eBook Gratuit

APPRENEZ common-lisp

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#common-lisp

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: common-lisp

It is an unofficial and free common-lisp ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official common-lisp.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Commencer avec Common-Lisp

Remarques

Ceci est une fonction simple hello world dans Common Lisp. Des exemples vont imprimer le texte `Hello, World!` (sans les guillemets, suivi d'une nouvelle ligne) à la sortie standard.

Common Lisp est un langage de programmation largement utilisé de manière interactive en utilisant une interface appelée REPL. Le REPL (Read Eval Print Loop) permet de taper du code, de l'évaluer (voir) et de voir les résultats immédiatement. L'invite pour la REPL (à laquelle on tape le code à exécuter) est indiquée par `CL-USER>` . Parfois, quelque chose d'autre que `CL-USER` apparaîtra avant le `>` mais ceci est toujours une REPL.

Après l'invite, il y a un code, généralement un seul mot (un nom de variable) ou un formulaire (une liste de mots / formes entre `(` et `)`) (un appel de fonction ou une déclaration, etc.). Sur la ligne suivante, il y aura toute sortie que le programme imprime (ou rien si le programme n'imprime rien), puis les valeurs renvoyées en évaluant l'expression. Normalement, une expression renvoie une valeur mais si elle renvoie plusieurs valeurs, elles apparaissent une fois par ligne.

Versions

Version	Date de sortie
Lisp commun	1984-01-01
ANSI Common Lisp	1994-01-01

Examples

Bonjour le monde

Ce qui suit est un extrait d'une session REPL avec Common Lisp dans laquelle un "Hello, World!" la fonction est définie et exécutée. Voir les remarques au bas de cette page pour une description plus détaillée d'une REPL.

```
CL-USER> (defun hello ()
           (format t "Hello, World!~%"))
HELLO
CL-USER> (hello)
Hello, World!
NIL
CL-USER>
```

Ceci définit la "fonction" de zéro argument nommé `hello` , qui écrira la chaîne `"Hello, World!"` suivi d'une nouvelle ligne à la sortie standard et retourne `NIL` .

Pour définir une fonction on écrit

```
(defun name (parameters...)
  code...)
```

Dans ce cas, la fonction est appelée `hello` , ne prend aucun paramètre et le code qu'elle exécute fait un appel de fonction. La valeur renvoyée par une fonction lisp est le dernier bit de code de la fonction à exécuter, donc `hello` renvoie le résultat `(format t "Hello, World!~%")` .

Dans lisp pour appeler une fonction, on écrit `(function-name arguments...)` où `function-name` est le nom de la fonction et `arguments...` est la liste d'arguments (séparés par des espaces) de l'appel. Il y a des cas particuliers qui ressemblent à des appels de fonction mais qui ne sont pas, par exemple, dans le code ci-dessus, il n'y a pas de fonction `defun` appelée, elle est traitée spécialement et définit une fonction à la place.

À la deuxième invite de la REPL, après avoir défini la fonction `hello` , nous l'appelons sans paramètres en écrivant `(hello)` . Cela appellera à son tour la fonction de `format` avec les paramètres `t` et `"Hello, World!~%"` . La fonction `format` produit une sortie formatée en fonction des arguments qui lui sont donnés (un peu comme une version avancée de `printf` en C). Le premier argument lui indique où sortir, avec `t` signifiant sortie standard. Le second argument lui indique quoi imprimer (et comment interpréter les paramètres supplémentaires). La directive (code spécial dans le deuxième argument) `~%` indique au format d'imprimer une nouvelle ligne (c'est-à-dire que sous UNIX, elle peut écrire `\n` et sur Windows `\r\n`). Le format retourne généralement `NIL` (un peu comme `NULL` dans les autres langues).

Après la deuxième invite, nous voyons que `Hello, World` a été imprimé et à la ligne suivante que la valeur renvoyée était `NIL` .

Bonjour, Nom

Ceci est un exemple légèrement plus avancé qui montre quelques fonctionnalités supplémentaires du lisp commun. Nous commençons par un simple `Hello, World!` fonction et démontrer un développement interactif au REPL. Notez que tout texte à partir d' un point - virgule `;` , au reste de la ligne est un commentaire.

```
CL-USER> (defun hello ()
       (format t "Hello, World!~%")) ;We start as before
HELLO
CL-USER> (hello)
Hello, World!
NIL
CL-USER> (defun hello-name (name) ;A function to say hello to anyone
       (format t "Hello, ~a~%" name)) ;~a prints the next argument to format
HELLO-NAME
CL-USER> (hello-name "Jack")
Hello, Jack
NIL
CL-USER> (hello-name "jack") ;doesn't capitalise names
Hello, jack
NIL
CL-USER> (defun hello-name (name) ;format has a feature to convert to title case
```

```
       (format t "Hello, ~:(~a~)~%" name)) ;anything between ~:( and ~) gets it
WARNING: redefining COMMON-LISP-USER::HELLO-NAME in DEFUN
HELLO-NAME
CL-USER> (hello-name "jack")
Hello, Jack
NIL
CL-USER> (defun hello-name (name)
       (format t "Hello, ~:(~a~)!~%" name))
WARNING: redefining COMMON-LISP-USER::HELLO-NAME in DEFUN
HELLO-NAME
CL-USER> (hello-name "jack") ;now this works
Hello, Jack!
NIL
CL-USER> (defun hello (&optional (name "world")) ;we can take an optional argument
       (hello-name name)) ;name defaults to "world"
WARNING: redefining COMMON-LISP-USER::HELLO in DEFUN
HELLO
CL-USER> (hello)
Hello, World!
NIL
CL-USER> (hello "jack")
Hello, Jack!
NIL
CL-USER> (hello "john doe") ;note that this capitalises both names
Hello, John Doe!
NIL
CL-USER> (defun hello-person (name &key (number))
       (format t "Hello, ~a ~r" name number)) ;~r prints a number in English
HELLO-PERSON
CL-USER> (hello-person "Louis" :number 16) ;this doesn't quite work
Hello, Louis sixteen
NIL
CL-USER> (defun hello-person (name &key (number))
       (format t "Hello, ~:(~a ~:r~)!" name number)) ;~:r prints an ordinal
WARNING: redefining COMMON-LISP-USER::HELLO-PERSON in DEFUN
HELLO-PERSON
CL-USER> (hello-person "Louis" :number 16)
Hello, Louis Sixteenth!
NIL
CL-USER> (defun hello-person (name &key (number))
       (format t "Hello, ~:(~a ~@r~)!" name number)) ;~@r prints Roman numerals
WARNING: redefining COMMON-LISP-USER::HELLO-PERSON in DEFUN
HELLO-PERSON
CL-USER> (hello-person "Louis" :number 16)
Hello, Louis Xvi!
NIL
CL-USER> (defun hello-person (name &key (number)) ;capitalisation was wrong
       (format t "Hello, ~:(~a~) ~:@r!" name number))
WARNING: redefining COMMON-LISP-USER::HELLO-PERSON in DEFUN
HELLO-PERSON
CL-USER> (hello-person "Louis" :number 16) ;thats better
Hello, Louis XVI!
NIL
CL-USER> (hello-person "Louis") ;we get an error because NIL is not a number
Hello, Louis ; Evaluation aborted on #<SB-FORMAT:FORMAT-ERROR {1006641AB3}>.
CL-USER> (defun say-person (name &key (number 1 number-p)
                                      (title nil) (roman-number t))
       (let ((number (if number-p
                 (typecase number
                   (integer
                (format nil (if roman-number " ~:@r" " ~:(~:r~)") number))
```

```
                   (otherwise
                 (format nil " ~:(~a~)" number)))
                  "")) ; here we define a variable called number
          (title (if title
                 (format nil "~:(~a~) " title)
                 ""))) ; and here one called title
          (format nil "~a~:(~a~)~a" title name number))) ;we use them here

SAY-PERSON
CL-USER> (say-person "John") ;some examples
"John"
CL-USER> (say-person "john doe")
"John Doe"
CL-USER> (say-person "john doe" :number "JR")
"John Doe Jr"
CL-USER> (say-person "john doe" :number "Junior")
"John Doe Junior"
CL-USER> (say-person "john doe" :number 1)
"John Doe I"
CL-USER> (say-person "john doe" :number 1 :roman-number nil) ;this is wrong
"John Doe First"
CL-USER> (defun say-person (name &key (number 1 number-p)
                                      (title nil) (roman-number t))
        (let ((number (if number-p
                  (typecase number
                    (integer
                 (format nil (if roman-number " ~:@r" " the ~:(~:r~)") number))
                    (otherwise
                 (format nil " ~:(~a~)" number)))
                  ""))
          (title (if title
                 (format nil "~:(~a~) " title)
                 "")))
          (format nil "~a~:(~a~)~a" title name number)))
WARNING: redefining COMMON-LISP-USER::SAY-PERSON in DEFUN
SAY-PERSON
CL-USER> (say-person "john doe" :number 1 :roman-number nil) ;thats better
"John Doe the First"
CL-USER> (say-person "louis" :title "king" :number 16 :roman-number nil)
"King Louis the Sixteenth"
CL-USER> (say-person "louis" :title "king" :number 16 :roman-number t)
"King Louis XVI"
CL-USER> (defun hello (&optional (name "World") &rest arguments) ;now we will just
        (apply #'hello-name name arguments)) ;pass all arguments to hello-name
WARNING: redefining COMMON-LISP-USER::HELLO in DEFUN
HELLO
CL-USER> (defun hello-name (name &rest arguments) ;which will now just use
        (format t "Hello, ~a!" (apply #'say-person name arguments))) ;say-person
WARNING: redefining COMMON-LISP-USER::HELLO-NAME in DEFUN
HELLO-NAME
CL-USER> (hello "louis" :title "king" :number 16) ;this works now
Hello, King Louis XVI!
NIL
CL-USER>
```

Cela met en évidence certaines des fonctionnalités avancées de la fonction de `format` de Common Lisp, ainsi que certaines fonctionnalités telles que les paramètres facultatifs et les arguments de mot-clé (par exemple `:number`). Cela donne également un exemple de développement interactif à une REPL en langage courant.

Le programme simple Hello World dans REPL

Common Lisp REPL est un environnement interactif. Chaque formulaire écrit après l'invite est évalué et sa valeur est ensuite imprimée à la suite de l'évaluation. Ainsi, le programme le plus simple possible «Bonjour, Monde!» Dans Common Lisp est:

```
CL-USER> "Hello, World!"
"Hello, World!"
CL-USER>
```

Ce qui se passe ici est qu'un costant de chaîne est donné en entrée à la REPL, il est évalué et le résultat est imprimé. Ce que l'on peut voir dans cet exemple est que les chaînes, comme les nombres, les symboles spéciaux comme `NIL` et `T` et quelques autres littéraux, sont *des* formulaires *auto-* évaluables: c'est-à-dire qu'ils se évaluent eux-mêmes.

Expressions de base

Essayons une expression de base dans le REPL:

```
CL-USER> (+ 1 2 3)
6
CL-USER> (- 3 1 1)
1
CL-USER> (- 3)
-3
CL-USER> (+ 5.3 (- 3 2) (* 2 2))
10.3
CL-USER> (concatenate 'string "Hello, " "World!")
"Hello, World!"
CL-USER>
```

Le bloc de construction de base d'un programme Common Lisp est le *formulaire* . Dans ces exemples, nous avons *des formes de fonctions* , c'est-à-dire des expressions, écrites sous forme de liste, dans lesquelles le premier élément est un opérateur (ou une fonction) et le reste des éléments les opérandes "). L'écriture de formulaires dans le REPL entraîne leur évaluation. Dans les exemples, vous pouvez voir des expressions simples dont les arguments sont des nombres constants, des chaînes et des symboles (dans le cas de `'string` , qui est le nom d'un type). Vous pouvez également voir que les opérateurs arithmétiques peuvent prendre n'importe quel nombre d'arguments.

Il est important de noter que les parenthèses font partie intégrante de la syntaxe et ne peuvent pas être utilisées librement comme dans d'autres langages de programmation. Par exemple, ce qui suit est une erreur:

```
(+ 5 ((+ 2 4)))
> Error: Car of ((+ 2 4)) is not a function name or lambda-expression. ...
```

Dans Common Lisp, les formulaires peuvent également être des données, des symboles, des formes de macros, des formulaires spéciaux et des formes lambda. Ils peuvent être écrits pour être évalués, retourner zéro, une ou plusieurs valeurs, ou peuvent être donnés en entrée à une

macro, qui les transforment sous d'autres formes.

Somme de la liste d'entiers

```
(defun sum-list-integers (list)
    (reduce '+ list))

; 10
(sum-list-integers '(1 2 3 4))

; 55
(sum-list-integers '(1 2 3 4 5 6 7 8 9 10))
```

Expressions lambda et fonctions anonymes

Une fonction anonyme peut être définie sans nom via une expression Lambda . Pour définir ce type de fonctions, le mot-clé `lambda` est utilisé à la place du mot-clé `defun` . Les lignes suivantes sont toutes équivalentes et définissent des fonctions anonymes qui génèrent la somme de deux nombres:

```
(lambda (x y) (+ x y))
(function (lambda (x y) (+ x y)))
#'(lambda (x y) (+ x y))
```

Leur utilité est notable lors de la création de formulaires Lambda , c.-à-d. Un formulaire qui est une liste où le premier élément est l'expression lambda et les éléments restants sont les arguments de la fonction anonyme. Exemples (exécution en ligne):

```
(print ((lambda (x y) (+ x y)) 1 2)) ; >> 3

(print (mapcar (lambda (x y) (+ x y)) '(1 2 3) '(2 -5 0))) ; >> (3 -3 3)
```

Ressources d'apprentissage Common Lisp

Livres en ligne

Ce sont des livres librement accessibles en ligne.

- Practical Common Lisp de Peter Seibel est une bonne introduction à la technologie CL pour les programmeurs expérimentés, qui essaie depuis le début de mettre en évidence ce qui rend CL différent des autres langages.
- Common Lisp: Une introduction douce au calcul symbolique par David S. Touretzky est une bonne introduction pour les nouveaux venus en programmation.
- Common Lisp: Une approche interactive de Stuart C. Shapiro a été utilisée comme manuel de cours et des notes de cours accompagnent le livre sur le site Web.
- Common Lisp, le langage de Guy L. Steele est une description du langage Common Lisp. Selon le CLiki, il est obsolète, mais il contient de meilleures descriptions de la macro en boucle et du format que le Common Lisp Hyperspec.
- On Lisp de Paul Graham est un excellent livre pour les Lispers expérimentés.

- Let Over Lambda de Doug Hoyte est un livre avancé sur les macros Lisp. Plusieurs personnes ont recommandé que vous soyez à l'aise avec On Lisp avant de lire ce livre et que le démarrage soit lent.

Références en ligne

- Le Common Lisp Hyperspec est *le* document de référence de langage pour Common Lisp.
- Le Common Lisp Cookbook est une liste de recettes Lisp utiles. Contient également une liste d'autres sources en ligne d'informations sur la CL.
- Common Lisp Quick Reference a des feuilles de référence imprimables Lisp.
- Lispdoc.com recherche plusieurs sources d'informations Lisp (Common Lisp pratique, Lisp réussi, On Lisp, HyperSpec) pour la documentation.
- L1sp.org est un service de redirection pour la documentation.

Livres hors ligne

Ce sont des livres que vous devrez probablement acheter ou prêter dans une bibliothèque.

- ANSI Common Lisp de Paul Graham .
- Recettes Lisp communes par Edmund Weitz .
- Paradigmes de la programmation de l'intelligence artificielle a de nombreuses applications intéressantes de Lisp, mais n'est pas une bonne référence pour l'IA.

Communautés en ligne

- Le CLiki a une excellente page de démarrage . Une excellente ressource pour tout ce qui concerne CL. Possède une longue liste de livres Lisp .
- Common Lisp subreddit a beaucoup de liens utiles et de documents de référence dans la barre latérale.
- IRC: #lisp, #ccl, #sbcl et autres sur Freenode .
- Common-Lisp.net fournit l'hébergement pour de nombreux projets communs et groupes d'utilisateurs.

Bibliothèques

- Quicklisp est un gestionnaire de bibliothèque pour Common Lisp et possède une longue liste de bibliothèques prises en charge .
- Quickdocs héberge la documentation de la bibliothèque pour de nombreuses bibliothèques CL.
- Awesome CL est une liste organisée par les communautés de bibliothèques, de frameworks et d'autres éléments brillants, classés par catégorie.

Environnements Lisp préemballés

Ce sont des environnements d'édition Lisp faciles à installer et à utiliser car tout ce dont vous avez besoin est pré-packagé et pré-configuré.

- Portacle est un environnement Common Lisp portable et multiplateforme. Il embarque un Emacs légèrement personnalisé avec Slime, SBCL (une implémentation Common Lisp

populaire), Quicklisp et Git. Aucune installation nécessaire, c'est donc un moyen très rapide et facile de démarrer.
- Lispbox est un IDE (Emacs + SLIME), un environnement Common Lisp (Clozure Common Lisp) et un gestionnaire de bibliothèque (Quicklisp), pré-emballés en tant qu'archives pour Windows, Mac OSX et Linux. Descendant de "Lisp in a Box" Recommandé dans le livre Practical Common Lisp.
- Non préemballé, mais SLIME transforme Emacs en IDE Common Lisp et dispose d'un manuel d'utilisation pour vous aider à démarrer. Nécessite une implémentation Common Lisp distincte.

Implémentations Lisp communes

Cette section répertorie certaines implémentations CL et leurs manuels. Sauf indication contraire, il s'agit d'implémentations de logiciels libres. Voir aussi la liste des implémentations Common Lisp du logiciel libre , et la liste des implémentations Common Lisp commerciales de Wikipedia .

- Allegro Common Lisp (ACL) et manuel . Commercial, mais a une édition Express gratuite et des vidéos de formation sur Youtube .
- CLISP et manuel .
- Clozure Common Lisp (CCL) et manuel .
- L'université Common Lisp de l'Université Carnegie Mellon (CMUCL) dispose d'un manuel et d'une autre page d' informations utiles .
- Embout Common Lisp (ECL) et manuel .
- LispWorks et manuel . Commercial, mais a une édition personnelle avec quelques limitations .
- Steel Bank Common Lisp (SBCL) et manuel .
- Scieneer Common Lisp (SCL) et manual sont des implémentations commerciales Linux et Unix, mais ont une version d'évaluation gratuite et une utilisation non commerciale sans restriction .

Lire Commencer avec Common-Lisp en ligne: https://riptutorial.com/fr/common-lisp/topic/534/commencer-avec-common-lisp

Chapitre 2: ANSI Common Lisp, le standard de langage et sa documentation

Examples

Common Lisp HyperSpec

Common Lisp a un standard, qui a été publié en 1994 comme norme ANSI.

Le Common Lisp HyperSpec , abrégé CLHS, fourni par LispWorks est une documentation HTML souvent utilisée, dérivée du document standard. HyperSpec peut également être téléchargé et utilisé localement .

Les environnements de développement Lisp courants permettent généralement de rechercher la documentation HyperSpec pour les symboles Lisp.

- Pour GNU Emacs, il y a clhs.el.
- SLIME pour GNU Emacs fournit une version de hyperspec.el .

Voir aussi: cliki sur CLHS

Déclarations de syntaxe EBNF dans la documentation

La norme ANSI CL utilise une notation syntaxique EBNF étendue. La documentation dupliquée sur Stackoverflow doit utiliser la même notation syntaxique pour réduire la confusion.

Exemple:

```
specialized-lambda-list::=
     ({var | (var parameter-specializer-name)}*
      [&optional {var | (var [initform [supplied-p-parameter] ])}*]
      [&rest var]
      [&key{var | ({var | (keywordvar)} [initform [supplied-p-parameter] ])}*
           [&allow-other-keys] ]
      [&aux {var | (var [initform] )}*] )
```

Notation:

- `[foo]` -> zéro ou un `foo`
- `{foo}*` -> zéro ou plus `foo`
- `foo | bar` -> `foo` ou `bar`

Common Lisp the Language, 2e édition, par Guy L. Steele Jr.

Ce livre est connu sous le nom de CLtL2.

Ceci est la deuxième édition du livre Common Lisp the Language. Il a été publié en 1990, avant

que la norme ANSI CL ne soit définitive. Il a pris la définition de la langue d'origine de la première édition (publiée en 1984) et décrit tous les changements intervenus dans le processus de normalisation jusqu'en 1990, ainsi que certaines extensions (comme la fonction d'itération SERIES).

Remarque: CLTL2 décrit une version de Common Lisp qui diffère légèrement de la norme publiée en 1994. Utilisez donc toujours la norme, et non CLtL2, comme référence.

CLtL2 peut toujours être utile, car il fournit des informations introuvables dans le document de spécification Common Lisp.

Il existe une version HTML de Common Lisp the Language, 2e édition .

CLiki - Révisions et clarifications ANSI proposées

Sur CLiki, un Wiki pour Common Lisp et *un* logiciel *gratuit* Common Lisp, une liste de révisions et de clarifications ANSI proposées est maintenue.

Comme le standard Common Lisp n'a pas changé depuis 1994, les utilisateurs ont trouvé plusieurs problèmes avec le document de spécification. Ceux-ci sont documentés sur la page CLiki.

Common Lisp Guide de référence rapide

Le Guide de référence rapide de Common Lisp est un document qui peut être imprimé et relié sous forme de livret dans différentes dispositions pour avoir une référence rapide imprimée pour Common Lisp.

Le standard ANSI Common Lisp au format Texinfo (particulièrement utile pour GNU Emacs)

GNU Emacs utilise un format spécial pour la documentation: *info* .

Le standard Common Lisp a été converti au format Texinfo, qui peut être utilisé pour créer une documentation avec le lecteur d' *informations* de GNU Emacs.

Voir ici: dpans2texi.el convertit les sources TeX du projet de norme ANSI Common Lisp (dpANS) au format Texinfo.

Une autre version a été faite pour GCL: gcl.info.tgz .

Lire ANSI Common Lisp, le standard de langage et sa documentation en ligne: https://riptutorial.com/fr/common-lisp/topic/2900/ansi-common-lisp--le-standard-de-langage-et-sa-documentation

Chapitre 3: ASDF - Une autre installation de définition de système

Remarques

ASDF - Une autre installation de définition de système

ASDF est un outil permettant de spécifier la manière dont les systèmes Common Lisp sont constitués de composants (sous-systèmes et fichiers) et d'opérer sur ces composants dans le bon ordre afin qu'ils puissent être compilés, chargés, testés, etc.

Examples

Système ASDF simple avec une structure de répertoire plate

Considérez ce projet simple avec une structure de répertoire plate:

```
example
|-- example.asd
|-- functions.lisp
|-- main.lisp
|-- packages.lisp
`-- tools.lisp
```

Le fichier `example.asd` n'est en réalité qu'un autre fichier Lisp avec un peu plus qu'un appel de fonction spécifique à ASDF. En supposant que votre projet dépend des systèmes `drakma` et `clsql` , son contenu peut être quelque chose comme ceci:

```
(asdf:defsystem :example
    :description "a simple example project"
    :version "1.0"
    :author "TheAuthor"
    :depends-on (:clsql
                 :drakma)
    :components ((:file "packages")
                 (:file "tools" :depends-on ("packages"))
                 (:file "functions" :depends-on ("packages"))
                 (:file "main" :depends-on ("packages"
                                            "functions"))))
```

Lorsque vous chargez ce fichier Lisp, vous indiquez à ASDF votre `:example` système, mais vous ne chargez pas encore le système lui-même. Cela se fait soit par `(asdf:require-system :example)` ou `(ql:quickload :example)` .

Et lorsque vous chargez le système, ASDF va:

1. Charger les dépendances - dans ce cas, les systèmes ASDF `clsql` et `drakma`
2. *Compiler et charger* les composants de votre système, c'est-à-dire les fichiers Lisp, en

fonction des dépendances données

1. `packages` premier (pas de dépendances)
2. `functions` après les `packages` (comme cela ne dépend que des `packages`), mais avant le `main` (qui en dépend)
3. `main functions` après (comme cela dépend des `packages` et des `functions`)
4. `tools` tout moment après les `packages`

Garder en tete:

- Entrez les dépendances nécessaires (par exemple, les définitions de macros sont nécessaires avant utilisation). Si vous ne le faites pas, ASDF commettra une erreur lors du chargement de votre système.
- Tous les fichiers listés se terminent par `.lisp` mais ce postfix devrait être déposé dans le script asdf
- Si votre système porte le même nom que son fichier `.asd` et que vous déplacez (ou créez un lien symbolique) son dossier dans le dossier `quicklisp/local-projects/` , vous pouvez alors charger le projet en utilisant `(ql:quickload "example")` .
- Les bibliothèques dont dépend votre système doivent être connues d'ASDF (via la variable `ASDF:*CENTRAL-REGISTRY`) ou Quicklisp (via la `QUICKLISP-CLIENT:*LOCAL-PROJECT-DIRECTORIES*` ou disponibles dans l'un de ses dists)

Comment définir une opération de test pour un système

```
(in-package #:asdf-user)

(defsystem #:foo
  :components ((:file "foo"))
  :in-order-to ((asdf:test-op (asdf:load-op :foo)))
  :perform (asdf:test-op (o c)
                     (uiop:symbol-call :foo-tests 'run-tests)))

(defsystem #:foo-tests
  :name "foo-test"
  :components ((:file "tests")))

;; Afterwards to run the tests we type in the REPL
(asdf:test-system :foo)
```

Remarques:

- Nous supposons que le *système* : foo-tests définit un *paquet* nommé "FOO-TESTS"
- `run-tests` est le point d'entrée pour le coureur de test
- uoip: l'appel de symbole permet de définir une méthode qui appelle une fonction qui n'a pas encore été lue. Le paquet dans lequel la fonction est définie n'existe pas lorsque nous définissons le système

Dans quel paquet dois-je définir mon système ASDF?

ASDF fournit le package `ASDF-USER` aux développeurs pour définir leurs packages.

Lire ASDF - Une autre installation de définition de système en ligne: https://riptutorial.com/fr/common-lisp/topic/670/asdf---une-autre-installation-de-definition-de-systeme

Chapitre 4: Boucles de base

Syntaxe

- (do ({var | (var [init-form [step-form]])} *) (end-test-form result-form *) déclaration * {tag | statement} *)
- (do * ({var | (var [init-form [step-form]])} *) (end-test-form result-form *) déclaration * {tag | statement} *)
- (dolist (var list-form [result-form]) déclaration * {tag | statement} *)
- (dotimes (var count-form [result-form]) déclaration * {tag | statement} *)

Examples

dotimes

`dotimes` est une macro pour une itération d'entier sur une seule variable de 0 sous une valeur de paramètre. Un des exemples simples serait:

```
CL-USER> (dotimes (i 5)
           (print i))

0
1
2
3
4
NIL
```

Notez que `NIL` est la valeur renvoyée, puisque nous n'en avons pas fourni nous-mêmes; la variable commence à 0 et dans toute la boucle devient des valeurs de 0 à N-1. Après la boucle, la variable devient le N:

```
CL-USER> (dotimes (i 5 i))
5

CL-USER> (defun 0-to-n (n)
           (let ((list ()))
             (dotimes (i n (nreverse list))
               (push i list))))
0-TO-N
CL-USER> (0-to-n 5)
(0 1 2 3 4)
```

faire une liste

`dolist` est une macro en boucle créée pour parcourir facilement les listes. Une des utilisations les plus simples serait:

```
CL-USER> (dolist (item '(a b c d))
           (print item))

A
B
C
D
NIL ; returned value is NIL
```

Notez que puisque nous n'avons pas fourni de valeur de retour, `NIL` est renvoyé (et A, B, C, D sont imprimés dans `*standard-output*`).

`dolist` peut également renvoyer des valeurs:

```
;;This may not be the most readable summing function.
(defun sum-list (list)
  (let ((sum 0))
    (dolist (var list sum)
      (incf sum var))))

CL-USER> (sum-list (list 2 3 4))
9
```

Boucle simple

La macro de **boucle** a deux formes: la forme "simple" et la forme "étendue". La forme étendue est traitée dans une autre rubrique de documentation, mais la boucle simple est utile pour les boucles très simples.

La forme en **boucle** simple prend un certain nombre de formes et les répète jusqu'à ce que la boucle soit sortie en utilisant **return** ou une autre sortie (par exemple, **throw**).

```
(let ((x 0))
  (loop
     (print x)
     (incf x)
     (unless (< x 5)
       (return))))

0
1
2
3
4
NIL
```

Lire Boucles de base en ligne: https://riptutorial.com/fr/common-lisp/topic/2053/boucles-de-base

Chapitre 5: Citation

Syntaxe

- (objet citation) -> objet

Remarques

Certains objets (par exemple des symboles de mots-clés) ne doivent pas être cités car ils s'évaluent eux-mêmes.

Examples

Exemple de citation simple

Quote est un **opérateur spécial** qui empêche l'évaluation de son argument. Il renvoie son argument, non évalué.

```
CL-USER> (quote a)
A

CL-USER> (let ((a 3))
           (quote a))
A
```

'est un alias pour l'opérateur spécial QUOTE

La `'thing` notation est égale à `(quote thing)` .

Le *lecteur* fera l'expansion:

```
> (read-from-string "'a")
(QUOTE A)
```

Les citations sont utilisées pour empêcher toute évaluation ultérieure. L'objet cité est évalué à lui-même.

```
> 'a
A

> (eval '+ 1 2)
3
```

Si les objets cités sont modifiés de manière destructive, les conséquences ne sont pas définies!

Évitez les opérations destructrices sur les objets cités. Les objets cotés sont des objets littéraux. Ils sont peut-être intégrés au code d'une manière ou d'une autre. Comment cela fonctionne et les effets des modifications ne sont pas spécifiés dans le standard Common Lisp, mais cela peut avoir des conséquences indésirables, comme modifier des données partagées, essayer de modifier des données protégées en écriture ou créer des effets secondaires involontaires.

~~(delete 5 '(1 2 3 4 5))~~

Citation et auto-évaluation d'objets

Notez que de nombreux types de données n'ont pas besoin d'être cités, car ils s'évaluent eux-mêmes. `QUOTE` est particulièrement utile pour les symboles et les listes, pour empêcher l'évaluation en tant que formulaires Lisp.

Il n'est pas nécessaire de citer un exemple pour d'autres types de données pour empêcher l'évaluation: chaînes, nombres, caractères, objets CLOS, ...

Voici un exemple pour les chaînes. Les résultats de l'évaluation sont des chaînes, qu'elles soient ou non citées dans la source.

```
> (let ((some-string-1 "this is a string")
        (some-string-2 '"this is a string with a quote in the source")
        (some-string-3 (quote "this is another string with a quote in the source")))
    (list some-string-1 some-string-2 some-string-3))

("this is a string"
 "this is a string with a quote in the source"
 "this is another string with a quote in the source")
```

Citer pour les objets est donc facultatif.

Lire Citation en ligne: https://riptutorial.com/fr/common-lisp/topic/1315/citation

Chapitre 6: CLOS - le système d'objet Lisp commun

Examples

Créer une classe CLOS de base sans parents

Une classe CLOS est décrite par:

- un nom
- une liste de superclasses
- une liste de slots
- autres options comme la documentation

Chaque slot a:

- un nom
- un formulaire d'initialisation (facultatif)
- un argument d'initialisation (facultatif)
- un type (facultatif)
- une chaîne de documentation (facultatif)
- fonctions d'accesseur, de lecteur et / ou de graveur (facultatif)
- autres options comme l'allocation

Exemple:

```
(defclass person ()
  ((name
    :initform      "Erika Mustermann"
    :initarg       :name
    :type          string
    :documentation "the name of a person"
    :accessor      person-name)
   (age
    :initform      25
    :initarg       :age
    :type          number
    :documentation "the age of a person"
    :accessor      person-age))
  (:documentation "a CLOS class for persons with name and age"))
```

Une méthode d'impression par défaut:

```
(defmethod print-object ((p person) stream)
  "The default print-object method for a person"
  (print-unreadable-object (p stream :type t :identity t)
    (with-slots (name age) p
      (format stream "Name: ~a, age: ~a" name age))))
```

Création d'instances:

```
CL-USER > (make-instance 'person)
#<PERSON Name: Erika Mustermann, age: 25 4020169AB3>

CL-USER > (make-instance 'person :name "Max Mustermann" :age 24)
#<PERSON Name: Max Mustermann, age: 24 4020169FEB>
```

Mixins et interfaces

Common Lisp n'a pas d'interfaces dans le sens où certains langages (par exemple, Java), et il y a moins besoin de ce type d'interface étant donné que Common Lisp prend en charge plusieurs fonctions d'héritage et génériques. Cependant, le même type de modèle peut être facilement réalisé en utilisant des classes mixin. Cet exemple montre la spécification d'une interface de collection avec plusieurs fonctions génériques correspondantes.

```
;; Specification of the COLLECTION "interface"

(defclass collection () ()
  (:documentation "A collection mixin."))

(defgeneric collection-elements (collection)
  (:documentation "Returns a list of the elements in the collection."))

(defgeneric collection-add (collection element)
  (:documentation "Adds an element to the collection."))

(defgeneric collection-remove (collection element)
  (:documentation "Removes the element from the collection, if it is present."))

(defgeneric collection-empty-p (collection)
  (:documentation "Returns whether the collection is empty or not."))

(defmethod collection-empty-p ((c collection))
  "A 'default' implementation of COLLECTION-EMPTY-P that tests
whether the list returned by COLLECTION-ELEMENTS is the empty
list."
  (endp (collection-elements c)))
```

Une implémentation de l'interface est juste une classe qui a le mixin comme l'une de ses super-classes et des définitions des fonctions génériques appropriées. (À ce stade, notez que la classe mixin est uniquement destinée à signaler l'intention que la classe implémente "l'interface". Cet exemple fonctionnerait tout aussi bien avec quelques fonctions génériques et une documentation indiquant qu'il existe des méthodes sur la fonction pour la classe.)

```
;; Implementation of a sorted-set class

(defclass sorted-set (collection)
  ((predicate
    :initarg :predicate
    :reader sorted-set-predicate)
   (test
    :initarg :test
    :initform 'eql
```

```
    :reader sorted-set-test)
   (elements
    :initform '()
    :accessor sorted-set-elements
    ;; We can "implement" the COLLECTION-ELEMENTS function, that is,
    ;; define a method on COLLECTION-ELEMENTS, simply by making it
    ;; a reader (or accessor) for the slot.
    :reader collection-elements)))

(defmethod collection-add ((ss sorted-set) element)
  (unless (member element (sorted-set-elements ss)
                  :test (sorted-set-test ss))
    (setf (sorted-set-elements ss)
          (merge 'list
                 (list element)
                 (sorted-set-elements ss)
                 (sorted-set-predicate ss)))))

(defmethod collection-remove ((ss sorted-set) element)
  (setf (sorted-set-elements ss)
        (delete element (sorted-set-elements ss))))
```

Enfin, on peut voir à quoi ressemble une instance de la classe **triée** quand on utilise les fonctions "interface":

```
(let ((ss (make-instance 'sorted-set :predicate '<)))
  (collection-add ss 3)
  (collection-add ss 4)
  (collection-add ss 5)
  (collection-add ss 3)
  (collection-remove ss 5)
  (collection-elements ss))
;; => (3 4)
```

Lire CLOS - le système d'objet Lisp commun en ligne: https://riptutorial.com/fr/common-lisp/topic/673/clos---le-systeme-d-objet-lisp-commun

Chapitre 7: Contre les cellules et les listes

Examples

Listes comme convention

Certaines langues incluent une structure de données de liste. Common Lisp et d'autres langages de la famille Lisp utilisent largement les listes (et le nom Lisp est basé sur l'idée d'un processeur LISt). Cependant, Common Lisp n'inclut pas réellement un type de données de liste primitif. Au lieu de cela, les listes existent par convention. La convention repose sur deux principes:

1. Le symbole **nul** est la liste vide.
2. Une liste non vide est une *contre cellule* dont la *voiture* est le premier élément de la liste et dont le *cdr* est le reste de la liste.

C'est tout ce qu'il y a aux listes. Si vous avez lu l'exemple appelé *Qu'est-ce qu'une cellule contre?* , alors vous savez qu'une cellule contre dont la voiture est X et dont le cdr est Y peut être écrite comme **(X. Y)** . Cela signifie que nous pouvons écrire des listes basées sur les principes ci-dessus. La liste des éléments 1, 2 et 3 est simplement:

```
(1 . (2 . (3 . nil)))
```

Cependant, étant donné que les listes sont si courantes dans la famille de langues Lisp, il existe des conventions d'impression spéciales au-delà de la simple notation par paires pour les cellules.

1. Le symbole **nil** peut aussi être écrit comme **()** .
2. Lorsque le cdr d'une cellule contre est une autre liste (soit une **(** ou une cellule contre)), au lieu d'écrire la cellule contre avec la notation en pointillés, la "liste de notation" est utilisée.

La notation de liste est illustrée plus clairement par plusieurs exemples:

```
(x . (y . z))   === (x y . z)
(x . NIL)       === (x)
(1 . (2 . NIL)) === (1 2)
(1 . ())        === (1)
```

L'idée est que les éléments de la liste sont écrits dans l'ordre entre parenthèses jusqu'à ce que le cdr final de la liste soit atteint. Si le cdr final est **nul** (la liste vide), la parenthèse finale est écrite. Si le cdr final n'est pas **nul** (auquel cas la liste s'appelle une *liste incorrecte*), alors un point est écrit et ensuite ce dernier est écrit.

Qu'est-ce qu'une cellule contre?

Une cellule cons, également appelée paire pointée (en raison de sa représentation imprimée), est simplement une paire de deux objets. Une contre cellule est créée par la fonction `cons` , et les éléments de la paire sont extraits à l'aide des fonctions `car` et `cdr` .

```
(cons "a" 4)
```

Par exemple, cela retourne une paire dont le premier élément (qui peut être extrait avec `car`) est `"a"` , et dont le deuxième élément (qui peut être extrait avec `cdr`) est `4` .

```
(car (cons "a" 4))
;;=> "a"

(cdr (cons "a" 4))
;;=> 3
```

Les cellules peuvent être imprimées en notation par *points* :

```
(cons 1 2)
;;=> (1 . 2)
```

Cons cellules peuvent également être lus dans la notation de la paire en pointillés, de sorte que

```
(car '(x . 5))
;;=> x

(cdr '(x . 5))
;;=> 5
```

(La forme imprimée des contre-cellules peut aussi être un peu plus compliquée. Pour en savoir plus à ce sujet, voyez l'exemple sur les cellules contre comme des listes.)

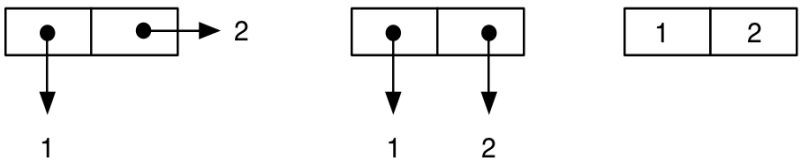
C'est tout; par contre les cellules ne sont que des paires d'éléments créés par la fonction `cons` , et les éléments peuvent être extraits avec `car` et `cdr` . En raison de leur simplicité, les cellules en opposition peuvent constituer un élément de base utile pour des structures de données plus complexes.

Esquisse contre des cellules

Pour mieux comprendre la sémantique des conses et des listes, une représentation graphique de ce type de structures est souvent utilisée. Une cellule cons est généralement représentée avec deux cases en contact, qui contiennent soit deux flèches qui pointent vers la `car` et les valeurs `cdr` , soit directement les valeurs. Par exemple, le résultat de:

```
(cons 1 2)
;; -> (1 . 2)
```

peut être représenté avec l'un de ces dessins:



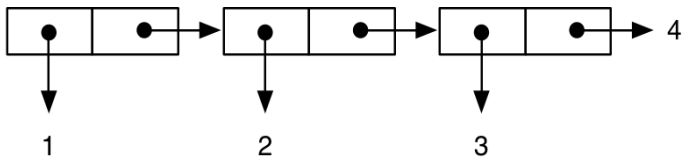
Notez que ces représentations sont purement conceptuelles et ne dénotent pas le fait que les

valeurs sont *contenues* dans la cellule ou sont *pointées* depuis la cellule: en général, cela dépend de l'implémentation, du type des valeurs, du niveau d'optimisation, etc. Dans le reste de l'exemple, nous utiliserons le premier type de dessin, celui qui est le plus couramment utilisé.

