm