Common Lisp Essentials

for Scheme programmers

Sebastián González
23 June 2009

UCL INGI

with many thanks to Dr. Pascal Costanza

Agenda

Language culture

1. History
2. Philosophy
3. Community

Language abstractions

4. Lisp-1 vs. Lisp-2
5. Lambda lists
6. Packages
7. Gen. assignment
8. Type system

Language extensions

9. Object system

1

History

>Basic Misconceptions

- Scheme is a cleaned-up version of all Lisps.
  - ➡ Common Lisp is the newer dialect!
  - ➡ *The Evolution of Lisp* (Steele and Gabriel) www.dreamsongs.com/Essays.html
- Common Lisp is slow.
  - ➡ Advanced, mature compilers.
- Common Lisp is not standard.
  - ➡ ANSI standard (first ever for an OOPL!)
- Common Lisp is dead.
  - ➡ Web applications, games, home appliances, and many more.

>History

- 1956: McCarthy's LISt Processing language, for symbolic data processing.
- 1975: "Scheme – An Interpreter for Extended Lambda Calculus" (Sussman, Steele)
- 1976-1980: 'Lambda Papers' (Sussman, Steele)

➡

> *No amount of language design can force a programmer to write clear programs. [...] The emphasis should not be on eliminating 'bad' language constructs, but on discovering or inventing helpful ones.*

>History

- 1982: "An Overview of Common LISP" (Steele et al.)
- 1984: "Common Lisp the Language" (Steele et al.)

>CL's First Goals

- Commonality among Lisp dialects
- Portability for "a broad class of machines"
- Consistency across interpreter & compilers
- Expressiveness based on experience
- Compatibility with previous Lisp dialects
- Efficiency: possibility to build optimizing compilers
- Stability: only "slow" changes to the language

>CL's First Non-Goals

- Graphics
- Multiprocessing
- Object-oriented programming

>History

- 1989: "Common Lisp the Language, 2nd Edition" (Steele et al.)

➡ *There are now many implementations of Common Lisp [...]. What is more, all the goals [...] have been achieved, most notably that of portability. Moving large bodies of Lisp code from one computer to another is now routine.*

>Further History

- Lisp Machines (80s)
- IEEE Scheme (1990)
- ANSI Common Lisp (1996)
    - 1100 pages describing 1000 funcs and vars
- ISO ISLISP (1997, mostly a CL subset)
- R5RS (1998, macros now officially supported)
- R6RS (2007)

2

Philosophy

>Scheme Philosophy

- Focus on *simplicity* and *homogeneity*.
  - ➡ Occam's Razor
    when there are two explanations for the same phenomenon, then the explanation which uses the smallest number of assumptions and concepts must be the right one
- Single paradigm.
  - "everything is a lambda expression"
- Advocates functional programming
  - side effects should be marked with a bang (!)

>CL Philosophy

- Focus on *expresiveness*, *pragmatics* and *efficiency*.
- CL integrates the OOP, FP and IP paradigms.
- IP: assignment, iteration, go.
- FP: lexical closures, first-class functions.
- IP & FP: many functions come both with and without side effects.

  cons & push
  adjoin & pushnew
  remove & delete
  reverse & nreverse
  etc.

3

Community

4 Abstractions

Pragmatics

1. Truth and falsity
2. Evaluation order
3. Lisp-1 vs. Lisp-2
4. Lambda lists
5. Generalised asignment

Control flow

6. Loop
7. Throw / catch
8. Conditions

Efficiency & correctness

9. Type system

Large scale

10. Dynamic scoping
11. Packages
12. CLOS

Meta & extensibility

13. Macros
14. MOP

>Truth and Falsehood

- Scheme
  - #t and every non-#f value vs. #f
  - predicates end in "?"
- Common Lisp
  - t and every non-nil value vs. nil
  - predicates usually end in "p" or "-p"
    - notable exceptions: eq, eql, equal