Donc, par exemple:

```
(cons 1 (cons 2 (cons 3 4)))   ; improper “dotted” list
;; -> (1 2 3 . 4)
```

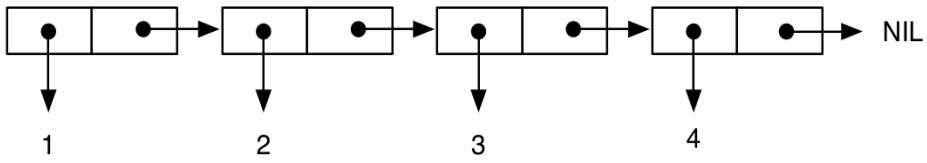
est représenté comme:



tandis que:

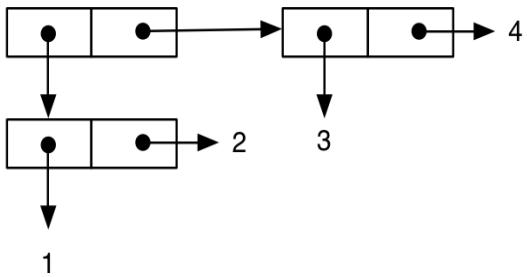
```
(cons 1 (cons 2 (cons 3 (cons 4 nil))))  ;; proper list, equivalent to: (list 1 2 3 4)
;; -> (1 2 3 4)
```

est représenté par:



Voici une structure arborescente:

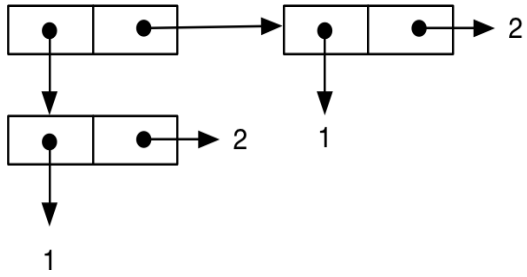
```
(cons (cons 1 2) (cons 3 4))
;; -> ((1 . 2) 3 . 4)         ; note the printing as an improper list
```



L'exemple final montre comment cette notation peut nous aider à comprendre les aspects sémantiques importants du langage. Tout d'abord, nous écrivons une expression similaire à la précédente:

```
(cons (cons 1 2) (cons 1 2))
;; -> ((1 . 2) 1 . 2)
```

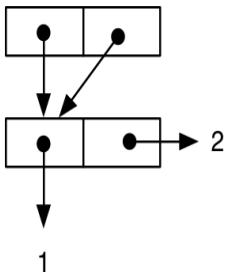
qui peut être représenté de la manière habituelle comme:



Ensuite, nous écrivons une expression différente, apparemment équivalente à la précédente, ce qui semble confirmé par une représentation imprimée du résultat:

```
(let ((cell-a (cons 1 2)))
  (cons cell-a cell-a))
;; -> ((1 . 2) 1 . 2)
```

Mais, si l' on trace le diagramme, nous pouvons voir que la sémantique de l'expression est différente, puisque la *même* cellule est la valeur à la *fois* de la `car` partie et la `cdr` partie des extérieurs `cons` (ce qui est, `cell-a` est *partagée)* :



et les tests suivants permettent de vérifier que la sémantique des deux résultats est réellement différente au niveau du langage:

```
(let ((c1 (cons (cons 1 2) (cons 1 2)))
      (c2 (let ((cell-a (cons 1 2)))
            (cons cell-a cell-a))))
  (list (eq (car c1) (cdr c1))
        (eq (car c2) (cdr c2)))
;; -> (NIL T)
```

Le premier `eq` est *faux* car la `car` et `cdr` de `c1` sont structurellement égaux (c'est *vrai* par `equal`), mais ne sont pas "identiques" (ie "la même structure partagée"), alors que dans le second test le résultat est *vrai* puisque le `car` et `cdr` de `c2` sont *identiques* , c'est-à-dire qu'elles ont *la même structure* .

Lire Contre les cellules et les listes en ligne: https://riptutorial.com/fr/common-lisp/topic/2622/contre-les-cellules-et-les-listes

Chapitre 8: Correspondance de motif

Examples

Vue d'ensemble

Les deux bibliothèques principales fournissant une correspondance de modèle dans Common Lisp sont Optima et Trivia . Les deux fournissent une API et une syntaxe de correspondance similaires. Cependant, trivia fournit une interface unifiée pour étendre la correspondance, `defpattern` .

Envoi des demandes de clack

Comme une demande de clack est représentée en tant que plist, nous pouvons utiliser la correspondance de modèle comme point d'entrée de l'application clack pour acheminer la requête vers les contrôleurs appropriés.

```
(defvar *app*
  (lambda (env)
    (match env
      ((plist :request-method :get
              :request-uri uri)
       (match uri
         ("/" (top-level))
         ((ppcre "/tag/(\\w+)/$" name) (tag-page name)))))))
```

Remarque: pour lancer `*app*` nous le transmettons à clackup. ej `(clack:clackup *app*)`

match défun

En utilisant une correspondance de motif, on peut entrelacer la définition de fonction et la correspondance de motif, comme pour SML.

```
(trivia:defun-match fib (index)
  "Return the corresponding term for INDEX."
  (0 1)
  (1 1)
  (index (+ (fib (1- index)) (fib (- index 2)))))

(fib 5)
;; => 8
```

Modèles de constructeur

Les cons-cells, les structures, les vecteurs, les listes et autres peuvent être associés à des modèles de constructeur.

```
(loop for i from 1 to 30
```

```
      do (format t "~5<~a~;~>"
                 (match (cons (mod i 3)
                              (mod i 5))
                   ((cons 0 0) "Fizzbuzz")
                   ((cons 0 _) "Fizz")
                   ((cons _ 0) "Buzz")
                   (_ i)))
      when (zerop (mod i 5)) do (terpri))
; 1    2    Fizz 4    Buzz
; Fizz 7    8    Fizz Buzz
; 11   Fizz 13   14   Fizzbuzz
; 16   17   Fizz 19   Buzz
; Fizz 22   23   Fizz Buzz
; 26   Fizz 28   29   Fizzbuzz
```

Motif de garde

Les patrons de garde peuvent être utilisés pour vérifier qu'une valeur satisfait à une forme de test arbitraire.

```
(dotimes (i 5)
  (format t "~d: ~a~%"
          i (match i
              ((guard x (oddp x)) "Odd!")
              (_ "Even!"))))
; 0: Even!
; 1: Odd!
; 2: Even!
; 3: Odd!
; 4: Even!
```

Lire Correspondance de motif en ligne: https://riptutorial.com/fr/common-lisp/topic/2933/correspondance-de-motif

Chapitre 9: Créer des fichiers binaires

Examples

BuildingApp

Les binaires Common Lisp autonomes peuvent être `buildapp` avec `buildapp` . Avant de pouvoir l'utiliser pour générer des fichiers binaires, nous devons l'installer et le construire.

La façon la plus simple de savoir comment utiliser `quicklisp` et Common Lisp (cet exemple utilise [`sbcl`], mais cela ne devrait pas faire la différence.)

```
$ sbcl

This is SBCL 1.3.5.nixos, 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.

* (ql:quickload :buildapp)
To load "buildapp":
  Load 1 ASDF system:
    buildapp
; Loading "buildapp"

(:BUILDAPP)

* (buildapp:build-buildapp)
;; loading system "buildapp"
[undoing binding stack and other enclosing state... done]
[saving current Lisp image into /home/inaimathi/buildapp:
writing 4800 bytes from the read-only space at 0x20000000
writing 3216 bytes from the static space at 0x20100000
writing 47349760 bytes from the dynamic space at 0x1000000000
done]
NIL

* (quit)

$ ls -lh buildapp
-rwxr-xr-x 1 inaimathi inaimathi 46M Aug 13 20:12 buildapp
$
```

Une fois que ce binaire est construit, vous pouvez l'utiliser pour construire des binaires de vos programmes Common Lisp. Si vous avez l'intention de le faire beaucoup, vous devriez probablement le placer quelque part sur votre `PATH` afin de pouvoir l'exécuter avec `buildapp` depuis n'importe quel répertoire.

Buildapp Hello World

Le binaire le plus simple possible

1. N'a pas de dépendances
2. Ne prend pas d'arguments en ligne de commande
3. Juste écrit "Bonjour tout le monde!" `stdout`

Après avoir construit `buildapp` , vous pouvez simplement ...

```
$ buildapp --eval '(defun main (argv) (declare (ignore argv)) (write-line "Hello, world!"))' --entry main --output hello-world
[undoing binding stack and other enclosing state... done]
[saving current Lisp image into hello-world:
writing 4800 bytes from the read-only space at 0x20000000
writing 3216 bytes from the static space at 0x20100000
writing 43220992 bytes from the dynamic space at 0x1000000000
done]

$ ./hello-world
Hello, world!

$
```

Buildapp Bonjour Web World

Un exemple plus réaliste concerne un projet que vous `--eval` avec plusieurs fichiers sur disque (plutôt qu'une option `--eval` transmise à `buildapp`), et certaines dépendances à `buildapp` .

Étant donné que des choses arbitraires peuvent se produire pendant la recherche et le chargement de systèmes `asdf` (y compris le chargement d'autres systèmes potentiellement non liés), il ne suffit pas d'inspecter les fichiers `asd` des projets dont vous `asdf` pour déterminer ce que vous devez charger. . L'approche générale consiste à utiliser `quicklisp` pour charger le système cible, puis à appeler `ql:write-asdf-manifest-file` pour écrire un manifeste complet de tout ce qui est chargé.

Voici un système de jouet construit avec `hunchentoot` pour illustrer comment cela pourrait se produire dans la pratique:

```
;;;; buildapp-hello-web-world.asd

(asdf:defsystem #:buildapp-hello-web-world
  :description "An example application to use when getting familiar with buildapp"
  :author "inaimathi <leo.zovic@gmail.com>"
  :license "Expat"
  :depends-on (#:hunchentoot)
  :serial t
  :components ((:file "package")
               (:file "buildapp-hello-web-world"))
```

```
;;;; package.lisp

(defpackage #:buildapp-hello-web-world
  (:use #:cl #:hunchentoot))
```

```
;;;; buildapp-hello-web-world.lisp

(in-package #:buildapp-hello-web-world)

(define-easy-handler (hello :uri "/") ()
  (setf (hunchentoot:content-type*) "text/plain")
  "Hello Web World!")

(defun main (argv)
  (declare (ignore argv))
  (start (make-instance 'easy-acceptor :port 4242))
  (format t "Press any key to exit...~%")
  (read-char))
```

```
;;;; build.lisp
(ql:quickload :buildapp-hello-web-world)
(ql:write-asdf-manifest-file "/tmp/build-hello-web-world.manifest")
(with-open-file (s "/tmp/build-hello-web-world.manifest" :direction :output :if-exists
:append)
  (format s "~a~%" (merge-pathnames
            "buildapp-hello-web-world.asd"
            (asdf/system:system-source-directory
             :buildapp-hello-web-world))))
```

```
#### build.sh
sbcl --load "build.lisp" --quit

buildapp --manifest-file /tmp/build-hello-web-world.manifest --load-system hunchentoot --load-
system buildapp-hello-web-world --output hello-web-world --entry buildapp-hello-web-world:main
```

Une fois que vous avez ces fichiers enregistrés dans un répertoire nommé `buildapp-hello-web-world` , vous pouvez le faire

```
$ cd buildapp-hello-web-world/

$ sh build.sh
This is SBCL 1.3.7.nixos, 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.
To load "cffi":
  Load 1 ASDF system:
    cffi
; Loading "cffi"
........
To load "buildapp-hello-web-world":
  Load 1 ASDF system:
    buildapp-hello-web-world
; Loading "buildapp-hello-web-world"
....
;; loading system "cffi"
;; loading system "hunchentoot"
;; loading system "buildapp-hello-web-world"
[undoing binding stack and other enclosing state... done]
```

```
[saving current Lisp image into hello-web-world:
writing 4800 bytes from the read-only space at 0x20000000
writing 4624 bytes from the static space at 0x20100000
writing 66027520 bytes from the dynamic space at 0x1000000000
done]

$ ls -lh hello-web-world
-rwxr-xr-x 1 inaimathi inaimathi 64M Aug 13 21:17 hello-web-world
```

Cela produit un binaire qui correspond exactement à ce que vous pensez, compte tenu de ce qui précède.

```
$ ./hello-web-world
Press any key to exit...
```

Vous devriez alors pouvoir lancer un autre shell, faire `curl localhost:4242` et voir la réponse en texte clair de `Hello Web World!` faire imprimer

Lire Créer des fichiers binaires en ligne: https://riptutorial.com/fr/common-lisp/topic/5457/creer-des-fichiers-binaires

Chapitre 10: Egalité et autres prédicats de comparaison

Examples

La différence entre EQ et EQL

1. `EQ` vérifie si deux valeurs ont la même adresse de mémoire: en d'autres termes, il vérifie si les deux valeurs sont en fait le *même* objet *identique* . Donc, il peut être considéré comme le test d'identité, et ne devrait être appliqué *qu'aux* structures: conses, tableaux, structures, objets, généralement pour voir si vous traitez avec le même objet "atteint" par différents chemins différentes variables.

2. `EQL` vérifie si deux structures sont le même objet (comme `EQ`) *ou* si elles sont les mêmes valeurs non structurées (c'est-à-dire les mêmes valeurs numériques pour les nombres du même type ou les valeurs de caractères). Comme il inclut l'opérateur `EQ` et peut être utilisé également sur des valeurs non structurées, il s'agit de l'opérateur le plus important et le plus utilisé, et presque toutes les fonctions primitives nécessitant une comparaison d'égalité, comme `MEMBER` , *utilisent par défaut cet opérateur* .

Donc, il est toujours vrai que `(EQ XY)` implique `(EQL XY)` , tandis que le viceversa ne tient pas.

Quelques exemples peuvent faire la différence entre les deux opérateurs:

```
(eq 'a 'a)
T ;; => since two s-expressions (QUOTE A) are “internalized” as the same symbol by the reader.
(eq (list 'a) (list 'a))
NIL ;; => here two lists are generated as different objects in memory
(let* ((l1 (list 'a))
       (l2 l1))
  (eq l1 l2))
T ;; => here there is only one list which is accessed through two different variables
(eq 1 1)
?? ;; it depends on the implementation: it could be either T or NIL if integers are “boxed”
(eq #\a #\a)
?? ;; it depends on the implementation, like for numbers
(eq 2d0 2d0)
?? ;; => dependes on the implementation, but usually is NIL, since numbers in double
   ;;    precision are treated as structures in many implementations
(let ((a1 2d0)
      (a2 2d0))
  (eq a1 a2))
?? ;; => also in this case the results depends on the implementation
```

Essayons les mêmes exemples avec `EQL` :

```
(eql 'a 'a)
T ;; => equal because they are the same value, as for EQ
(eql (list 'a) (list 'a))
```

```
NIL ;; => different because they different objects in memory, as for EQ
(let* ((l1 (list 'a))
       (l2 l1))
  (eql l1 l2))
T ;; => as above
(eql 1 1)
T ;; they are the same number, even if integers are “boxed”
(eql #\a #\a)
T ;; they are the same character
(eql 2d0 2d0)
T ;; => they are the same number, even if numbers in double precision are treated as
   ;;   structures in many implementations
(let ((a1 2d0)
      (a2 2d0))
  (eql a1 a2))
T ;; => as before
(eql 2 2.0)
NIL;; => since the two values are of a different numeric type
```

À partir des exemples, nous pouvons voir pourquoi l'opérateur `EQL` devrait être utilisé pour vérifier de manière portable la «similitude» de toutes les valeurs, structurées et non structurées, et pourquoi de nombreux experts déconseillent l'utilisation de l' `EQ` en général.

Égalité structurelle avec EQUAL, EQUALP, TREE-EQUAL

Ces trois opérateurs implémentent une équivalence structurelle, c'est-à-dire qu'ils vérifient si des objets complexes différents ont une structure équivalente avec un composant équivalent.

`EQUAL` se comporte comme `EQL` pour les données non structurées, tandis que pour les structures construites par conses (listes et arbres) et les deux types spéciaux de tableaux, chaînes et vecteurs bit, il effectue *une équivalence structurelle* , renvoyant true sur deux structures isomorphes et les composants élémentaires sont égaux par `EQUAL` . Par exemple:

```
(equal (list 1 (cons 2 3)) (list 1 (cons 2 (+ 2 1))))
T ;; => since the two arguments are both equal to (1 (2 . 3))
(equal "ABC" "ABC")
T ;; => equality on strings
(equal "Abc" "ABC")
NIL ;; => case sensitive equality on strings
(equal '(1 . "ABC") '(1 . "ABC"))
T ;; => equal since it uses EQL on 1 and 1, and EQUAL on "ABC" and "ABC"
(let* ((a (make-array 3 :initial-contents '(1 2 3)))
       (b (make-array 3 :initial-contents '(1 2 3)))
       (c a))
  (values (equal a b)
          (equal a c)))
NIL ;; => the structural equivalence is not used for general arrays
T   ;; => a and c are alias for the same object, so it is like EQL
```

`EQUALP` renvoie true dans tous les cas où `EQUAL` est vrai, mais il utilise également une équivalence structurelle pour les tableaux de tout type et dimension, pour les structures et pour les tables de hachage (mais pas pour les instances de classe!). De plus, il utilise une équivalence insensible à la casse pour les chaînes.

```
(equalp "Abc" "ABC")
T ;; => case insensitive equality on strings
(equalp (make-array 3 :initial-contents '(1 2 3))
        (make-array 3 :initial-contents (list 1 2 (+ 2 1))))
T ;; => the structural equivalence is used also for any kind of arrays
(let ((hash1 (make-hash-table))
      (hash2 (make-hash-table)))
      (setf (gethash 'key hash1) 42)
      (setf (gethash 'key hash2) 42)
      (print (equalp hash1 hash2))
      (setf (gethash 'another-key hash1) 84)
      (equalp hash1 hash2))
T   ;; => after the first two insertions, hash1 and hash2 have the same keys and values
NIL ;; => after the third insertion, hash1 and hash2 have different keys and values
(progn (defstruct s) (equalp (make-s) (make-s)))
T ;; => the two values are structurally equal
(progn (defclass c () ()) (equalp (make-instance 'c) (make-instance 'c)))
NIL ;; => two structurally equivalent class instances returns NIL, it's up to the user to
    ;;    define an equality method for classes
```

Finalement, `TREE-EQUAL` peut être appliqué aux structures construites par `cons` et vérifie si elles sont isomorphes, comme `EQUAL` , mais en laissant à l'utilisateur le choix de la fonction à utiliser pour comparer les leafs, c'est-à-dire cela peut être de tout autre type de données (par défaut, le test utilisé sur atome est `EQL`). Par exemple:

```
(let ((l1 '(1 . ("A" . 2)))
      (l2 '(1 . ("A" . 2))))
  (tree-equal l1 l2 :test #'eql))
NIL ;; => since (eql "A" "A") gives NIL
(let ((l1 '(1 . ("A" . 2)))
      (l2 '(1 . ("A" . 2))))
  (tree-equal l1 l2 :test #'equal))
T ;; since (equal "A" "A") gives T
```

Opérateurs de comparaison sur des valeurs numériques

Les valeurs numériques peuvent être comparées avec `=` et les autres opérateurs de comparaison numérique (`/=` , `<` , `<=` , `>` , `>=`) qui ignorent la différence dans la représentation physique des différents types de nombres et effectuent la comparaison des valeurs mathématiques correspondantes . Par exemple:

```
(= 42 42)
T ;; => both number have the sme numeric type and the same value
(= 1 1.0 1d0)
T ;; => all the tree values represent the number 1, while for instance (eql 1 1d0) => NIL
  ;;    since it returns true only if the operands have the same numeric type
(= 0.0 -0.0)
T ;; => again, the value is the same, while (eql 0.0 -0.0) => NIL
(= 3.0 #c(3.0 0.0))
T ;; => a complex number with 0 imaginary part is equal to a real number
(= 0.33333333 11184811/33554432)
T ;; => since a float number is passed to RATIONAL before comparing it to another number
  ;; => and (RATIONAL 0.33333333) => 11184811/33554432 in 32-bit IEEE floats architectures
(= 0.33333333 0.33333334)
T ;; => since the result of RATIONAL on both numbers is equal in 32-bit IEEE floats
architectures
```

```
(= 0.33333333d0 0.33333334d0)
NIL ;; => since the RATIONAL of the two numbers in double precision is different
```

À partir de ces exemples, nous pouvons conclure que `=` est l'opérateur qui devrait normalement être utilisé pour effectuer une comparaison entre des valeurs numériques, sauf si nous voulons être strict sur le fait que deux valeurs numériques sont égales *seulement* si elles ont également le même type numérique, quel cas `EQL` devrait être utilisé.

