RailsOnLisp

Thomas de Grivel ¡thoxdg@gmail.com¿

http://kmx.io

October 10, 2019

Common Lisp

Common Lisp

Common Lisp

Common Lisp is the programmable programming language.

Common Lisp

Standardised in 1994 by ANSI
`http://www.lispworks.com/documentation/HyperSpec/Front/`

Common Lisp

Several compilers implementing the ANSI standard :

- 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

Lisp essays by Paul Graham
`http://www.paulgraham.com/lisp.html`

Common Lisp

Install SBCL

Ubuntu :

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

MacOS X :

```
brew install sbcl
```

Install repo

```
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
```

Configure SBCL

Edit ~/.sbclrc

```
;; ASDF
(require :asdf)

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

Launch SBCL

```
$ sbcl
This is SBCL 1.5.3, an implementation of ANSI Common Lisp.
More information about SBCL is available at <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.

* _
```

Install Slime

```
* (repo:install :slime)

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

Configure emacs

Edit ~/.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

Launch emacs and slime

```
$ emacs

M-x slime

CL-USER> _
```

The REPL

REPL : `read`, `eval`, `print loop`

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

Symbols

A symbol compares faster than a string (pointers comparison). To get a symbol through `eval` we have to quote it.

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

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

HELLO WORLD
```

Symbols

If the symbol is not quoted then we end up in the interactive debugger :

```
;; 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> _
```

Functions

`defun` defines a function. If the first element of a list (between parentheses) is a function or a symbol naming a function then the list is treated as a function call.

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

Lambda

`lambda` introduces an anonymous function. We can affect an anonymous function to a symbol, not unlike `defun`.

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

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

Higher order functions

A function is a value like others and can be passed to another function. We call these functions higher order.

```
;; 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))    ; equivalent to above
15
CL-USER> (reduce '+ '(1 2 3 4 5))    ; not equivalent will resolve function a
15
CL-USER> _
```

Les macros

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> _
```

RailsOnLisp

RailsOnLisp

Cloner RailsOnLisp/rol.git

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

$ _
```

Configurer le PATH

Éditer `~/.profile`

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

Installer RailsOnLisp

```
$ . ~/.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

$ _
```