>Truth and Falsehood

- CL:
  ```
  (cdr (assoc key alist))
  ```
- Scheme:
  ```
  (let ((val (assq key a-list)))
     (cond ((not (null? val)) (cdr val))
           (else nil)))
  ```
- *Ballad Dedicated to the Growth of Programs*
  (Google for it)

>Evaluation Orders

- In Scheme, (+ i j k) may be evaluated in any order
  - this is specified
  - so never say: (+ i (set! i (+ i 1)))
- In CL, things are evaluated left to right.
  - specified in all useful cases
  - so (+ i (setf i (+ i 1))) is well defined.

>Iteration vs. Recursion

- Scheme: proper tail recursion.
- CL: no requirements, but usually optional tail recursion elimination.

  (proclaim '(optimize speed))

- Scheme: do, named let
- CL: do, do*, dolist, dotimes, loop

>Special Variables

- In CL, all global variables are dynamically scoped ("special variables").
- (Note: not the functions!)
- Dynamic scope: global scope + dynamic extent.
- By convention, names are marked with *
  - ➡ *package* *features* *print-base*

>Symbols

- **Symbolic computation** is the kind of programming that relies on a symbol data type.
- Symbols are central to all Lisp dialects.
- Common Lisp has advanced facilities to work with symbols.

>Packages

- Packages are containers for symbols, used as namespaces or “shared vocabularies”.
- Packages help avoiding name pollution and clashes.
- The CL reader uses packages to translate the literal names it finds into symbols.

  (find-symbol "CAR" "CL") → 'car
  (find-symbol "CAr" "CL") → nil
- Symbols can be internal, external or inherited.
- So we don’t export functions etc., but symbols.

>Symbol Literals

- Unqualified (current package)
  - ➡ foo, Foo, FoO, FOO
- Qualified
  - ➡ *External* – acme:foo
  - ➡ *Internal* – acme::foo
  - ➡ *Keywords* – :foo keyword:foo

    (eq ':foo :foo) → T
  - ➡ *Uninterned* – #:foo

    (eq '#:foo '#:foo) → NIL

>Packages: How it Works

- (in-package “BANK”)
  (export 'withdraw)
  (defun withdraw (x) ...)
- Allows other packages to say:
  (bank:withdraw 500)
- Or:
  (use-package “BANK”)
  (withdraw 500)

>Packages: Utilities

```
(defpackage bank
    (:documentation “Sample package”)
    (:use common-lisp)
    (:export withdraw deposit consult ...))
```

>Lisp-1 vs. Lisp-2

- In Scheme, a symbol may be bound to a value, and functions in particular are values.
- In CL, functions and values have different namespaces. In a form,
  - car position is interpreted in function space
  - cdr positions are interpreted in value space
- So you can say

```
(flet ((fun (x) (1+ x)))
  (let ((fun 42))
    (fun fun)))
```

>Lisp-1 vs. Lisp-2

- There are accessors for each namespace:
  - (symbol-function 'fun) or #'fun or (function fun)
  - (symbol-value 'fun) or fun
- Call functional values as:
  (fun 42) or (funcall #'fun 42) or (apply #'fun (list 42))
  Functions are first-class just like in Scheme

>Why Lisp-1?

- Homogeneity: let all positions in a form be evaluated the same. You can say (((f x) y) z)
- Avoid having separate binding manipulation constructs for each namespace.
  - CL:
    let / flet
    boundp / fboundp
    symbol-value / symbol-function
    defun / defvar

>But why Lisp-2?

- In practice, having the possibility of reusing names for functions and variables is very handy.
  - No need to prepend 'get-' to getters

```
(let ((age (age person)))
  (+ age 10))
```

- Lisp-2 is *practical*. About 80% of CL programmers use it.

>Lambda Lists

- CL's parameter lists provide a convenient solution to several common coding problems.