Opérateurs de comparaison sur les caractères et les chaînes

Common Lisp dispose de 12 opérateurs de type spécifique pour comparer deux caractères, dont six sensibles à la casse et les autres insensibles à la casse. Leurs noms ont un modèle simple pour se rappeler facilement leur signification:

Sensible aux majuscules et minuscules	Insensible à la casse
CHAR =	CHAR-EQUAL
CHAR / =	CHAR-NOT-EQUAL
CHAR <	CHAR-LESSP
CHAR <=	CHAR-NOT-GREATERP
CHAR>	CHAR-GREATERP
CHAR> =	CHAR-NOT-LESSP

Deux caractères d'un même cas sont dans le même ordre que les codes correspondants obtenus par `CHAR-CODE` , tandis que pour les comparaisons insensibles à la casse, l'ordre relatif entre deux caractères quelconques parmi les deux plages `a..z` , `A..Z` dépend de l'implémentation . Exemples:

```
(char= #\a #\a)
T ;; => the operands are the same character
(char= #\a #\A)
NIL ;; => case sensitive equality
(CHAR-EQUAL #\a #\A)
T ;; => case insensitive equality
(char> #\b #\a)
T ;; => since in all encodings (CHAR-CODE #\b) is always greater than (CHAR-CODE #\a)
(char-greaterp #\b \#A)
T ;; => since for case insensitive the ordering is such that A=a, B=b, and so on,
  ;;    and furthermore either 9<A or Z<0.
(char> #\b #\A)
?? ;; => the result is implementation dependent
```

Pour les chaînes, les opérateurs spécifiques sont `STRING=` , `STRING-EQUAL` , etc. avec le mot STRING au lieu de CHAR. Deux chaînes sont égales si elles ont le même nombre de caractères *et* si les caractères correspondants sont égaux selon `CHAR=` ou `CHAR-EQUAL` si le test est sensible à la casse ou non.

L'ordre entre les chaînes est en ordre lexicographique sur les caractères des deux chaînes. Lorsqu'une comparaison d'ordre réussit, le résultat n'est pas `T` , mais l'indice du premier caractère dans lequel les deux chaînes diffèrent (ce qui équivaut à vrai, puisque tout objet non NIL est un «booléen généralisé» dans Common Lisp).

Une chose importante est que *tous* les opérateurs de comparaison sur la chaîne acceptent quatre paramètres de mots-clés: `start1` , `end1` , `start2` , `end2` , qui peuvent être utilisés pour restreindre la comparaison à une seule série de caractères contigus. L'index de début, s'il est omis, est 0, l'index de fin est omis est égal à la longueur de la chaîne et la comparaison est effectuée sur la sous-chaîne à partir du caractère index `:start` et fin avec le caractère indexé `:end - 1` inclus.

Enfin, notez qu'une chaîne, même avec un seul caractère, ne peut pas être comparée à un caractère.

Exemples:

```
(string= "foo" "foo")
T ;; => both strings have the same lenght and the characters are `CHAR=` in order
(string= "Foo" "foo")
NIL ;; => case sensitive comparison
(string-equal "Foo" "foo")
T ;; => case insensitive comparison
(string= "foobar" "barfoo" :end1 3 :start2 3)
T ;; => the comparison is perform on substrings
(string< "fooarr" "foobar")
3 ;; => the first string is lexicographically less than the second one and
  ;;   the first character different in the two string has index 3
(string< "foo" "foobar")
3 ;; => the first string is a prefix of the second and the result is its length
```

En tant que cas particulier, les opérateurs de comparaison de chaînes peuvent également être appliqués aux symboles et la comparaison est effectuée sur le `SYMBOL-NAME` du symbole. Par exemple:

```
(string= 'a "A")
T ;; since (SYMBOL-NAME 'a) is "A"
(string-equal '|a| 'a)
T ;; since the the symbol names are "a" and "A" respectively
```

Comme note finale, `EQL` sur les caractères est équivalent à `CHAR=` ; `EQUAL` sur les chaînes est équivalent à `STRING=` , tandis que `EQUALP` sur les chaînes est équivalent à `STRING-EQUAL` .

Survol

Dans Common Lisp, il existe de nombreux prédicats différents pour comparer les valeurs. Ils peuvent être classés dans les catégories suivantes:

1. Opérateurs d'égalité générique: EQ, EQL, EQUAL, EQUALP. Ils peuvent être utilisés pour des valeurs de tout type et renvoyer toujours une valeur booléenne T ou NIL.
2. Opérateurs d'égalité de type spécifique: = et = pour les nombres, CHAR = CHAR = CHAR-EQUAL CHAR-NOT-EQUAL pour les caractères, STRING = STRING = STRING-EQUAL

STRING-NOT-EQUAL pour les chaînes, TREE-EQUAL pour les conses.

3. Opérateurs de comparaison pour les valeurs numériques: <, <=,>,> =. Ils peuvent être appliqués à tout type de nombre et comparer la valeur mathématique du nombre, indépendamment du type réel.
4. Opérateurs de comparaison pour les caractères, comme CHAR <, CHAR-LESSP, etc., qui comparent les caractères de manière sensible à la casse ou insensible à la casse, selon un ordre d'implémentation respectant l'ordre alphabétique naturel.
5. Les opérateurs de comparaison pour les chaînes, comme STRING <, STRING-LESSP, etc., comparent les chaînes lexicographiquement, soit de manière sensible à la casse, soit insensible à la casse, en utilisant les opérateurs de comparaison de caractères.

Lire Egalité et autres prédicats de comparaison en ligne: https://riptutorial.com/fr/common-lisp/topic/10064/egalite-et-autres-predicats-de-comparaison

Chapitre 11: Expressions régulières

Examples

Utilisation avec correspondance de motif pour lier des groupes capturés

Le trivia de bibliothèque de correspondance de modèle fournit un `trivia.ppcre` système qui permet aux groupes capturés d'être liés à travers la correspondance de modèle

```
(trivia:match "John Doe"
  ((trivia.ppcre:ppcre "(.*)\\W+(.*)" first-name last-name)
   (list :first-name first-name :last-name last-name)))

;; => (:FIRST-NAME "John" :LAST-NAME "Doe")
```

- Remarque: la bibliothèque Optima fournit une fonctionnalité similaire dans le système `optima.ppcre`

Groupes de registres de liaison avec CL-PPCRE

`CL-PPCRE:REGISTER-GROUPS-BIND` correspondre une chaîne à une expression régulière et, si elle correspond, lie les groupes de registres de l'expression rationnelle aux variables. Si la chaîne ne correspond pas, `NIL` est renvoyé.

```
(defun parse-date-string (date-string)
  (cl-ppcre:register-groups-bind
      (year month day)
      ("(\\d{4})-(\\d{2})-(\\d{2})" date-string)
    (list year month day)))

(parse-date-string "2016-07-23") ;=> ("2016" "07" "23")
(parse-date-string "foobar")     ;=> NIL
(parse-date-string "2016-7-23")  ;=> NIL
```

Lire Expressions régulières en ligne: https://riptutorial.com/fr/common-lisp/topic/2897/expressions-regulieres

Chapitre 12: Fonctionne comme des valeurs de première classe

Syntaxe

- (nom de la fonction); récupère l'objet fonction de ce nom
- #'prénom ; sucre syntaxique pour (nom de la fonction)
- (symbole de fonction de symbole); renvoie la fonction liée au symbole
- (fonction funcall args ...); fonction d'appel avec args
- (appliquer la fonction arglist); fonction d'appel avec des arguments donnés dans une liste
- (appliquer la fonction arg1 arg2 ... argn arglist); fonction d'appel avec des arguments donnés par arg1, arg2, ..., argn, et le reste dans la liste arglist

Paramètres

Paramètre	Détails
prénom	un symbole (non évalué) qui nomme une fonction
symbole	un symbole
fonction	une fonction qui doit être appelée
args ...	zéro ou plusieurs arguments (*pas* une liste d'arguments)
argliste	une liste contenant des arguments à transmettre à une fonction
arg1, arg2, ..., argn	chacun est un seul argument à transmettre à une fonction

Remarques

Lorsque l'on parle de langages de type Lisp, il existe une distinction commune entre ce que l'on appelle un Lisp-1 et un Lisp-2. Dans un Lisp-1, les symboles ont seulement une valeur et si un symbole fait référence à une fonction, alors la valeur de ce symbole sera cette fonction. Dans un Lisp-2, les symboles peuvent avoir des valeurs et des fonctions associées distinctes. Par conséquent, un formulaire spécial est nécessaire pour faire référence à la fonction stockée dans un symbole au lieu de la valeur.

Common Lisp est essentiellement un Lisp-2, mais il y a en fait plus de 2 espaces de noms (des éléments auxquels les symboles peuvent se référer) - les symboles peuvent se référer à des valeurs, des fonctions, des types et des tags, par exemple.

Examples

Définir des fonctions anonymes

Les fonctions en Common Lisp sont *des valeurs de première classe* . Une fonction anonyme peut être créée en utilisant `lambda` . Par exemple, voici une fonction de 3 arguments que nous appelons alors en utilisant `funcall`

```
CL-USER> (lambda (a b c) (+ a (* b c)))
#<FUNCTION (LAMBDA (A B C)) {10034F484B}>
CL-USER> (defvar *foo* (lambda (a b c) (+ a (* b c))))
*FOO*
CL-USER> (funcall *foo* 1 2 3)
7
```

Les fonctions anonymes peuvent également être utilisées directement. Common Lisp fournit une syntaxe pour cela.

```
((lambda (a b c) (+ a (* b c)))    ; the lambda expression as the first
                                   ; element in a form
  1 2 3)                           ; followed by the arguments
```

Les fonctions anonymes peuvent également être stockées en tant que fonctions globales:

```
(let ((a-function (lambda (a b c) (+ a (* b c)))))      ; our anonymous function
  (setf (symbol-function 'some-function) a-function))   ; storing it

(some-function 1 2 3)                                   ; calling it with the name
```

Les expressions lambda cotées ne sont pas des fonctions

Notez que les expressions lambda citées ne sont pas des fonctions dans Common Lisp. Cela ne fonctionne **pas** :

```
(funcall '(lambda (x) x)
         42)
```

Pour convertir une expression lambda citée en une fonction, utilisez `coerce` , `eval` ou `funcall` :

```
CL-USER > (coerce '(lambda (x) x) 'function)
#<anonymous interpreted function 4060000A7C>

CL-USER > (eval '(lambda (x) x))
#<anonymous interpreted function 4060000B9C>

CL-USER > (compile nil '(lambda (x) x))
#<Function 17 4060000CCC>
```

Se référant aux fonctions existantes

Tout symbole dans Common Lisp possède un emplacement pour une variable à lier et un emplacement distinct pour une fonction à lier.

Notez que la dénomination dans cet exemple est uniquement illustrative. Les variables globales ne doivent pas être nommées `foo` , mais `*foo*` . La dernière notation est une convention pour indiquer clairement que la variable est une variable *spéciale* utilisant *la liaison dynamique* .

```
CL-USER> (boundp 'foo) ;is FOO defined as a variable?
NIL
CL-USER> (defvar foo 7)
FOO
CL-USER> (boundp 'foo)
T
CL-USER> foo
7
CL-USER> (symbol-value 'foo)
7
CL-USER> (fboundp 'foo) ;is FOO defined as a function?
NIL
CL-USER> (defun foo (x y) (+ (* x x) (* y y)))
FOO
CL-USER> (fboundp 'foo)
T
CL-USER> foo
7
CL-USER> (symbol-function 'foo)
#<FUNCTION FOO>
CL-USER> (function foo)
#<FUNCTION FOO>
CL-USER> (equalp (quote #'foo) (quote (function foo)))
T
CL-USER> (eq (symbol-function 'foo) #'foo)
T
CL-USER> (foo 4 3)
25
CL-USER> (funcall foo 4 3)
;get an error: 7 is not a function
CL-USER> (funcall #'foo 4 3)
25
CL-USER> (defvar bar #'foo)
BAR
CL-USER> bar
#<FUNCTION FOO>
CL-USER> (funcall bar 4 3)
25
CL-USER> #'+
#<FUNCTION +>
CL-USER> (funcall #'+ 2 3)
5
```

Fonctions d'ordre supérieur

Common Lisp contient de nombreuses fonctions d'ordre supérieur auxquelles sont passées des fonctions pour les arguments et les appelle. Peut-être les plus fondamentaux sont `funcall` et `apply` :

```
CL-USER> (list 1 2 3)
(1 2 3)
CL-USER> (funcall #'list 1 2 3)
(1 2 3)
```

```
CL-USER> (funcall #'list 1 2 3 4 5)
(1 2 3 4 5)
CL-USER> (apply #'list '(1 2 3))
(1 2 3)
CL-USER> (apply #'list 1 2 '(4 5))
(1 2 3 4 5)
CL-USER> (apply #'+ 1 (list 2 3))
6
CL-USER> (defun my-funcall (function &rest args)
           (apply function args))
MY-FUNCALL
CL-USER> (my-funcall #'list 1 2 3)
(1 2 3)
```

Il y a beaucoup d'autres fonctions d'ordre supérieur qui, par exemple, appliquent une fonction plusieurs fois aux éléments d'une liste.

```
CL-USER> (map 'list #'/ '(1 2 3 4))
(1 1/2 1/3 1/4)
CL-USER> (map 'vector #'+ '(1 2 3 4 5) #(5 4 3 2 10))
#(6 6 6 6 15)
CL-USER> (reduce #'+ '(1 2 3 4 5))
15
CL-USER> (remove-if #'evenp '(1 2 3 4 5))
(1 3 5)
```

Résumer une liste

La fonction de **réduction** peut être utilisée pour additionner les éléments d’une liste.

```
(reduce '+ '(1 2 3 4))
;;=> 10
```

Par défaut, **réduire** effectue une réduction *associative à gauche* , ce qui signifie que la somme 10 est calculée comme suit:

```
(+ (+ (+ 1 2) 3) 4)
```

Les deux premiers éléments sont additionnés en premier, puis ce résultat (3) est ajouté à l'élément suivant (3) pour produire 6, qui à son tour s'ajoute à 4, pour produire le résultat final.

Cela est plus sûr que l' utilisation **appliquer** (par exemple, dans **(appliquer « + » (1 2 3 4))** parce que la longueur de la liste des arguments qui peuvent être transmis à **appliquer** est limitée (voir **appel-arguments limite),** et **réduire** fonctionnera avec des fonctions qui ne prennent que deux arguments.

En spécifiant l'argument mot **-** clé **from-end** , **réduire** traite la liste dans l'autre sens, ce qui signifie que la somme est calculée dans l'ordre inverse. C'est

```
(reduce '+ (1 2 3 4) :from-end t)
;;=> 10
```

est l'informatique

```
(+ 1 (+ 2 (+ 3 4)))
```

Implémenter l'inverse et le revappend

Common Lisp a déjà une fonction **inverse** , mais si ce n'était pas le cas, il pourrait être implémenté facilement en utilisant **Reduce** . Étant donné une liste comme

```
(1 2 3) === (cons 1 (cons 2 (cons 3 '())))
```

la liste inversée est

```
(cons 3 (cons 2 (cons 1 '()))) === (3 2 1)
```

Cela peut ne pas être une utilisation évidente de **réduire** , mais si nous avons une fonction "renversée", disons **xcons** , de telle sorte que

```
(xcons 1 2) === (2 . 1)
```

alors

```
(xcons (xcons (xcons () 1) 2) 3)
```

ce qui est une réduction.

```
(reduce (lambda (x y)
          (cons y x))
        '(1 2 3 4)
        :initial-value '())
;=> (4 3 2 1)
```

Common Lisp a une autre fonction utile, **revappend** , qui est une combinaison de **reverse** et **append** . Conceptuellement, il inverse une liste et l'ajoute à une queue:

```
(revappend '(3 2 1) '(4 5 6))
;;=> (1 2 3 4 5 6)
```

Cela peut également être implémenté avec **réduire** . En fait, c'est la même chose que l'implémentation d' **inverse** ci-dessus, sauf que la valeur initiale devrait être **(4 5 6)** au lieu de la liste vide.

```
(reduce (lambda (x y)
          (cons y x))
        '(3 2 1)
        :initial-value '(4 5 6))
;=> (1 2 3 4 5 6)
```

Fermetures

Les fonctions se souviennent de la portée lexicale dans laquelle elles ont été définies. À cause de cela, nous pouvons inclure un lambda dans un let pour définir des fermetures.

```
(defvar *counter* (let ((count 0))
                    (lambda () (incf count))))

(funcall *counter*) ;; => 1
(funcall *counter*) ;; = 2
```

Dans l'exemple ci-dessus, la variable compteur n'est accessible qu'à la fonction anonyme. Ceci est plus clairement vu dans l'exemple suivant

```
(defvar *counter-1* (make-counter))
(defvar *counter-2* (make-counter))

(funcall *counter-1*) ;; => 1
(funcall *counter-1*) ;; => 2
(funcall *counter-2*) ;; => 1
(funcall *counter-1*) ;; => 3
```

Définition de fonctions prenant en charge des fonctions et des fonctions de retour

Un exemple simple:

```
CL-USER> (defun make-apply-twice (fun)
           "return a new function that applies twice the function`fun' to its argument"
           (lambda (x)
             (funcall fun (funcall fun x))))
MAKE-APPLY-TWICE
CL-USER> (funcall (make-apply-twice #'1+) 3)
5
CL-USER> (let ((pow4 (make-apply-twice (lambda (x) (* x x)))))
           (funcall pow4 3))
81
```

L'exemple classique de composition fonctionnelle : (*f* ∘ *g* ∘ *h*) (*x*) = *f* (*g* (*h* (*x*)):

```
CL-USER> (defun compose (&rest funs)
           "return a new function obtained by the functional compositions of the parameters"
           (if (null funs)
               #'identity
               (let ((rest-funs (apply #'compose (rest funs))))
                 (lambda (x) (funcall (first funs) (funcall rest-funs x))))))
COMPOSE
CL-USER> (defun square (x) (* x x))
SQUARE
CL-USER> (funcall (compose #'square #'1+ #'square) 3)
100  ;; => equivalent to (square (1+ (square 3)))
```

Lire Fonctionne comme des valeurs de première classe en ligne: https://riptutorial.com/fr/common-

lisp/topic/1259/fonctionne-comme-des-valeurs-de-premiere-classe

Chapitre 13: format

Paramètres

Liste Lambda	`(format DESTINATION CONTROL-STRING &REST FORMAT-ARGUMENTS)`
`DESTINATION`	la chose à écrire. Cela peut être un flux de sortie, `t` (raccourci pour `*standard-output*`) ou `nil` (qui crée une chaîne à écrire)
`CONTROL-STRING`	la chaîne de template. Il peut s'agir d'une chaîne primitive ou de directives de commande avec préfixe tilde spécifiant et transformant d'une manière ou d'une autre des arguments supplémentaires.
`FORMAT-ARGUMENTS`	arguments supplémentaires potentiels requis par le `CONTROL-STRING` .

