

# RailsOnLisp

Thomas de Grivel [thoxdg@gmail.com](mailto:thoxdg@gmail.com)

<http://kmx.io>

10 octobre 2019

# Common Lisp

# Common Lisp

## Introduction

Common Lisp is the programmable programming language. Lisp essays by Paul

Graham

<http://www.paulgraham.com/lisp.html>

Standardised in 1994 by ANSI

Common Lisp the Language, 2nd Edition

<https://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html>

Common Lisp Hyperspec

<http://www.lispworks.com/documentation/HyperSpec/Front/Cliki>

<http://cliki.net>

De nombreux compilateurs respectant le standard ANSI existent :

- SBCL (open-source, x86, amd64, Windows, Linux, OSX, \*BSD)
- ABCL (open-source, jvm)
- Clozure CL (open-source, x86, amd64, Windows, Linux, OSX, FreeBSD)
- ECL (open-source, compiles to C)
- LispWorks (proprietary, x86, amd64, Windows, Linux, OSX, FreeBSD)
- Allegro CL (proprietary, x86, amd64, sparc, Windows, Linux, OSX, FreeBSD)

# Common Lisp Installation

Ubuntu :

```
sudo apt-get install sbcl
```

MacOS X :

```
brew install sbcl
```

```
mkdir -p ~/common-lisp/thodg  
cd ~/common-lisp/thodg  
git clone https://github.com/thodg/repo.git  
cd ~/common-lisp  
ln -s thodg/repo/repo.manifest
```



Éditer ~/.sbclrc

```
;; ASDF
(require :asdf)

;; repo
(load "~/common-lisp/thodg/repo/repo")
(repo:boot)
```

```
$ sbcl
```

This is SBCL 1.5.3, an implementation of ANSI Common Lisp.

More information about SBCL is available at [<http://www.sbcl.org/>](http://www.sbcl.org/).

SBCL is free software, provided as is, with absolutely no warranty. It is mostly in the public domain; some portions are provided under BSD-style licenses. See the CREDITS and COPYING files in the distribution for more information.

```
* _
```

```
* (repo:install :slime)
```

```
$ /usr/bin/git -C /home/dx/common-lisp/slime clone https://github.com/slime/slime  
Cloning into 'slime'...
```

Éditer ~/.emacs

```
;; Common Lisp
(add-to-list 'load-path "~/common-lisp/slime/slime/")
(require 'slime-autoloads)
(add-to-list 'slime-contribs 'slime-fancy)
(setq inferior-lisp-program
      "sbcl")
(setq slime-net-coding-system
      'utf-8-unix)
```

# Common Lisp

## Demo

```
$ emacs
```

```
M-x slime
```

```
CL-USER> _
```

REPL : read, eval, print loop

```
(loop
  ;; setup REPL vars
  ;; handle errors, interactive debugger
  (print
    (eval
      (read)))
  (force-output)) ;; flush output buffers
```

Un symbole est plus rapide à comparer qu'une chaîne de caractères (comparaison de pointeurs). Pour récupérer un symbole à travers `eval` il faut le citer en le préfixant d'une apostrophe.

```
;; SLIME
```

```
CL-USER> 'hello-world
```

```
HELLO WORLD
```

```
CL-USER> (quote hello-world) ; equivalent sans syntaxe
```

```
HELLO WORLD
```

<http://www.gigamonkeys.com/book/programming-in-the-large-packages-and-symbols.html>



Si on ne quote pas le symbole on tombe dans le debugger interactif.

```
;; SLIME  
CL-USER> hello-world
```

The variable HELLO-WORLD is unbound.  
[Condition of type UNBOUND-VARIABLE]

Restarts:

- 0: [CONTINUE] Retry using HELLO-WORLD.
- 1: [USE-VALUE] Use specified value.
- 2: [STORE-VALUE] Set specified value and use it.
- 3: [RETRY] Retry SLIME REPL evaluation request.
- 4: [\*ABORT] Return to SLIME's top level.
- 5: [ABORT] abort thread (#<THREAD "repl-thread" RUNNING {1003B91BC3}>)

Backtrace:

- 0: (SB-INT:SIMPLE-EVAL-IN-LEXENV HELLO-WORLD #<NULL-LEXENV>)
  - 1: (EVAL HELLO-WORLD)
- more--

4

; Evaluation aborted on #<UNBOUND-VARIABLE HELLO-WORLD {1004AF3523}>.  
CL-USER> \_

Pour définir une fonction on utilise `defun`. Si le premier élément d'une liste (entre parenthèses) est une fonction ou un symbole nommant une fonction alors c'est un appel de fonction.

```
;; SLIME
CL-USER> (defun hello-world ()
           (format t "Hello world !"))

HELLO-WORLD
CL-USER> (hello-world)
Hello world !
NIL
CL-USER> _
```

Une fonction anonyme est introduite par `lambda`. On peut affecter une fonction anonyme à un symbole, reproduisant l'effet de `defun`.

```
;; SLIME
CL-USER> (setf (symbol-function 'hello-world)
               (lambda ()
                 (format t "Hello world !")))

CL-USER> (hello-world)
Hello world !
NIL
CL-USER> _
```

Une fonction est une valeur comme une autre et peut être passée en paramètre d'une autre fonction. On appelle ces fonctions les fonctions d'ordre supérieur.

```
;; SLIME
CL-USER> (mapcar (lambda (x) (* x x)) '(1 2 3 4 5))
(1 4 9 16 25)
CL-USER> (reduce #' + '(1 2 3 4 5))
15
CL-USER> (reduce (function +) '(1 2 3 4 5))
15
CL-USER> (reduce ' + '(1 2 3 4 5))
15
CL-USER> _
```

Les paramètres d'une macro ne sont pas évalués. Cela permet de faire des DSL et de la meta-programmation. Une macro génère du code qui est à son tour évalué.

```
;; SLIME
CL-USER> (defmacro hello (arg)
           `(format nil "Hello ~A !"
                     (string-capitalize ',arg)))

HELLO
CL-USER> (hello world)
"Hello World !"
CL-USER> _
```

On Lisp, Paul Graham

<http://lib.store.yahoo.net/lib/paulgraham/onlisp.pdf>

- Pour quoter entièrement : `'(a b c)` ou `(quote (a b c))`  
⇒ `(a b c)`
- Pour quoter partiellement : `'(a b ,c)` ou `(list 'a 'b c)`  
⇒ `(a b 123)` si `c = 123`

Backquote arrete l'evaluation et virgule la réactive localement.

```
;; SLIME
CL-USER> (let ((c 123))
           `(a b ,c))
(A B 123)
```

# RailsOnLisp

# RailsOnLisp

## Installation



```
$ mkdir ~/common-lisp/RailsOnLisp
$ cd ~/common-lisp/RailsOnLisp
$ git clone https://github.com/RailsOnLisp/rol.git
Cloning into 'rol' ...

$ _
```

Éditer ~/.profile

```
if [ -d "$HOME/common-lisp/RailsOnLisp/rol/bin" ]; then
    PATH="$HOME/common-lisp/RailsOnLisp/rol/bin:$PATH"
fi
```

```
$ . ~/.profile    # sourcer .profile ou lancer un nouveau shell
$ rol install
Cloning into 'rol-assets' ...
Cloning into 'rol-files' ...
Cloning into 'rol-log' ...
Cloning into 'rol-server' ...
Cloning into 'rol-template' ...
Cloning into 'rol-uri' ...

$ ls -l ~/common-lisp/RailsOnLisp/rol

$ _
```

RailsOnLisp

Demo

```
$ rol new test
```

```
$ _
```