>λ Lists: Optional Args

- CL:
```
(defun foo (a b &optional (c 0) d)
  (list a b c d))

(foo 1 2)       → (1 2 0 NIL)
(foo 1 2 3)     → (1 2 3 NIL)
(foo 1 2 3 4)   → (1 2 3 4)
```

>λ Lists: Variable Arity

- Scheme:
  (define (format ctrl-string . objects) ...)
  (define (+ . numbers) ...)
- CL:
  (defun format (stream string &rest values) ...)
  (defun + (&rest numbers) ...)

>λ Lists: Keyword Args

```
(defun find (item list &key (test #'eql) (key #'identity)) ...)

(find “Karl” *list-of-persons*
        :key #'person-name
        :test #'string=)
```

>λ Lists: Keyword Args (L2R)

```
(defun withdraw (...) ...)

  ...
  (flet ((withdraw (&rest args
                    &key amount
                    &allow-other-keys)
           (if (> amount 100000)
               (apply #'withdraw :amount 100000 args)
               (apply #'withdraw args))))
     ...)
  ...
```

>Lambda Lists

• &rest, &body	rest parameters
• &optional	optional parameters
• &key, &allow-other-keys	keyword parameters
• &environment	lexical environment
• &aux	local variables
• &whole	the whole form

>Generalised Asignment

- ...or "generalized references"
- like ":=" or "=" in Algol-style languages, with arbitrary left-hand sides
- (setf (some-form ...) (some-value ...))
- predefined acceptable forms for left-hand sides + framework for user-defined forms

Python	CL
x = 10	(setf x 10)
a[0] = 10	(setf (aref a 0) 10)
hash['key'] = 10	(setf (gethash 'key hash) 10)
o.field = 10	(setf (field o) 10)

>Generalised Assignment

- Earlier dialects of Lisp would often have pairs of functions for reading and writing data.
- The **setf** macro improves CL's orthogonality.
- In CL there are only "getters", and setters come for free.
  - (age person) → 32
  - (setf (age person) 42) → 42

>Assignment Functions

- ```
  (defun make-cell (value) (vector value))
  (defun cell-value (cell) (svref cell 0))
  (defun (setf cell-value) (value cell)
    (setf (svref cell 0) value))
  ```
- `(setf (cell-value some-cell) 42)`
- macros also supported
  ```

>Type System

- A type is a possibly infinite set of objects.
- CL allows optional declaration of types.
  (declaim (type integer *my-counter*))
  (declare (integer x y z))
  (the integer (* x y))
- Usually, CL implementations take type declarations as a promise for code optimization.
- Creation of new types: deftype, defstruct, defclass, define-condition.

>Type System

Type queries

- (type-of 1) → 'bit
- (type-of 2) → '(integer 0 536870911)
- (type-of "hola") → (simple-array character (4))
- (typep 3 '(integer 0 2)) → nil
- (typep 'a '(and symbol (not null))) → t
- (subtypep 'integer 'number) → t

>Finally

- CL defines a large number of predefined data structures and operations:

  CLOS, structures, conditions, numerical tower, extensible characters, optionally typed arrays, multidimensional arrays, hash tables, filenames, streams, printer, reader.

- Apart from these differences, Scheme and Common Lisp are almost the same. ;)

CLOS

the common lisp
object system

>Class-based OOP

```
class OutputStream {
   void println(Object obj) { ... }
   ...
}
```

↓

```
out.println(pascal);
```

>...in Lisp syntax...

out.println(pascal);

(send out ‘println pascal)

>...the receiver is just another argument...

(call receiver message args ...)

↓

(call message receiver args ...)

↓

(call message all-args ...)

>..."call" is redundant...

(call message args ...)

↓

(message args ...)

>...so now we have generic functions!

out.println(pascal);

↓

(println out pascal)

>Classes

```
(defclass person (standard-object)
    ((name :accessor person-name
           :initarg :name)
     (address :accessor person-address
              :initarg :address))
    (:documentation "Basic person."))
```

>Classes and Superclasses