Remarques

La documentation CLHS pour les directives `FORMAT` se trouve à la Section 22.3 . Avec SLIME, vous pouvez taper `Cc Cd ~` pour rechercher la documentation CLHS d'une directive de format spécifique.

Examples

Utilisation de base et directives simples

Les deux premiers arguments à formater sont un flux de sortie et une chaîne de contrôle. L'utilisation de base ne nécessite pas d'arguments supplémentaires. Passer `t` comme le flux écrit dans `*standard-output*` .

```
> (format t "Basic Message")
Basic Message
nil
```

Cette expression va écrire `Basic Message` sur la sortie standard et renvoyer la valeur `nil` .

`nil` le flux est `nil` , le flux crée une nouvelle chaîne et la renvoie.

```
> (format nil "Basic Message")
"Basic Message"
```

La plupart des directives de chaîne de contrôle nécessitent des arguments supplémentaires. La `~a` directive ("esthétique") imprimera tout argument comme par la procédure `princ` . Ceci imprime le formulaire sans aucun caractère d'échappement (les mots-clés sont imprimés sans les deux-

points principaux, les chaînes sans les guillemets qui les entourent, etc.).

```
> (format nil "A Test: ~a" 42)
"A Test: 42"
> (format nil "Multiples: ~a ~a ~a ~a" 1 (list 2 3) "four five" :six)
"Multiples: 1 (2 3) four five SIX"
> (format nil "A Test: ~a" :test)
"A Test: TEST"
> (format nil "A Test: ~a" "Example")
"A Test: Example"
```

`~a` entrée optionnelle droite ou gauche basée sur des entrées supplémentaires.

```
> (format nil "A Test: ~10a" "Example")
"A Test: Example   "
> (format nil "A Test: ~10@a" "Example")
"A Test:    Example"
```

La directive `~s` est comme `~a` , mais elle imprime des caractères d'échappement.

```
> (format nil "A Test: ~s" 42)
"A Test: 42"
> (format nil "Multiples: ~s ~s ~s ~s" 1 (list 2 3) "four five" :six)
"Multiples: 1 (2 3) \"four five\" :SIX"
> (format nil "A Test: ~s" :test)
"A Test: :TEST"
> (format nil "A Test: ~s" "Example")
"A Test: \"Example\""
```

Itérer sur une liste

On peut parcourir une liste en utilisant les directives `~{` et `~}` .

```
CL-USER> (format t "~{~a, ~}~%" '(1 2 3 4 5))
1, 2, 3, 4, 5,
```

`~^` peut être utilisé pour échapper s'il ne reste plus d'éléments.

```
CL-USER> (format t "~{~a~^, ~}~%" '(1 2 3 4 5))
1, 2, 3, 4, 5
```

Un argument numérique peut être donné à `~{` pour limiter le nombre d'itérations pouvant être effectuées:

```
CL-USER> (format t "~3{~a~^, ~}~%" '(1 2 3 4 5))
1, 2, 3,
```

`~@{` va parcourir les arguments restants au lieu d'une liste:

```
CL-USER> (format t "~a: ~@{~a~^, ~}~%" :foo 1 2 3 4 5)
FOO: 1, 2, 3, 4, 5
```

Les sous-listes peuvent être répétées en utilisant `~:{` :

```
CL-USER> (format t "~:{(~a, ~a) ~}~%" '((1 2) (3 4) (5 6)))
(1, 2) (3, 4) (5, 6)
```

Expressions conditionnelles

Les expressions conditionnelles peuvent être effectuées avec `~[` et `~]` . Les clauses de l'expression sont séparées en utilisant `~;` .

Par défaut, `~[` prend un entier dans la liste des arguments et sélectionne la clause correspondante. Les clauses commencent à zéro.

```
(format t "~@{~[First clause~;Second clause~;Third clause~;Fourth clause~]~%~}"
        0 1 2 3)
; First clause
; Second clause
; Third clause
; Fourth clause
```

La dernière clause peut être séparée avec `~:;` au lieu de faire la clause sinon.

```
(format t "~@{~[First clause~;Second clause~;Third clause~:;Too high!~]~%~}"
        0 1 2 3 4 5)
; First clause
; Second clause
; Third clause
; Too high!
; Too high!
; Too high!
```

Si l'expression conditionnelle commence par `~:[` , elle attendra un booléen généralisé au lieu d'un entier. Il ne peut avoir que deux clauses; le premier est imprimé si le booléen est `NIL` et le second si c'est vrai.

```
(format t "~@{~:[False!~;True!~]~%~}"
        t nil 10 "Foo" '())
; True!
; False!
; True!
; True!
; False!
```

Si l'expression conditionnelle commence par `~@[` , il ne devrait y avoir qu'une seule clause, qui est imprimée si l'entrée, un booléen généralisé, était véridique. Le booléen ne sera pas consommé s'il est véridique.

```
(format t "~@{~@[~s is truthy!~%~]~}"
        t nil 10 "Foo" '())
; T is truthy!
; 10 is truthy!
; "Foo" is truthy!
```

Lire format en ligne: https://riptutorial.com/fr/common-lisp/topic/687/format

Chapitre 14: Les booléens et les booléens généralisés

Examples

Vrai et faux

Le symbole spécial `T` représente la valeur *true* dans Common Lisp, tandis que le symbole spécial `NIL` représente *false* :

```
CL-USER> (= 3 3)
T
CL-USER> (= 3 4)
NIL
```

Ils sont appelés «variables constantes» (sic!) Dans la norme, car ce sont des variables dont la valeur *ne peut pas* être modifiée. Par conséquent, vous ne pouvez pas utiliser leurs noms pour des variables normales, comme dans l'exemple suivant, incorrect, par exemple:

```
CL-USER> (defun my-fun(t)
           (+ t 1))
While compiling MY-FUN :
Can't bind or assign to constant T.
```

En fait, on peut les considérer simplement comme des constantes ou comme des symboles auto-évalués. `T` et `NIL` sont aussi des spécialités dans d'autres sens. Par exemple, `T` est également un type (le sur-type de tout autre type), tandis que `NIL` est également la liste vide:

```
CL-USER> (eql NIL '())
T
CL-USER> (cons 'a (cons 'b nil))
(A B)
```

Booléens Généralisés

En fait, toute valeur différente de `NIL` est considérée comme une *vraie* valeur dans Common Lisp. Par exemple:

```
CL-USER> (let ((a (+ 2 2)))
           (if a
               a
               "Oh my! 2 + 2 is equal to NIL!"))
4
```

Ce fait peut être combiné avec les opérateurs booléens pour rendre les programmes plus concis. Par exemple, l'exemple ci-dessus équivaut à:

```
CL-USER> (or (+ 2 2) "Oh my! 2 + 2 is equal to NIL!")
4
```

La macro `OR` évalue ses arguments dans l'ordre de gauche à droite et s'arrête dès qu'elle trouve une valeur non NIL, en la renvoyant. Si tous sont `NIL` , la valeur renvoyée est `NIL` :

```
CL-USER> (or (= 1 2) (= 3 4) (= 5 6))
NIL
```

De manière analogue, la macro `AND` évalue ses arguments de gauche à droite et retourne la valeur du dernier, si tous sont évalués en non-NIL, sinon arrête l'évaluation dès qu'elle trouve `NIL` , en la renvoyant:

```
CL-USER> (let ((a 2)
               (b 3))
           (and (/= b 0) (/ a b)))
2/3
CL-USER> (let ((a 2)
               (b 0))
           (and (/= b 0) (/ a b)))
NIL
```

Pour ces raisons, `AND` et `OR` peuvent être considérés comme plus similaires aux structures de contrôle d'autres langages, plutôt qu'aux opérateurs booléens.

Lire Les booléens et les booléens généralisés en ligne: https://riptutorial.com/fr/common-lisp/topic/3292/les-booleens-et-les-booleens-generalises

Chapitre 15: Les fonctions

Remarques

Les fonctions anonymes peuvent être créées avec `LAMBDA` . Les fonctions locales peuvent être définies en utilisant `LABELS` ou `FLET` . Leurs paramètres sont définis de la même manière que dans les fonctions nommées globales.

Examples

Paramètres requis

```
(defun foobar (x y)
  (format t "X: ~s~@
             Y: ~s~%"
          x y))

(foobar 10 20)
; X: 10
; Y: 20
;=> NIL
```

Paramètres facultatifs

Les paramètres facultatifs peuvent être spécifiés après les paramètres requis, en utilisant le mot clé `&OPTIONAL` . Il peut y avoir plusieurs paramètres facultatifs après cela.

```
(defun foobar (x y &optional z)
  (format t "X (~s) and Y (~s) are required.~@
             Z (~s) is optional.~%"
          x y z))

(foobar 10 20)
; X (10) and Y (20) are required.
; Z (NIL) is optional.
;=> NIL
(foobar 10 20 30)
; X (10) and Y (20) are required.
; Z (30) is optional.
;=> NIL
```

Alue par défaut

Une valeur par défaut peut être donnée pour les paramètres facultatifs en spécifiant le paramètre avec une liste; la deuxième valeur est la valeur par défaut. La forme de valeur par défaut ne sera évaluée que si l'argument a été donné, de sorte qu'il peut être utilisé pour les effets secondaires, tels que signaler une erreur.

```
(defun foobar (x y &optional (z "Default"))
  (format t "X (~s) and Y (~s) are required.~@
             Z (~s) is optional.~%"
          x y z))

(foobar 10 20)
; X (10) and Y (20) are required.
; Z ("Default") is optional.
;=> NIL
(foobar 10 20 30)
; X (10) and Y (20) are required.
; Z (30) is optional.
;=> NIL
```

Vérifiez si un argument optionnel a été donné

Un troisième membre peut être ajouté à la liste après la valeur par défaut; un nom de variable qui est vrai si l'argument a été donné ou `NIL` s'il n'a pas été donné (et la valeur par défaut est utilisée).

```
(defun foobar (x y &optional (z "Default" zp))
  (format t "X (~s) and Y (~s) are required.~@
             Z (~s) is optional. It ~:[wasn't~;was~] given.~%"
          x y z zp))

(foobar 10 20)
; X (10) and Y (20) are required.
; Z ("Default") is optional. It wasn't given.
;=> NIL
(foobar 10 20 30)
; X (10) and Y (20) are required.
; Z (30) is optional. It was given.
;=> NIL
```

Fonction sans paramètres

Les fonctions nommées globales sont définies avec `DEFUN` .

```
(defun foobar ()
  "Optional documentation string. Can contain line breaks.

Must be at the beginning of the function body. Some will format the
docstring so that lines are indented to match the first line, although
the built-in DESCRIBE-function will print it badly indented that way.

Ensure no line starts with an opening parenthesis by escaping them
\(like this), otherwise your editor may have problems identifying
toplevel forms."
  (format t "No parameters.~%"))

(foobar)
; No parameters.
;=> NIL

(describe #'foobar) ; The output is implementation dependant.
; #<FUNCTION FOOBAR>
```

```
;   [compiled function]
;
; Lambda-list: ()
; Derived type: (FUNCTION NIL (VALUES NULL &OPTIONAL))
; Documentation:
;   Optional documentation string. Can contain line breaks.
;
;   Must be at the beginning of the function body. Some will format the
;   docstring so that lines are indented to match the first line, although
;   the built-in DESCRIBE-function will print it badly indented that way.
; Source file: /tmp/fileInaZ1P
;=> No values
```

Le corps de la fonction peut contenir un nombre quelconque de formulaires. Les valeurs du dernier formulaire seront renvoyées par la fonction.

Paramètre de repos

Un seul paramètre de repos peut être donné avec le mot-clé `&REST` après les arguments requis. Si un tel paramètre existe, la fonction peut prendre plusieurs arguments, qui seront regroupés dans une liste du paramètre rest. Notez que la variable `CALL-ARGUMENTS-LIMIT` détermine le nombre maximum d'arguments pouvant être utilisés dans un appel de fonction. Le nombre d'arguments est donc limité à une valeur spécifique d'implémentation d'au moins 50 arguments ou plus.

```
(defun foobar (x y &rest rest)
  (format t "X (~s) and Y (~s) are required.~@
             The function was also given following arguments: ~s~%"
          x y rest))

(foobar 10 20)
; X (10) and Y (20) are required.
; The function was also given following arguments: NIL
;=> NIL
(foobar 10 20 30 40 50 60 70 80)
; X (10) and Y (20) are required.
; The function was also given following arguments: (30 40 50 60 70 80)
;=> NIL
```

Paramètres de repos et de mots-clés ensemble

Le paramètre rest peut être avant les paramètres mot clé. Dans ce cas, il contiendra la liste de propriétés donnée par l'utilisateur. Les valeurs de mot-clé seront toujours liées au paramètre de mot-clé correspondant.

```
(defun foobar (x y &rest rest &key (z 10 zp))
  (format t "X (~s) and Y (~s) are required.~@
             Z (~s) is a keyword argument. It ~:[wasn't~;was~] given.~@
             The function was also given following arguments: ~s~%"
          x y z zp rest))
```

```
(foobar 10 20)
; X (10) and Y (20) are required.
; Z (10) is a keyword argument. It wasn't given.
; The function was also given following arguments: NIL
;=> NIL
(foobar 10 20 :z 30)
; X (10) and Y (20) are required.
; Z (30) is a keyword argument. It was given.
; The function was also given following arguments: (:Z 30)
;=> NIL
```

Le mot `&ALLOW-OTHER-KEYS` clé `&ALLOW-OTHER-KEYS` peut être ajouté à la fin de la liste lambda pour permettre à l'utilisateur de donner des arguments de mot-clé non définis en tant que paramètres. Ils iront dans la liste de repos.

```
(defun foobar (x y &rest rest &key (z 10 zp) &allow-other-keys)
  (format t "X (~s) and Y (~s) are required.~@
             Z (~s) is a keyword argument. It ~:[wasn't~;was~] given.~@
             The function was also given following arguments: ~s~%"
          x y z zp rest))

(foobar 10 20 :z 30 :q 40)
; X (10) and Y (20) are required.
; Z (30) is a keyword argument. It was given.
; The function was also given following arguments: (:Z 30 :Q 40)
;=> NIL
```

Variables Auxiliaires

Le mot clé `&AUX` peut être utilisé pour définir des variables locales pour la fonction. Ce ne sont pas des paramètres. l'utilisateur ne peut pas les fournir.

`&AUX` variables `&AUX` sont rarement utilisées. Vous pouvez toujours utiliser `LET` ou une autre manière de définir des variables locales dans le corps de la fonction.

`&AUX` variables `&AUX` ont l'avantage que les variables locales du corps entier de la fonction sont déplacées vers le haut et qu'un niveau d'indentation (par exemple, introduit par un LET) est inutile.

```
(defun foobar (x y &aux (z (+ x y)))
  (format t "X (~d) and Y (~d) are required.~@
             Their sum is ~d."
          x y z))

(foobar 10 20)
; X (10) and Y (20) are required.
; Their sum is 30.
;=> NIL
```

Une utilisation typique peut être la résolution des paramètres "désignateur". Encore une fois, vous n'avez pas besoin de le faire de cette façon; Utiliser `let` est tout aussi idiomatique.

```
(defun foo (a b &aux (as (string a)))
  "Combines A and B in a funny way.  A is a string designator, B a string."
  (concatenate 'string as " is funnier than " b))
```

RETURN-FROM, sortie d'un bloc ou d'une fonction

Les fonctions établissent toujours un bloc autour du corps. Ce bloc porte le même nom que le nom de la fonction. Cela signifie que vous pouvez utiliser `RETURN-FROM` avec ce nom de bloc pour revenir de la fonction et renvoyer des valeurs.

Vous devriez éviter de revenir tôt si possible.

```
(defun foobar (x y)
  (when (oddp x)
    (format t "X (~d) is odd. Returning immediately.~%" x)
    (return-from foobar "return value"))
  (format t "X: ~s~@
             Y: ~s~%"
          x y))

(foobar 10 20)
; X: 10
; Y: 20
;=> NIL
(foobar 9 20)
; X (9) is odd. Returning immediately.
;=> "return value"
```

Paramètres de mot clé

Les paramètres de mot-clé peuvent être définis avec le mot-clé `&KEY` . Ils sont toujours facultatifs (voir l'exemple des paramètres facultatifs pour plus de détails sur la définition). Il peut y avoir plusieurs paramètres de mot-clé.

```
(defun foobar (x y &key (z "Default" zp))
  (format t "X (~s) and Y (~s) are required.~@
             Z (~s) is a keyword argument. It ~:[wasn't~;was~] given.~%"
          x y z zp))

(foobar 10 20)
; X (10) and Y (20) are required.
; Z ("Default") is a keyword argument. It wasn't given.
;=> NIL
(foobar 10 20 :z 30)
; X (10) and Y (20) are required.
; Z (30) is a keyword argument. It was given.
;=> NIL
```

Lire Les fonctions en ligne: https://riptutorial.com/fr/common-lisp/topic/2126/les-fonctions

Chapitre 16: les macros

Remarques

Le but des macros

Les macros sont destinées à générer du code, à transformer du code et à fournir de nouvelles notations. Ces nouvelles notations peuvent être plus adaptées pour mieux exprimer le programme, par exemple en fournissant des constructions au niveau du domaine ou de nouveaux langages intégrés.

Les macros peuvent rendre le code source plus explicite, mais le débogage peut être rendu plus difficile. En règle générale, il ne faut pas utiliser de macros quand une fonction normale fera l'affaire. Lorsque vous les utilisez, évitez les pièges habituels, essayez de vous en tenir aux schémas et conventions de dénomination couramment utilisés.

Commande Macroexpansion

Par rapport aux fonctions, les macros sont développées dans l'ordre inverse. le plus en premier, le dernier en dernier. Cela signifie que, par défaut, il est impossible d'utiliser une macro interne pour générer la syntaxe requise pour une macro externe.

Ordre d'évaluation

Parfois, les macros doivent déplacer les formulaires fournis par l'utilisateur. Il faut s'assurer de ne pas changer l'ordre dans lequel ils sont évalués. L'utilisateur peut se fier aux effets secondaires se produisant dans l'ordre.

Évaluer une seule fois

L'expansion d'une macro doit souvent utiliser plusieurs fois la même valeur fournie par l'utilisateur. Il est possible que le formulaire ait des effets secondaires ou qu'il appelle une fonction coûteuse. Ainsi, la macro doit s'assurer de n'évaluer que de telles formes une seule fois. Cela se fera généralement en assignant la valeur à une variable locale (dont le nom est `GENSYM` ed).

Fonctions utilisées par les macros, en utilisant EVAL-WHEN

Les macros complexes comportent souvent des parties de leur logique implémentées dans des fonctions distinctes. Il ne faut cependant pas oublier que les macros sont développées avant la compilation du code. Lors de la compilation d'un fichier, les fonctions et les variables définies dans le même fichier ne seront pas disponibles par défaut lors de l'exécution de la macro. Toutes les définitions de fonctions et de variables, dans le même fichier, utilisées par une macro doivent être placées dans un `EVAL-WHEN` . `EVAL-WHEN` doit avoir toutes les trois fois spécifié, lorsque le code joint doit également être évalué pendant le chargement et le runtime.