```
(defclass person (standard-object)
    ((name :accessor person-name
           :initarg :name)
     (address :accessor person-address
              :initarg :address))
    (:documentation "Basic person."))
```

>Slots and Options

```
(defclass person (standard-object)
    ((name :accessor person-name
           :initarg :name)
     (address :accessor person-address
              :initarg :address))
    (:documentation "Basic person."))
```

>Class Options

```
(defclass person (standard-object)
    ((name :accessor person-name
               :initarg :name)
     (address :accessor person-address
               :initarg :address))
    (:documentation "Basic person."))
```

> Instances & Accessors

```
(defclass person (standard-object)
   ((name :accessor person-name :initarg :name)
    (address :accessor person-address :initarg :address))
   (:documentation "Basic person."))

(defparameter *dilbert*
  (make-instance 'person :name "Dilbert" :address "Brussels"))


(person-name *dilbert*) → "Dilbert"
```

>Generic Functions

- Invented when Lispers implemented OOP.
- Generic functions were already needed.
  Mathematical operations are generic!
  They work on ints, floats, complex, etc.

```
(defgeneric + (x y)
   :documentation “returns the sum of x and y”)
(defmethod + ((x int) (y int)) ...)
(defmethod + ((x float) (y float)) ...)
(defmethod + ((x complex) (y complex)) ...)
```

>Generic Functions

- Methods belong to the generic function.
- The GF is responsible for determining what method(s) to run in response to a particular invocation.
  - ➡ Multiple dispatch: consider all the arguments when selecting applicable and most specific methods.
  - ➡ Advice: add qualified methods that are called before, after or around everything else.

> Inheritance

- (defgeneric display (object))
- (defmethod display ((object person))
  (print (person-name object))
  (print (person-address object)))
- (defclass employee (person)
  ((employer :accessor person-employer
  :initarg :employer)))
- (defmethod display ((object employee))
  (call-next-method)
  (display (person-employer object)))

>GFs & Methods

- (defmethod display ((object person))
  ...)
- (defmethod display :before ((object person))
  ...)
- Standard method combination allows for primary, :before, :after and :around methods.

>GFs & Methods

- (defgeneric display (object)
  (:method-combination progn :most-specific-last))
- (defmethod display progn ((object person))
  (print (person-name object))
  (print (person-address object)))
- (defmethod display progn ((object employee))
  (print (person-employer object)))

>Single Dispatch

```
public class Object {
  public boolean equals(Object other) {
    return this == other;
} }
```

```
public class Person {
  public boolean equals(Person other) {
    this.name().equals(other.name());
} }
```

Now consider:

```
Object a = new Person(“juan”);
Object b = new Person(“juan”);
a.equals(b)
```

>Single Dispatch

```
public class Object {
  public boolean equals(Object other) {
    return this == other;
} }
```

```
public class Person {
  public boolean equals(Person other) {
    this.name().equals(other.name());
} }
```

Now consider:

```
Object a = new Person(“juan”);
Object b = new Person(“juan”);
a.equals(b)  →  false
```

> Single Dispatch

```
public class Object {
  public boolean equals(Object other) {
    return this == other;
} }
```

```
public class Person {
  public boolean equals(Object other) {
    this.name().equals(other.name());
} }
```

Now consider:

```
Object a = new Person("juan");
Object b = new Person("juan");
a.equals(b)
```

→ **false**

> Single Dispatch

```
public class Object {
  public boolean equals(Object other) {
    return this == other;
} }
```

```
public class Person {
  public boolean equals(Object other) {
    this.name().equals(other.name());
} }
```

Now consider:

```
Object a = new Person("juan");
Object b = new Person("juan");
a.equals(b)
```

→ **false**

dynamic method binding based on receiver only

> Single Dispatch

```
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"); }
}
```

What happens when you run the following main method?

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

> Single Dispatch