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (defun foobar () ...))
```

Cela ne s'applique pas aux fonctions appelées à partir de l'expansion de la macro, seulement celles appelées par la macro elle-même.

Examples

Modèles Macro communs

TODO: Peut-être déplacer les explications à des remarques et ajouter des exemples séparément

FOOF

Dans Common Lisp, il existe un concept de références généralisées . Ils permettent à un programmeur de définir des valeurs à différents "endroits" comme s'ils étaient des variables. Les macros qui utilisent cette capacité ont souvent un préfixe `F` dans le nom. Le lieu est généralement le premier argument de la macro.

Exemples de la norme: `INCF` , `DECF` , `ROTATEF` , `SHIFTF` , `REMF` .

Un exemple idiot, une macro qui retourne le signe d'un magasin de nombres dans un lieu:

```
(defmacro flipf (place)
  `(setf ,place (- ,place)))
```

AVEC-FOO

Les macros qui acquièrent et libèrent en toute sécurité une ressource sont généralement nommées avec un `WITH-` WITH-. La macro doit généralement utiliser une syntaxe telle que:

```
(with-foo (variable details-of-the-foo...)
  body...)
```

Exemples tirés du standard: `WITH-OPEN-FILE` , `WITH-OPEN-STREAM` , `WITH-INPUT-FROM-STRING` , `WITH-OUTPUT-TO-STRING` .

Une approche pour implémenter ce type de macro qui peut éviter certains des pièges de la pollution de nom et de l'évaluation multiple involontaire consiste à implémenter une version fonctionnelle en premier. Par exemple, la première étape de l'implémentation d'une macro `with-widget` qui crée en toute sécurité un widget et le nettoie peut être une fonction:

```
(defun call-with-widget (args function)
  (let ((widget (apply #'make-widget args))) ; obtain WIDGET
    (unwind-protect (funcall function widget)  ; call FUNCTION with WIDGET
      (cleanup widget)                         ; cleanup
```

Comme il s'agit d'une fonction, l'étendue des noms au sein de la **fonction** ou du **fournisseur** ne pose aucun problème et facilite l'écriture d'une macro correspondante:

```
(defmacro with-widget ((var &rest args) &body body)
  `(call-with-widget (list ,@args) (lambda (,var) ,@body)))
```

DO-FOO

Les macros qui parcourent quelque chose sont souvent nommées avec un préfixe `DO` . La macro-syntaxe devrait généralement être sous la forme

```
(do-foo (variable the-foo-being-done return-value)
  body...)
```

Exemples de la norme: `DOTIMES` , `DOLIST` , `DO-SYMBOLS` .

FOOCASE, EFOOCASE, CFOOCASE

Les macros qui correspondent à une entrée par rapport à certains cas sont souvent nommées avec un correctif de type `CASE` . Il y a souvent une variable `E...CASE` , qui signale une erreur si l'entrée ne correspond à aucun des cas, et `C...CASE` , qui signale une erreur continue. Ils devraient avoir une syntaxe comme

```
(foocase input
  (case-to-match-against (optionally-some-params-for-the-case)
   case-body-forms...)
  more-cases...
  [(otherwise otherwise-body)])
```

Exemples tirés de la norme: `CASE` , `TYPECASE` , `HANDLER-CASE` .

Par exemple, une macro qui associe une chaîne à des expressions régulières et lie les groupes de registres à des variables. Utilise CL-PPCRE pour les expressions régulières.

```
(defmacro regexcase (input &body cases)
  (let ((block-sym (gensym "block"))
        (input-sym (gensym "input")))
```

```
    `(let ((,input-sym ,input))
       (block ,block-sym
         ,@(loop for (regex vars . body) in cases
                 if (eql regex 'otherwise)
                   collect `(return-from ,block-sym (progn ,vars ,@body))
                 else
                   collect `(cl-ppcre:register-groups-bind ,vars
                                (,regex ,input-sym)
                              (return-from ,block-sym
                                (progn ,@body)))))))))

(defun test (input)
  (regexcase input
    ("(\\d+)-(\\d+)" (foo bar)
      (format t "Foo: ~a, Bar: ~a~%" foo bar))
    ("Foo: (\\w+)$" (foo)
      (format t "Foo: ~a.~%" foo))
    (otherwise (format t "Didn't match.~%"))))

(test "asd 23-234 qwe")
; Foo: 23, Bar: 234
(test "Foo: Foobar")
; Foo: Foobar.
(test "Foo: 43 - 23")
; Didn't match.
```

DEFINE-FOO, DEFFOO

Les macros qui définissent des choses sont généralement nommées avec un `DEFINE-` ou `DEF` .

Exemples de la norme: `DEFUN` , `DEFMACRO` , `DEFINE-CONDITION` .

Macros anaphoriques

Une macro anaphorique est une macro qui introduit une variable (souvent `IT`) qui capture le résultat d'un formulaire fourni par l'utilisateur. Un exemple courant est le If anaphorique, qui est comme un `IF` régulier, mais définit également la variable `IT` pour faire référence au résultat du formulaire de test.

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))

(defun test (property plist)
  (aif (getf plist property)
       (format t "The value of ~s is ~a.~%" property it)
       (format t "~s wasn't in ~s!~%" property plist)))

(test :a '(:a 10 :b 20 :c 30))
; The value of :A is 10.
(test :d '(:a 10 :b 20 :c 30))
; :D wasn't in (:A 10 :B 20 :C 30)!
```

MACROEXPAND

L'expansion des macros est le processus consistant à transformer les macros en code réel. Cela se produit généralement dans le cadre du processus de compilation. Le compilateur développera toutes les formes de macro avant de compiler le code. L'expansion de la macro se produit également lors de l' *interprétation* du code Lisp.

On peut appeler `MACROEXPAND` manuellement pour voir comment se développe une forme de macro.

```
CL-USER> (macroexpand '(with-open-file (file "foo")
                        (do-something-with file)))
(LET ((FILE (OPEN "foo")) (#:G725 T))
  (UNWIND-PROTECT
      (MULTIPLE-VALUE-PROG1 (PROGN (DO-SOMETHING-WITH FILE)) (SETQ #:G725 NIL))
    (WHEN FILE (CLOSE FILE :ABORT #:G725))))
```

`MACROEXPAND-1` est identique, mais ne s'étend qu'une seule fois. Ceci est utile lorsque vous essayez de donner un sens à une forme de macro qui se développe en une autre forme de macro.

```
CL-USER> (macroexpand-1 '(with-open-file (file "foo")
                          (do-something-with file)))
(WITH-OPEN-STREAM (FILE (OPEN "foo")) (DO-SOMETHING-WITH FILE))
```

Notez que ni `MACROEXPAND` ni `MACROEXPAND-1` développent le code Lisp à tous les niveaux. Ils développent uniquement la forme de macro de niveau supérieur. Pour macroexpand une forme complète à tous les niveaux, il faut un *code walker* pour le faire. Cette fonctionnalité n'est pas fournie dans le standard Common Lisp.

Backquote - écriture de modèles de code pour les macros

Code de retour des macros. Comme le code dans Lisp est constitué de listes, on peut utiliser les fonctions de manipulation de liste régulières pour le générer.

```
;; A pointless macro
(defmacro echo (form)
  (list 'progn
        (list 'format t "Form: ~a~%" (list 'quote form))
        form))
```

C'est souvent très difficile à lire, en particulier dans les macros plus longues. La macro lecteur Backquote permet d'écrire des modèles cités remplis en évaluant sélectivement des éléments.

```
(defmacro echo (form)
  `(progn
     (format t "Form: ~a~%" ',form)
     ,form))

(macroexpand '(echo (+ 3 4)))
;=> (PROGN (FORMAT T "Form: ~a~%" '(+ 3 4)) (+ 3 4))
```

Cette version ressemble presque à un code normal. Les virgules sont utilisées pour évaluer `FORM` ; tout le reste est retourné tel quel. Notez que dans `',form` le guillemet simple est en dehors de la virgule, donc il sera retourné.

On peut aussi utiliser `,@` pour épisser une liste dans la position.

```
(defmacro echo (&rest forms)
  `(progn
     ,@(loop for form in forms collect `(format t "Form: ~a~%" ,form))
     ,@forms))

(macroexpand '(echo (+ 3 4)
                    (print "foo")
                    (random 10)))
;=> (PROGN
;     (FORMAT T "Form: ~a~%" (+ 3 4))
;     (FORMAT T "Form: ~a~%" (PRINT "foo"))
;     (FORMAT T "Form: ~a~%" (RANDOM 10))
;     (+ 3 4)
;     (PRINT "foo")
;     (RANDOM 10))
```

Backquote peut également être utilisé en dehors des macros.

Symboles uniques pour empêcher les conflits de noms dans les macros

L'expansion d'une macro doit souvent utiliser des symboles qui n'ont pas été passés en tant qu'arguments par l'utilisateur (en tant que noms de variables locales, par exemple). Il faut s'assurer que ces symboles ne peuvent pas entrer en conflit avec un symbole que l'utilisateur utilise dans le code environnant.

Ceci est généralement réalisé en utilisant `GENSYM` , une fonction qui retourne un nouveau symbole.

Mal

Considérons la macro ci-dessous. Il crée une `DOTIMES` DOTIMES qui collecte également le résultat du corps dans une liste qui est renvoyée à la fin.

```
(defmacro dotimes+collect ((var count) &body body)
  `(let ((result (list)))
     (dotimes (,var ,count (nreverse result))
       (push (progn ,@body) result))))

(dotimes+collect (i 5)
  (format t "~a~%" i)
  (* i i))
; 0
; 1
; 2
; 3
; 4
;=> (0 1 4 9 16)
```

Cela semble fonctionner dans ce cas, mais si l'utilisateur avait un nom de variable `RESULT` , qu'il utilise dans le corps, les résultats ne seraient probablement pas ceux attendus par l'utilisateur. Considérez cette tentative d'écrire une fonction qui collecte une liste de sommes de tous les entiers jusqu'à `N` :

```
(defun sums-upto (n)
  (let ((result 0))
    (dotimes+collect (i n)
      (incf result i))))

(sums-upto 10) ;=> Error!
```

Bien

Pour résoudre le problème, nous devons utiliser `GENSYM` pour générer un nom unique pour la variable `RESULT` dans la macro.

```
(defmacro dotimes+collect ((var count) &body body)
  (let ((result-symbol (gensym "RESULT")))
    `(let ((,result-symbol (list)))
       (dotimes (,var ,count (nreverse ,result-symbol))
         (push (progn ,@body) ,result-symbol)))))

(sums-upto 10) ;=> (0 1 3 6 10 15 21 28 36 45)
```

TODO: Comment faire des symboles à partir de chaînes

TODO: éviter les problèmes avec les symboles dans différents paquets

si-laisser, quand-laisser, -let macros

Ces macros fusionnent le flux de contrôle et la liaison. Ils constituent une amélioration par rapport aux macros anaphoriques anaphoriques car ils permettent au développeur de communiquer le sens par le biais du nommage. En tant que tel, leur utilisation est recommandée par rapport à leurs homologues anaphoriques.

```
(if-let (user (get-user user-id))
  (show-dashboard user)
  (redirect 'login-page))
```

`FOO-LET` macros `FOO-LET` lient une ou plusieurs variables, puis utilisent ces variables comme formulaire de test pour le conditionnel correspondant (`IF` , `WHEN`). Plusieurs variables sont combinées avec `AND` . La branche choisie est exécutée avec les liaisons en vigueur. Une simple implémentation d'une variable de `IF-LET` pourrait ressembler à ceci:

```
(defmacro if-let ((var test-form) then-form &optional else-form)
  `(let ((,var ,test-form))
     (if ,var ,then-form ,else-form)))

(macroexpand '(if-let (a (getf '(:a 10 :b 20 :c 30) :a))
               (format t "A: ~a~%" a)
               (format t "Not found.~%")))
; (LET ((A (GETF '(:A 10 :B 20 :C 30) :A)))
;   (IF A
;       (FORMAT T "A: ~a~%" A)
;       (FORMAT T "Not found.~%")))
```

Une version prenant en charge plusieurs variables est disponible dans la bibliothèque Alexandria .

Utilisation de macros pour définir des structures de données

Une utilisation courante des macros consiste à créer des modèles pour des structures de données qui obéissent à des règles communes mais peuvent contenir des champs différents. En écrivant une macro, vous pouvez autoriser la configuration détaillée de la structure de données sans avoir à répéter le code standard, ni utiliser une structure moins efficace (comme un hachage) en mémoire uniquement pour simplifier la programmation.

Par exemple, supposons que nous souhaitons définir un certain nombre de classes ayant une gamme de propriétés différentes, chacune avec un getter et un setter. De plus, pour certaines (mais pas toutes) de ces propriétés, nous souhaitons que le compositeur appelle une méthode sur l'objet lui indiquant que la propriété a été modifiée. Bien que Common LISP ait déjà un raccourci pour écrire des getters et des setters, écrire un setter personnalisé standard de cette manière nécessiterait normalement de dupliquer le code qui appelle la méthode de notification dans chaque setter, ce qui pourrait être compliqué . Cependant, en définissant une macro, cela devient beaucoup plus facile:

```
(defmacro notifier (class slot)
  "Defines a setf method in (class) for (slot) which calls the object's changed method."
   `(defmethod (setf ,slot) (val (item ,class))
     (setf (slot-value item ',slot) val)
     (changed item ',slot)))

(defmacro notifiers (class slots)
  "Defines setf methods in (class) for all of (slots) which call the object's changed method."
  `(progn
     ,@(loop for s in slots collecting `(notifier ,class ,s))))

(defmacro defclass-notifier-slots (class nslots slots)
  "Defines a class with (nslots) giving a list of slots created with notifiers, and (slots)
giving a list of slots created with regular accessors."
  `(progn
     (defclass ,class ()
       ( ,@(loop for s in nslots collecting `(,s :reader ,s))
         ,@(loop for s in slots collecting `(,s :accessor ,s))))
     (notifiers ,class ,nslots)))
```

Nous pouvons maintenant écrire `(defclass-notifier-slots foo (bar baz qux) (waldo))` et définir immédiatement une classe `foo` avec un slot `waldo` (créé par la seconde partie de la macro avec la spécification `(waldo :accessor waldo)`) , et slots `bar` , `baz` et `qux` avec les setters qui appellent la méthode `changed` (où le getter est défini par la première partie de la macro, `(bar :reader bar)` et le setter par la macro `notifier` invoquée).

En plus de nous permettre de définir rapidement plusieurs classes qui se comportent de cette façon, avec un grand nombre de propriétés, sans répétition, nous avons l'avantage habituel de réutiliser le code: si nous décidons plus tard de changer macro, et la structure de chaque classe qui l'utilise changera.

Lire les macros en ligne: https://riptutorial.com/fr/common-lisp/topic/1257/les-macros

Chapitre 17: LOOP, une macro Common Lisp pour itération

Examples

Boucles délimitées

Nous pouvons répéter une action un certain nombre de fois en utilisant la `repeat` .

```
CL-USER> (loop repeat 10 do (format t "Hello!~%"))
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
NIL
CL-USER> (loop repeat 10 collect (random 50))
(28 46 44 31 5 33 43 35 37 4)
```

Looping over Sequences

```
(loop for i in '(one two three four five six)
      do (print i))
(loop for i in '(one two three four five six) by #'cddr
      do (print i)) ;prints ONE THREE FIVE

(loop for i on '(a b c d e f g)
      do (print (length i))) ;prints 7 6 5 4 3 2 1
(loop for i on '(a b c d e f g) by #'cddr
      do (print (length i))) ;prints 7 5 3 1
(loop for i on '(a b c)
      do (print i)) ;prints (a b c) (b c) (c)

(loop for i across #(1 2 3 4 5 6)
      do (print i)) ; prints 1 2 3 4 5 6
(loop for i across "foo"
      do (print i)) ; prints #\f #\o #\o
(loop for element across "foo"
      for i from 0
      do (format t "~a ~a~%" i element)) ; prints 0 f\n1 o\n1 o
```

Voici un résumé des mots-clés

Mot-clé	Type de séquence	Type variable
dans	liste	élément de liste

Mot-clé	Type de séquence	Type variable
sur	liste	un peu de cdr de liste
à travers	vecteur	élément du vecteur

En boucle sur des tables de hachage

```
(defvar *ht* (make-hash-table))
(loop for (sym num) on
        '(one 1 two 2 three 3 four 4 five 5 six 6 seven 7 eight 8 nine 9 ten 10)
        by #'cddr
      do (setf (gethash sym *ht*) num))

(loop for k being each hash-key of *ht*
      do (print k)) ; iterate over the keys
(loop for k being the hash-keys in *ht* using (hash-value v)
      do (format t "~a=>~a~%" k v))
(loop for v being the hash-value in *ht*
      do (print v))
(loop for v being each hash-values of *ht* using (hash-key k)
      do (format t "~a=>~a~%" k v))
```

Formulaire LOOP simple

Formulaire LOOP simple sans mots-clés spéciaux:

```
(loop forms...)
```

Pour sortir de la boucle, nous pouvons utiliser `(return <return value>)` `

Quelques exemples:

```
(loop (format t "Hello~%"))  ; prints "Hello" forever
(loop (print (eval (read)))) ; your very own REPL
(loop (let ((r (read)))
        (typecase r
         (number (return (print (* r r))))
         (otherwise (format t "Not a number!~%")))))
```

En boucle sur les paquets

```
(loop for s being the symbols in 'cl
      do (print s))
(loop for s being the present-symbols in :cl
      do (print s))
(loop for s being the external-symbols in (find-package "COMMON LISP")
      do (print s))
(loop for s being each external-symbols of "COMMON LISP"
      do (print s))
(loop for s being each external-symbol in pack ;pack is a variable containing a package
      do (print s))
```

Boucles arithmétiques

```
(loop for i from 0 to 10
      do (print i)) ; prints 0 1 2 3 4 5 6 7 8 9 10
(loop for i from 0 below 10
      do (print i)) ; prints 0 1 2 3 4 5 6 7 8 9 10
(loop for i from 10 above 0
      do (print i)) ; prints 10 9 8 7 6 5 4 3 2 1
(loop for i from 10 to 0
      do (print i)) ; prints nothing
(loop for i from 10 downto 0
      do (print i)) ; prints 10 9 8 7 6 5 4 3 2 1 0
(loop for i downfrom 10 to 0
      do (print i)) ; same as above
(loop for i from 1 to 100 by 10
      do (print i)) ; prints 1 11 21 31 41 51 61 71 81 91
(loop for i from 100 downto 0 by 10
      do (print i)) ; prints 100 90 80 70 60 50 40 30 20 10 0
(loop for i from 1 to 10 by (1+ (random 3))
      do (print i)) ; note that (random 3) is evaluated only once
(let ((step (random 3)))
  (loop for i from 1 to 10 by (+ step 1)
        do (print i))) ; equivalent to the above
(loop for i from 1 to 10
      for j from 11 by 11
      do (format t "~2d ~3d~%" i j)) ;prints 1 11\n2 22\n...10 110
```

Destructuration dans les déclarations FOR

Nous pouvons déstructurer les listes d'objets composés

```
CL-USER> (loop for (a . b) in '((1 . 2) (3 . 4) (5 . 6)) collect a)
(1 3 5)
CL-USER> (loop for (a . b) in '((1 . 2) (3 . 4) (5 . 6)) collect b)
(2 4 6)
CL-USER> (loop for (a b c) in '((1 2 3) (4 5 6) (7 8 9) (10 11 12)) collect b)
(2 5 8 11)
```

Nous pouvons également déstructurer une liste elle-même

```
CL-USER> (loop for (a . b) on '(1 2 3 4 5 6) collect a)
(1 2 3 4 5 6)
CL-USER> (loop for (a . b) on '(1 2 3 4 5 6) collect b)
((2 3 4 5 6) (3 4 5 6) (4 5 6) (5 6) (6) NIL)
```

C'est utile quand on veut parcourir seulement certains éléments

```
CL-USER> (loop for (a . b) on '(1 2 3 4 5 6) by #'cddr collect a)
(1 3 5)
CL-USER> (loop for (a . b) on '(1 2 3 4 5 6) by #'cdddr collect a)
(1 4)
```

Utiliser `NIL` pour ignorer un terme:

```
(loop for (a nil . b) in '((1 2 . 3) (4 5 . 6) (7 8 . 9))
      collect (list a b)) ;=> ((1 3) (4 6) (7 9))
(loop for (a b) in '((1 2) (3 4) (5 6)) ;(a b) == (a b . nil)
      collect (+ a b)) ;=> (3 7 11)

; iterating over a window in a list
(loop for (pre x post) on '(1 2 3 4 5 3 2 1 2 3 4)
      for nth from 1
      while (and x post) ; checks that we have three elements of the list
      if (and (<= post x) (<= pre x)) collect (list :max x nth)
      if (and (>= post x) (>= pre x)) collect (list :min x nth))
; The above collects local minima/maxima
```

BOUCLE comme une expression

Contrairement aux boucles dans presque tous les autres langages de programmation utilisés aujourd'hui, la `LOOP` dans Common Lisp peut être utilisée comme une expression:

```
(let ((doubled (loop for x from 1 to 10
                     collect (* 2 x))))
    doubled) ;; ==> (2 4 6 8 10 12 14 16 18 20)

(loop for x from 1 to 10 sum x)
```

`MAXIMIZE` fait en sorte que la `LOOP` renvoie la plus grande valeur évaluée. `MINIMIZE` est l'opposé de `MAXIMIZE` .

```
(loop repeat 100
      for x = (random 1000)
      maximize x)
```

`COUNT` vous indique combien de fois une expression évaluée comme non `NIL` pendant la boucle:

```
(loop repeat 100
      for x = (random 1000)
      count (evenp x))
```

`LOOP` aussi des équivalents des fonctions `some` , `every` et `notany` :

```
(loop for ch across "foobar"
     thereis (eq ch #\a))

(loop for x in '(a b c d e f 1)
    always (symbolp x))

(loop for x in '(1 3 5 7)
    never (evenp x))
```

... sauf qu'ils ne sont pas limités à des itérations sur des séquences:

```
(loop for value = (read *standard-input* nil :eof)
   until (eq value :eof)
   never (stringp value))
```

Les verbes générant une valeur de `LOOP` peuvent également être écrits avec un suffixe -ing:

```
(loop repeat 100
      for x = (random 1000)
      minimizing x)
```

Il est également possible de capturer la valeur générée par ces verbes dans des variables (qui sont créées implicitement par la macro `LOOP`), de sorte que vous pouvez générer plusieurs valeurs à la fois:

```
 (loop repeat 100
     for x = (random 1000)
     maximizing x into biggest
     minimizing x into smallest
     summing x into total
     collecting x into xs
     finally (return (values biggest smallest total xs)))
```

Vous pouvez avoir plusieurs clauses de `collect` , de `count` , etc. qui collectent la même valeur de sortie. Ils seront exécutés en séquence.

Ce qui suit convertit une liste d'association (que vous pouvez utiliser avec `assoc`) dans une liste de propriétés (que vous pouvez utiliser avec `getf`):

```
(loop for (key . value) in assoc-list
      collect key
      collect value)
```

Bien que ce soit un meilleur style:

```
(loop for (key . value) in assoc-list
      append (list key value))
```

Exécution conditionnelle des clauses LOOP

`LOOP` a sa propre instruction `IF` qui peut contrôler la manière dont les clauses sont exécutées:

```
(loop repeat 1000
      for x = (random 100)
      if (evenp x)
        collect x into evens
      else
        collect x into odds
      finally (return (values evens odds)))
```

La combinaison de plusieurs clauses dans un corps IF nécessite une syntaxe spéciale:

```
 (loop repeat 1000
       for x = (random 100)
       if (evenp x)
          collect x into evens
```

```
          and do (format t "~a is even!~%" x)
       else
          collect x into odds
          and count t into n-odds
       finally (return (values evens odds n-odds)))
```

Itération parallèle

Plusieurs clauses `FOR` sont autorisées dans une `LOOP` . La boucle se termine lorsque la première de ces clauses se termine:

```
(loop for a in '(1 2 3 4 5)
      for b in '(a b c)
      collect (list a b))
;; Evaluates to: ((1 a) (2 b) (3 c))
```

D'autres clauses qui déterminent si la boucle doit continuer peuvent être combinées:

```
(loop for a in '(1 2 3 4 5 6 7)
      while (< a 4)
      collect a)
;; Evaluates to: (1 2 3)

(loop for a in '(1 2 3 4 5 6 7)
      while (< a 4)
      repeat 1
      collect a)
;; Evaluates to: (1)
```

Déterminez quelle liste est la plus longue, en supprimant l'itération dès que la réponse est connue:

```
(defun longerp (list-1 list-2)
    (loop for cdr1 on list-1
          for cdr2 on list-2
          if (null cdr1) return nil
          else if (null cdr2) return t
          finally (return nil)))
```

Numérotation des éléments d'une liste:

```
(loop for item in '(a b c d e f g)
      for x from 1
      collect (cons x item))
;; Returns ((1 . a) (2 . b) (3 . c) (4 . d) (5 . e) (6 . f) (7 . g))
```

Assurez-vous que tous les numéros d'une liste sont pairs, mais uniquement pour les 100 premiers éléments:

```
(assert
   (loop for number in list
         repeat 100
         always (evenp number)))
```

Itération imbriquée

La syntaxe spéciale `LOOP NAMED foo` vous permet de créer une boucle à partir de laquelle vous pouvez sortir. La sortie est effectuée à l'aide du `return-from` et peut être utilisée à partir de boucles imbriquées.

Ce qui suit utilise une boucle imbriquée pour rechercher un nombre complexe dans un tableau 2D:

```
(loop named top
      for x from 0 below (array-dimension *array* 1)
      do (loop for y from 0 below (array-dimension *array* 0))
               for n = (aref *array* y x)
              when (complexp n)
                do (return-from top (values n x y))))
```

Clause RETURN versus formulaire RETOUR.

Dans une `LOOP` , vous pouvez utiliser le formulaire Common Lisp `(return)` dans toute expression, ce qui entraînera une évaluation immédiate de la forme `LOOP` par rapport à la valeur `return` .

`LOOP` également une clause de `return` qui fonctionne de manière presque identique, la seule différence étant que vous ne l'entourez pas de parenthèses. La clause est utilisée dans le DSL de `LOOP` , tandis que le formulaire est utilisé dans les expressions.

```
(loop for x in list
      do (if (listp x) ;; Non-barewords after DO are expressions
             (return :x-has-a-list)))

;; Here, both the IF and the RETURN are clauses
(loop for x in list
     if (listp x) return :x-has-a-list)

;; Evaluate the RETURN expression and assign it to X...
;; except RETURN jumps out of the loop before the assignment
;; happens.
(loop for x = (return :nothing-else-happens)
      do (print :this-doesnt-print))
```

La chose après `finally` doit être une expression, donc la forme `(return)` doit être utilisée et non la clause de `return` :

```
 (loop for n from 1 to 100
       when (evenp n) collect n into evens
       else collect n into odds
      finally return (values evens odds)) ;; ERROR!

 (loop for n from 1 to 100
       when (evenp n) collect n into evens
       else collect n into odds
      finally (return (values evens odds))) ;; Correct usage.
```

En boucle sur une fenêtre d'une liste

Quelques exemples pour une fenêtre de taille 3:

```
;; Naïve attempt:
(loop for (first second third) on '(1 2 3 4 5)
      do (print (* first second third)))
;; prints 6 24 60 then Errors on (* 4 5 NIL)

;; We will try again and put our attempt into a function
(defun loop-3-window1 (function list)
  (loop for (first second third) on list
        while (and second third)
        do (funcall function first second third)))
(loop-3-window1 (lambda (a b c) (print (* a b c))) '(1 2 3 4 5))
;; prints 6 24 60 and returns NIL
(loop-3-window1 (lambda (a b c) (print (list a b c))) '(a b c d nil nil e f))
;; prints (a b c) (b c d) then returns NIL

;; A second attempt
(defun loop-3-window2 (function list)
  (loop for x on list
        while (nthcdr 2 x) ;checks if there are at least 3 elements
        for (first second third) = x
        do (funcall function first second third)))
(loop-3-window2 (lambda (a b c) (print (list a b c))) '(a b c d nil nil e f))
;; prints (a b c) (b c d) (c d nil) (c nil nil) (nil nil e) (nil e f)

;; A (possibly) more efficient function:
(defun loop-3-window2 (function list)
  (let ((f0 (pop list))
        (s0 (pop list)))
    (loop for first = f0 then second
          and second = s0 then third
          and third in list
          do (funcall function first second third))))

;; A more general function:
(defun loop-n-window (n function list)
  (loop for x on list
        while (nthcdr (1- n) x)
        do (apply function (subseq x 0 n))))
;; With potentially efficient implementation:
(define-compiler-macro loop-n-window (n function list &whole w)
  (if (typep n '(integer 1 #.call-arguments-limit))
     (let ((vars (loop repeat n collect (gensym)))
           (vars0 (loop repeat (1- n) collect (gensym)))
           (lst (gensym)))
       `(let ((,lst ,list))
          (let ,(loop for v in vars0 collect `(,v (pop ,lst)))
            (loop for
                  ,@(loop for v0 in vars0 for (v vn) on vars
                     collect v collect '= collect v0 collect 'then collect vn
                     collect 'and)
                  ,(car (last vars)) in ,lst
                  do ,(if (and (consp function) (eq 'function (car function))
     w
```

Lire LOOP, une macro Common Lisp pour itération en ligne: https://riptutorial.com/fr/common-lisp/topic/1369/loop--une-macro-common-lisp-pour-iteration

Chapitre 18: Mappage des fonctions sur les listes

Examples

Vue d'ensemble

Un ensemble de fonctions de cartographie de haut niveau est disponible dans Common Lisp, pour appliquer une fonction aux éléments d'une ou de plusieurs listes. Ils diffèrent dans la manière dont la fonction est appliquée aux listes et comment le résultat final est obtenu. Le tableau suivant récapitule les différences et montre pour chacun d'eux le formulaire LOOP équivalent. *f* est la fonction à appliquer, qui doit avoir un nombre d'arguments égal au nombre de listes; "Appliqué à la voiture" signifie qu'il est appliqué à son tour aux éléments des listes, "appliqué au cdr" signifie qu'il est appliqué à son tour aux listes, à leur cdr, à leur cddr, etc. la colonne «retourne» indique si le résultat global est obtenu en listant les résultats, en les concaténant (ils doivent donc être des listes!) ou simplement en utilisant des effets secondaires (et dans ce cas la première liste est renvoyée).

Fonction	Appliqué à	Résultats	LOOP équivalente
(mapcar fl $_1$ ... l $_n$)	voiture	liste des résultats	(boucle pour x $_1$ en l $_1$ … pour x $_n$ en l $_n$ collect (fx $_1$ … x $_n$))
(maplist fl $_1$ … l $_n$)	cdr	liste des résultats	(boucle pour x $_1$ sur l $_1$ … pour x $_n$ sur l $_n$ collect (fx $_1$ … x $_n$))
(mapcan fl $_1$ … l $_n$)	voiture	concaténation des résultats	(boucle pour x $_1$ en l $_1$ … pour x $_n$ en l $_n$ nconc (fx $_1$ … x $_n$))
(mapcon fl $_1$ … l $_n$)	cdr	concaténation des résultats	(boucle pour x $_1$ sur l $_1$ … pour x $_n$ sur l $_n$ nconc (fx $_1$ … x $_n$))
(mapc fl $_1$ … l $_n$)	voiture	l $_1$	(boucle pour x $_1$ en l $_1$ … pour x $_n$ en l $_n$ do (fx $_1$ … x $_n$) enfin (retour l $_1$))
(mapl fl $_1$ … l $_n$)	cdr	l $_1$	(boucle pour x $_1$ sur l $_1$ … pour x $_n$ sur l $_n$ do (fx $_1$ … x $_n$) enfin (retour l $_1$))

Notez que, dans tous les cas, les listes peuvent avoir des longueurs différentes et que l'application se termine lorsque la liste la plus courte est terminée.

Un autre couple de fonctions cartographiques est disponible: la `map` , qui peut être appliquée à des séquences (chaînes, vecteurs, listes), analogue à `mapcar` , et qui peut renvoyer tout type de séquence, spécifié comme premier argument, et `map-into` , analogue à `map` , mais cela modifie de

façon destructive son premier argument de séquence pour conserver les résultats de l'application de la fonction.

Exemples de MAPCAR

MAPCAR est la fonction la plus utilisée de la famille:

```
CL-USER> (mapcar #'1+ '(1 2 3))
(2 3 4)
CL-USER> (mapcar #'cons '(1 2 3) '(a b c))
((1 . A) (2 . B) (3 . C))
CL-USER> (mapcar (lambda (x y z) (+ (* x y) z))
                 '(1 2 3)
                 '(10 20 30)
                 '(100 200 300))
(110 240 390)
CL-USER> (let ((list '(a b c d e f g h i))) ; randomize this list
           (mapcar #'cdr
                   (sort (mapcar (lambda (x)
                                   (cons (random 100) x))
                                 list)
                         #'<=
                         :key #'car)))
(I D A G B H E C F)
```

Une utilisation idiomatique de `mapcar` consiste à transposer une matrice représentée par une liste de listes:

```
CL-USER> (defun transpose (list-of-lists)
           (apply #'mapcar #'list list-of-lists))
ROTATE
CL-USER> (transpose '((a b c) (d e f) (g h i)))
((A D G) (B E H) (C F I))

;  +---+---+---+              +---+---+---+
;  | A | B | C |              | A | D | G |
;  +---+---+---+              +---+---+---+
;  | D | E | F |    becomes   | B | E | H |
;  +---+---+---+              +---+---+---+
;  | G | H | I |              | C | F | I |
;  +---+---+---+              +---+---+---+
```

Pour une explication, voir cette réponse .

Exemples de MAPLIST

```
CL-USER> (maplist (lambda (list) (cons 0 list)) '(1 2 3 4))
((0 1 2 3 4) (0 2 3 4) (0 3 4) (0 4))
CL-USER> (maplist #'append
                  '(a b c d -)
                  '(1 2 3))
((A B C D - 1 2 3) (B C D - 2 3) (C D - 3))
```

Exemples de MAPCAN et MAPCON

MAPCAN:

```
CL-USER> (mapcan #'reverse '((1 2 3) (a b c) (100 200 300)))
(3 2 1 C B A 300 200 100)
CL-USER> (defun from-to (min max)
           (loop for i from min to max collect i))
FROM-TO
CL-USER> (from-to 1 5)
(1 2 3 4 5)
CL-USER> (mapcan #'from-to '(1 2 3) '(5 5 5))
(1 2 3 4 5 2 3 4 5 3 4 5)
```

L'une des utilisations de MAPCAN consiste à créer une liste de résultats sans valeurs NIL:

```
CL-USER> (let ((l1 '(10 20 40)))
           (mapcan (lambda (x)
                     (if (member x l1)
                         (list x)
                       nil))
                   '(2 4 6 8 10 12 14 16 18 20
                     18 16 14 12 10 8 6 4 2)))
(10 20 10)
```

MAPCON:

```
CL-USER> (mapcon #'copy-list '(1 2 3))
(1 2 3 2 3 3)
CL-USER> (mapcon (lambda (l1 l2) (list (length l1) (length l2))) '(a b c d) '(d e f))
(4 3 3 2 2 1)
```

Exemples de MAPC et MAPL

MAPC:

```
CL-USER> (mapc (lambda (x) (print (* x x))) '(1 2 3 4))

1
4
9
16
(1 2 3 4)
CL-USER> (let ((sum 0))
           (mapc (lambda (x y) (incf sum (* x y)))
                 '(1 2 3)
                 '(100 200 300))
           sum)
1400  ; => (1 x 100) + (2 x 200) + (3 x 300)
```

MAPL:

```
CL-USER> (mapl (lambda (list) (print (reduce #'+ list))) '(1 2 3 4 5))

15
14
```

```
12
9
5
(1 2 3 4 5)
```

Lire Mappage des fonctions sur les listes en ligne: https://riptutorial.com/fr/common-lisp/topic/6064/mappage-des-fonctions-sur-les-listes

Chapitre 19: Personnalisation

Examples

Plus de fonctionnalités pour la boucle REPL (Read-Eval-Print-Loop) dans un terminal

CLISP a une intégration avec GNU Readline.

Pour des améliorations pour d'autres implémentations, voir: Comment personnaliser le fichier REPL de SBCL .

Fichiers d'initialisation

La plupart des implémentations Lisp communes essaieront de charger un *fichier init* au démarrage:

la mise en oeuvre	Fichier d'init	Fichier Site / System Init
ABCL	`$HOME/.abclrc`	
Allegro CL	`$HOME/.clinit.cl`	
ECL	`$HOME/.eclrc`	
Fermoir	`$HOME/.clasprc`	
CLISP	`$HOME/.clisprc.lisp`	
Clozure CL	`home:ccl-init.lisp` ou `home:ccl-init.fasl` ou `home:.ccl-init.lisp`	
CMUCL	`$HOME/.cmucl-init.lisp`	
LispWorks	`$HOME/.lispworks`	
MKCL	`$HOME/.mkclrc`	
SBCL	`$HOME/.sbclrc`	`$SBCL_HOME/sbclrc` ou `/etc/sbclrc`
SCL	`$HOME/.scl-init.lisp`	

Exemples de fichiers d'initialisation:

la mise en oeuvre	Exemple de fichier Init
LispWorks	`Library/lib/7-0-0-0/config/a-dot-lispworks.lisp`

Paramètres d'optimisation

Common Lisp a un moyen d'influencer les stratégies de compilation. Il est logique de définir vos valeurs préférées.

Les valeurs d'optimisation sont comprises entre 0 (sans importance) et 3 (extrêmement important). **1 est la valeur neutre.**

Il est utile de toujours utiliser le code sécurisé (safety = 3) avec toutes les vérifications d'exécution activées.

Notez que l'interprétation des valeurs est spécifique à l'implémentation. La plupart des implémentations Common Lisp utilisent ces valeurs.

Réglage	Explication	valeur par défaut utile	valeur de livraison utile
`compilation-speed`	vitesse du processus de compilation	2	0
`debug`	facilité de débogage	2	1 ou 0
`safety`	vérification des erreurs d'exécution	3	2
`space`	la taille du code et l'espace d'exécution	2	2
`speed`	vitesse du code objet	2	3

Une déclaration d' `optimize` à utiliser avec `declaim` , `declare` et `proclaim` :

```
(optimize (compilation-speed 2)
          (debug 2)
          (safety 3)
          (space 2)
          (speed 2))
```

Notez que vous pouvez également appliquer des paramètres d'optimisation spéciaux à des parties du code dans une fonction utilisant la macro `LOCALLY` .

Lire Personnalisation en ligne: https://riptutorial.com/fr/common-lisp/topic/5679/personnalisation

Chapitre 20: Protocole de méta-objet CLOS

Examples

Obtenir les noms d'emplacement d'une classe

Disons que nous avons une classe comme

```
(defclass person ()
  (name email age))
```

Pour obtenir les noms des slots de la classe, nous utilisons les slots de classes de fonctions. Cela peut être trouvé dans le paquet de vadrouille rapprochée, fourni par le système de vadrouille rapprochée. Pour le charger, l'image lisp courante que nous utilisons `(ql:quickload :closer-mop)` . Nous devons également nous assurer que la classe est finalisée avant d'appeler des classes.

```
(let ((class (find-class 'person)))
  (c2mop:ensure-finalized class)
  (c2mop:class-slots class))
```

qui renvoie une liste d'objets de *définition de logement effectifs* :

```
(#<SB-MOP:STANDARD-EFFECTIVE-SLOT-DEFINITION S/TRANSFORMATIONS::NAME>
 #<SB-MOP:STANDARD-EFFECTIVE-SLOT-DEFINITION S/TRANSFORMATIONS::EMAIL>
 #<SB-MOP:STANDARD-EFFECTIVE-SLOT-DEFINITION S/TRANSFORMATIONS::AGE>)
```

Mettre à jour un emplacement lorsqu'un autre emplacement est modifié

Le CLOS MOP fournit le hook slot-value-using-class, qui est appelé lorsqu'un accès à une valeur est accédé, lu ou modifié. Parce que nous ne prenons en compte que les modifications dans ce cas, nous définissons une méthode pour `(setf slot-value-using-class)` .

```
(defclass document ()
  ((id :reader id :documentation "A hash computed with the contents of every other slot")
   (title :initarg :title :accessor title)
   (body :initarg :body :accessor body)))

(defmethod (setf c2mop:slot-value-using-class) :after
    (new class (object document) (slot c2mop:standard-effective-slot-definition))
  ;; To avoid this method triggering a call to itself, we check that the slot
  ;; the modification occurred in is not the slot we are updating.
  (unless (eq (slot-definition-name slot) 'id)
    (setf (slot-value object 'id) (hash-slots object))))
```

Notez que parce que la création de la `slot-value` instance n'est pas appelée, il peut être nécessaire de dupliquer le code dans l' `initialize-instance :after` méthode

```
(defmethod initialize-instance :after ((obj document) &key)
```

```
  (setf (slot-value obj 'id)
        (hash-slots obj)))
```

Lire Protocole de méta-objet CLOS en ligne: https://riptutorial.com/fr/common-lisp/topic/2901/protocole-de-meta-objet-clos

Chapitre 21: Récursivité

Remarques

Lisp est souvent utilisé dans des contextes éducatifs, où les étudiants apprennent à comprendre et à implémenter des algorithmes récursifs.