```
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"); }
}
```

What happens when you run the following main method?

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

```
bash$ java Main A
```

>Single Dispatch

```
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"); }
}
```

What happens when you run the following main method?

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

bash$ java Main A → “A/A”

> Single Dispatch

```
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"); }
}
```

What happens when you run the following main method?

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

bash$ java Main A → "A/A"

bash$ java Main B

> Single Dispatch

```
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"); }
}
```

What happens when you run the following main method?

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

bash$ java Main A → "A/A"

bash$ java Main B → "B/A"

> Single Dispatch

```
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"); }
}
```

What happens when you run the following main method?

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

bash$ java Main A → “A/A”

bash$ java Main B → “B/A” “B/B”

>Multiple Dispatch

```
(defclass A () ())
(defclass B (A) ())

(defmethod foo ((x A) (y A)) (print "A/A"))
(defmethod foo ((x A) (y B)) (print "A/B"))

(defmethod foo ((x B) (y A)) (print "B/A"))
(defmethod foo ((x B) (y B)) (print "B/B"))
```

If you try:

```
(defun test (class)
   (let ((obj (make-instance class)))
      (foo obj obj)))
```

>Multiple Dispatch

```
(defclass A () ())
(defclass B (A) ())

(defmethod foo ((x A) (y A)) (print "A/A"))
(defmethod foo ((x A) (y B)) (print "A/B"))

(defmethod foo ((x B) (y A)) (print "B/A"))
(defmethod foo ((x B) (y B)) (print "B/B"))
```

If you try:

```
(defun test (class)
   (let ((obj (make-instance class)))
      (foo obj obj)))
```

(test 'a)

>Multiple Dispatch

```
(defclass A () ())
(defclass B (A) ())

(defmethod foo ((x A) (y A)) (print "A/A"))
(defmethod foo ((x A) (y B)) (print "A/B"))

(defmethod foo ((x B) (y A)) (print "B/A"))
(defmethod foo ((x B) (y B)) (print "B/B"))
```

If you try:

```
(defun test (class)
  (let ((obj (make-instance class)))
    (foo obj obj)))
```

(test 'a) → "A/A"

>Multiple Dispatch

```
(defclass A () ())
(defclass B (A) ())

(defmethod foo ((x A) (y A)) (print "A/A"))
(defmethod foo ((x A) (y B)) (print "A/B"))

(defmethod foo ((x B) (y A)) (print "B/A"))
(defmethod foo ((x B) (y B)) (print "B/B"))
```

If you try:

```
(defun test (class)
  (let ((obj (make-instance class)))
    (foo obj obj)))
```

(test 'a) → "A/A"

(test 'b)

>Multiple Dispatch

```
(defclass A () ())
(defclass B (A) ())

(defmethod foo ((x A) (y A)) (print "A/A"))
(defmethod foo ((x A) (y B)) (print "A/B"))

(defmethod foo ((x B) (y A)) (print "B/A"))
(defmethod foo ((x B) (y B)) (print "B/B"))
```

If you try:

```
(defun test (class)
  (let ((obj (make-instance class)))
    (foo obj obj)))
```

(test 'a) → "A/A"

(test 'b) → "B/B"

Concluding Remarks

>Greenspun's Tenth Rule

"Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified bug-ridden slow implementation of half of Common Lisp."

> Important Literature

- Peter Norvig, Paradigms of Artificial Intelligence Programming (PAIP)
  - CL's SICP
- Paul Graham, On Lisp - *the* book about macros (out of print, but see www.paulgraham.com)
- Peter Seibel, Practical Common Lisp, 2005, www.gigamonkeys.com/book

>Important Literature

- Guy Steele, Common Lisp The Language, 2nd Edition (CLtL2 - pre-ANSI!)
- HyperSpec, (ANSI standard), Google for it!
- Pascal's highly opinionated guide http://p-cos.net/lisp/guide.html
- ISLISP: www.islisp.info