Le code de production écrit en Common Lisp ou en code portable présente plusieurs problèmes de récursivité: ils n'utilisent pas de fonctionnalités spécifiques à l'implémentation telles que l' *optimisation des appels de queue* , ce qui rend souvent nécessaire d'éviter la récursivité. Dans ces cas, les implémentations:

- A généralement une *limite de profondeur de récursivité en* raison des limites de taille des piles. Les algorithmes récursifs ne fonctionneront donc que pour des données de taille limitée.
- Ne fournissez pas toujours une optimisation des appels de fin, en particulier en combinaison avec des opérations de portée dynamique.
- Fournit uniquement une optimisation des appels de queue à certains niveaux d'optimisation.
- Ne fournissez pas généralement une *optimisation de l'appel de la queue* .
- Généralement, ne fournit pas *d'optimisation de l'appel* sur certaines plates-formes. Par exemple, les implémentations sur JVM peuvent ne pas le faire, car la JVM elle-même ne prend pas en charge l' *optimisation des appels de queue* .

Remplacer les appels de queue par des sauts rend généralement le débogage plus difficile; L'ajout de sauts entraînera l'indisponibilité des cadres de pile dans un débogueur. Comme alternative propose Common Lisp:

- Constructions `DOTIMES` , comme `DO` , `DOTIMES` , `LOOP` et autres
- Fonctions d'ordre supérieur, telles que `MAP` , `REDUCE` et autres
- Diverses structures de contrôle, y compris à bas niveau, `go to`

Examples

Modèle de récursivité 2 multi-condition

```
(defun fn (x)
     (cond (test-condition1 the-value1)
           (test-condition2 the-value2)
           ...
           ...
           ...
           (t (fn reduced-argument-x))))


CL-USER 2788 > (defun my-fib (n)
                 (cond ((= n 1) 1)
                       ((= n 2) 1)
                       (t (+
```

```
                          (my-fib (- n 1))
                          (my-fib (- n 2))))))
MY-FIB

CL-USER 2789 > (my-fib 1)
1

CL-USER 2790 > (my-fib 2)
1

CL-USER 2791 > (my-fib 3)
2

CL-USER 2792 > (my-fib 4)
3

CL-USER 2793 > (my-fib 5)
5

CL-USER 2794 > (my-fib 6)
8

CL-USER 2795 > (my-fib 7)
13
```

Modèle de récursivité 1 récursivité simple à une seule condition

```
(defun fn (x)
  (cond (test-condition the-value)
        (t (fn reduced-argument-x))))
```

Calculez le nième nombre de Fibonacci

```
;;Find the nth Fibonacci number for any n > 0.
;; Precondition: n > 0, n is an integer. Behavior undefined otherwise.
(defun fibonacci (n)
    (cond
        (                                   ;; Base case.
             ;; The first two Fibonacci numbers (indices 1 and 2) are 1 by definition.
            (<= n 2)                        ;; If n <= 2
            1                               ;; then return 1.
        )
        (t                                  ;; else
            (+                              ;; return the sum of
                                            ;; the results of calling
                (fibonacci (- n 1))         ;; fibonacci(n-1) and
                (fibonacci (- n 2))         ;; fibonacci(n-2).
                                            ;; This is the recursive case.
            )
        )
    )
)
```

Imprime récursivement les éléments d'une liste

```
;;Recursively print the elements of a list
```

```
(defun print-list (elements)
    (cond
        ((null elements) '()) ;; Base case: There are no elements that have yet to be printed.
Don't do anything and return a null list.
        (t
            ;; Recursive case
            ;; Print the next element.
            (write-line (write-to-string (car elements)))
            ;; Recurse on the rest of the list.
            (print-list (cdr elements))
        )
    )
)
```

Pour tester cela, lancez:

```
(setq test-list '(1 2 3 4))
(print-list test-list)
```

Le résultat sera:

```
1
2
3
4
```

Calculer la factorielle d'un nombre entier

Un algorithme facile à implémenter en tant que fonction récursive est factoriel.

```
;;Compute the factorial for any n >= 0. Precondition: n >= 0, n is an integer.
(defun factorial (n)
    (cond
        ((= n 0) 1) ;; Special case, 0! = 1
        ((= n 1) 1) ;; Base case, 1! = 1
        (t
            ;; Recursive case
            ;; Multiply n by the factorial of n - 1.
            (* n (factorial (- n 1)))
        )
    )
)
```

Lire Récursivité en ligne: https://riptutorial.com/fr/common-lisp/topic/3190/recursivite

Chapitre 22: Regroupement des formulaires

Examples

Quand le groupement est-il nécessaire?

Dans certains endroits de Common Lisp, une série de formulaires est évaluée dans l'ordre. Par exemple, dans le corps d'un **défunt** ou d'un **lambda** , ou dans le corps d'un **pointime** . Dans ces cas, l'écriture de plusieurs formulaires dans l'ordre fonctionne comme prévu. Dans quelques endroits, cependant, comme les parties *then* et *else* d'une expression **if** , une seule forme est autorisée. Bien sûr, on peut vouloir évaluer plusieurs expressions à ces endroits. Pour ces situations, une forme implicite de regroupement explicite est nécessaire.

Progn

L'objectif général opérateur spécial **progn** est utilisé pour l' évaluation de zéro ou de plusieurs formulaires. La valeur du dernier formulaire est renvoyée. Par exemple, dans ce qui suit, **(print 'hello)** est évalué (et son résultat est ignoré), puis **42** est évalué et son résultat (**42**) est renvoyé:

```
(progn
  (print 'hello)
  42)
;=> 42
```

S'il n'y a pas de formulaires dans le **progn** , alors **rien** n'est renvoyé:

```
(progn)
;=> NIL
```

En plus de regrouper une série de formulaires, **progn a** également pour **caractéristique** importante que si le formulaire de **progn** est un *formulaire de niveau supérieur* , tous les formulaires qu'il **contient** sont traités comme des formulaires de niveau supérieur. Cela peut être important lorsque vous écrivez des macros qui se développent dans plusieurs formulaires qui doivent tous être traités comme des formulaires de niveau supérieur.

Progn est également précieux car il renvoie *toutes* les valeurs de la dernière forme. Par exemple,

```
(progn
  (print 'hello)
  (values 1 2 3))
;;=> 1, 2, 3
```

En revanche, certaines expressions de regroupement renvoient uniquement la *valeur primaire* du formulaire produisant des résultats.

Progns implicites

Certaines formes utilisent des *progiciels implicites* pour décrire leur comportement. Par exemple, le **quand** et à **moins** que des macros, qui sont essentiellement à sens unique **si** les formes, décrivent leur comportement en termes d'un *progn implicite.* Cela signifie qu'une forme comme

```
(when (foo-p foo)
  form1
  form2)
```

est évalué et la condition **(foo-p foo)** est vraie, alors *form1* et *form2* sont regroupés comme s'ils étaient contenus dans un **progn** . L'expansion de la macro **quand** est essentiellement:

```
(if (foo-p foo)
  (progn
    form1
    form2)
  nil)
```

Prog1 et Prog2

Souvent, il est utile d'évaluer plusieurs expressions et de renvoyer le résultat de la première ou de la seconde forme plutôt que de la dernière. Ceci est facile à réaliser en utilisant **let** et, par exemple:

```
(let ((form1-result form1))
  form2
  form3
  ;; ...
  form-n-1
  form-n
  form1-result)
```

Comme cette forme est courante dans certaines applications, Common Lisp inclut **prog1** et **prog2** qui sont comme **progn** , mais renvoient respectivement le résultat des première et seconde formes. Par exemple:

```
(prog1
  42
  (print 'hello)
  (print 'goodbye))
;; => 42
```

```
(prog2
  (print 'hello)
  42
  (print 'goodbye))
;; => 42
```

Une distinction importante entre **prog1** / **prog2** et **progn est** cependant que **progn** renvoie *toutes* les valeurs de la dernière forme, alors que **prog1** et **prog2** ne renvoient que la valeur primaire de la première et de la deuxième forme. Par exemple:

```
(progn
  (print 'hello)
  (values 1 2 3))
;;=> 1, 2, 3

(prog1
  (values 1 2 3)
  (print 'hello))
;;=> 1              ; not 1, 2, 3
```

Pour les valeurs multiples avec évaluation de style **prog1** , utilisez plutôt **multiple-value-prog1** . Il n'y a pas de **prog2 multi-value** similaire, mais il n'est pas difficile à mettre en œuvre si vous en avez besoin.

Bloc

Le **bloc** opérateur spécial permet le regroupement de plusieurs formes Lisp (comme un `progn` implicite) et prend également un *nom* pour nommer le bloc. Lorsque les formulaires du bloc sont évalués, l'opérateur de **retour** spécial peut être utilisé pour quitter le bloc. Par exemple:

```
(block foo
  (print 'hello)     ; evaluated
  (return-from foo)
  (print 'goodbye))  ; not evaluated
;;=> NIL
```

Le retour peut également être fourni avec une valeur de retour:

```
(block foo
  (print 'hello)     ; evaluated
  (return-from foo 42)
  (print 'goodbye))  ; not evaluated
;;=> 42
```

Les blocs nommés sont utiles lorsqu'un morceau de code a un nom significatif ou lorsque des blocs sont imbriqués. Dans certains contextes, seule la capacité de revenir d'un bloc tôt est importante. Dans ce cas, vous pouvez utiliser **nil** comme nom de bloc et **retourner** . **Return** est comme un **retour** , sauf que le nom du bloc est toujours **nul** .

Remarque: les formulaires fermés ne sont pas des formulaires de premier niveau. C'est différent de `progn` , où les formulaires ci - joints d'un haut niveau `progn` forme sont toujours considérés comme des formes *de haut niveau.*

Tagbody

Pour beaucoup de contrôle dans les formulaires de groupe, l'opérateur spécial **tagbody** peut être très utile. Les formes à l'intérieur d'une forme de **tagbody** sont soit des *balises go* (qui ne sont que des symboles ou des entiers), soit des formulaires à exécuter. Dans un **tagbody** , l'opérateur spécial **go** est utilisé pour transférer l'exécution à un nouvel emplacement. Ce type de programmation peut être considéré comme assez bas car il permet des chemins d'exécution arbitraires. Voici un exemple détaillé de ce à quoi pourrait ressembler une for-loop lorsqu'elle est

implémentée en tant que **tagbody** :

```
(let (x)                ; for (x = 0; x < 5; x++) { print(hello); }
  (tagbody
     (setq x 0)
   prologue
     (unless (< x 5)
       (go end))
   begin
     (print (list 'hello x))
   epilogue
     (incf x)
     (go prologue)
   end))
```

Alors que **tagbody** et **go** ne sont pas couramment utilisés, peut-être à cause de "GOTO considéré comme dangereux", mais peuvent être utiles lors de l'implémentation de structures de contrôle complexes comme les machines à états. De nombreuses constructions d'itération se développent également dans un *tagbody implicite* . Par exemple, le corps d'un **dotimes** est spécifié comme une série d'étiquettes et de formes.

Quelle forme utiliser?

Lorsque vous écrivez des macros qui se développent dans des formulaires pouvant impliquer un regroupement, il est utile de prendre le temps de réfléchir à la structure de regroupement à développer.

Pour les formes de style de définition, par exemple, une macro de **définition de widget** qui apparaîtra généralement comme une forme de niveau supérieur, et que plusieurs **définitions de def** , **structures de flux** , etc., il est **préférable** d'utiliser un **progn** , traitées comme des formulaires de niveau supérieur. Pour les formulaires d'itération, un **tagbody** implicite est plus commun.

Par exemple, le corps de **dotimes** , **dolist** , et **font** chacun développer un **tagbody** implicite.

Pour les formulaires qui définissent un "morceau" de code nommé, un **bloc** implicite est souvent utile. Par exemple, alors que le corps d'un **defun** est à l'intérieur d'un **progn** implicite, ce **progn** implicite est dans un bloc partageant le nom de la fonction. Cela signifie que le **retour** peut être utilisé pour quitter la fonction. Une telle comp

Lire Regroupement des formulaires en ligne: https://riptutorial.com/fr/common-lisp/topic/4892/regroupement-des-formulaires

Chapitre 23: Ruisseaux

Syntaxe

- `(read-char &optional stream eof-error-p eof-value recursive-p)` => caractère
- `(write-char character &optional stream)` => caractère
- `(read-line &optional stream eof-error-p eof-value recursive-p)` => ligne manquante-newline-p
- `(write-line line &optional stream)` => ligne

Paramètres

Paramètre	Détail
`stream`	Le flux à lire ou à écrire.
`eof-error-p`	Si une erreur est signalée si la fin du fichier est rencontrée.
`eof-value`	Quelle valeur doit être renvoyée si eof est rencontré, et `eof-error-p` est faux.
`recursive-p`	La lecture-opération est-elle appelée de manière récursive à partir de `READ` . Habituellement, cela devrait être laissé comme `NIL` .
`character`	Le caractère à écrire ou le caractère lu.
`line`	La ligne à écrire ou la ligne qui a été lue.

Examples

Créer des flux d'entrée à partir de chaînes

La macro `WITH-INPUT-FROM-STRING` peut être utilisée pour créer un flux à partir d'une chaîne.

```
(with-input-from-string (str "Foobar")
  (loop for i from 0
        for char = (read-char str nil nil)
        while char
        do (format t "~d: ~a~%" i char)))
; 0: F
; 1: o
; 2: o
; 3: b
; 4: a
; 5: r
;=> NIL
```

La même chose peut être faite manuellement en utilisant `MAKE-STRING-INPUT-STREAM` .

```
(let ((str (make-string-input-stream "Foobar")))
  (loop for i from 0
        for char = (read-char str nil nil)
        while char
        do (format t "~d: ~a~%" i char)))
```

Ecrire un résultat dans une chaîne

La macro `WITH-OUTPUT-TO-STRING` peut être utilisée pour créer un flux de sortie de chaîne et renvoyer la chaîne résultante à la fin.

```
(with-output-to-string (str)
  (write-line "Foobar!" str)
  (write-string "Barfoo!" str))
;=> "Foobar!
;   Barfoo!"
```

La même chose peut être faite manuellement en utilisant `MAKE-STRING-OUTPUT-STREAM` et `GET-OUTPUT-STREAM-STRING` .

```
(let ((str (make-string-output-stream)))
  (write-line "Foobar!" str)
  (write-string "Barfoo!" str)
  (get-output-stream-string str))
```

Ruisseaux gris

Les flux gris sont une extension non standard qui autorise les flux définis par l'utilisateur. Il fournit des classes et des méthodes que l'utilisateur peut étendre. Vous devriez vérifier votre manuel d'implémentation pour voir s'il fournit des flux de gris.

Pour un exemple simple, un flux de saisie de caractères qui renvoie des caractères aléatoires pourrait être implémenté comme ceci:

```
(defclass random-character-input-stream (fundamental-character-input-stream)
  ((character-table
    :initarg :character-table
    :initform "abcdefghijklmnopqrstuvwxyz
" ; The newline is necessary.
    :accessor character-table))
  (:documentation "A stream of random characters."))

(defmethod stream-read-char ((stream random-character-input-stream))
  (let ((table (character-table stream)))
    (aref table (random (length table)))))

(let ((stream (make-instance 'random-character-input-stream)))
  (dotimes (i 5)
    (print (read-line stream))))
; "gyaexyfjsqdcpciaaftoytsygdeycrrzwivwcfb"
; "gctnoxpajovjqjbkiqykdflbhfspmexjaaggonhydhayvknwpdydyiabithpt"
```

```
; "nvfxwzczfalosaqw"
; "sxeiejcovrtesbpmoppfvvjfvx"
; "hjplqgstbodbalnmxhsvxdox"
;=> NIL
```

Fichier de lecture

Un fichier peut être ouvert pour la lecture en tant que flux en utilisant la macro `WITH-OPEN-FILE` .

```
(with-open-file (file #P"test.file")
  (loop for i from 0
        for line = (read-line file nil nil)
        while line
        do (format t "~d: ~a~%" i line)))
; 0: Foobar
; 1: Barfoo
; 2: Quuxbar
; 3: Barquux
; 4: Quuxfoo
; 5: Fooquux
;=> T
```

La même chose peut être faite manuellement en utilisant `OPEN` et `CLOSE` .

```
(let ((file (open #P"test.file"))
      (aborted t))
  (unwind-protect
       (progn
         (loop for i from 0
               for line = (read-line file nil nil)
               while line
               do (format t "~d: ~a~%" i line))
         (setf aborted nil))
    (close file :abort aborted)))
```

Notez que `READ-LINE` crée une nouvelle chaîne pour chaque ligne. Cela peut être lent. Certaines implémentations fournissent une variante qui peut lire une ligne dans un tampon de chaîne. Exemple: `READ-LINE-INTO` pour Allegro CL.

Ecrire dans un fichier

Un fichier peut être ouvert pour l'écriture en tant que flux à l'aide de la macro `WITH-OPEN-FILE` .

```
(with-open-file (file #P"test.file" :direction :output
                                    :if-exists :append
                                    :if-does-not-exist :create)
  (dolist (line '("Foobar" "Barfoo" "Quuxbar"
                  "Barquux" "Quuxfoo" "Fooquux"))
    (write-line line file)))
```

La même chose peut être faite manuellement avec `OPEN` et `CLOSE` .

```
(let ((file (open #P"test.file" :direction :output
```

```
                                :if-exists :append
                                :if-does-not-exist :create)))
  (dolist (line '("Foobar" "Barfoo" "Quuxbar"
                  "Barquux" "Quuxfoo" "Fooquux"))
    (write-line line file))
  (close file))
```

Copier un fichier

Copier byte-by-byte d'un fichier

La fonction suivante copie un fichier dans un autre en effectuant une copie exacte octet par octet, en ignorant le type de contenu (qui peut être soit des lignes de caractères dans certaines données d'encodage ou binaires):

```
(defun byte-copy (infile outfile)
  (with-open-file (instream infile :direction :input :element-type '(unsigned-byte 8)
                           :if-does-not-exist nil)
    (when instream
      (with-open-file (outstream outfile :direction :output :element-type '(unsigned-byte 8)
                                 :if-exists :supersede)
        (loop for byte = (read-byte instream nil)
           while byte
           do (write-byte byte outstream))))))
```

Le type `(unsigned-byte 8)` est le type d'octets de 8 bits. Les fonctions `read-byte` et `write-byte` fonctionnent sur des octets, au lieu de `read-char` et `write-char` qui fonctionnent sur les caractères. `read-byte` renvoie un octet lu dans le flux ou `NIL` à la fin du fichier si le deuxième paramètre facultatif est `NIL` (sinon, il signale une erreur).

Copie en vrac

Une copie exacte, plus efficace que la précédente. peut être fait en lisant et en écrivant les fichiers avec de gros blocs de données à chaque fois, au lieu d'octets uniques:

```
(defun bulk-copy (infile outfile)
  (with-open-file (instream infile :direction :input :element-type '(unsigned-byte 8)
                           :if-does-not-exist nil)
    (when instream
      (with-open-file (outstream outfile :direction :output :element-type '(unsigned-byte 8)
                                 :if-exists :supersede)
        (let ((buffer (make-array 8192 :element-type '(unsigned-byte 8))))
          (loop for bytes-read = (read-sequence buffer instream)
             while (plusp bytes-read)
             do (write-sequence buffer outstream :end bytes-read)))))))
```

`read-sequence` et `write-sequence` sont utilisés ici avec un tampon qui est un vecteur d'octets (ils peuvent fonctionner sur des séquences d'octets ou de caractères). `read-sequence` remplit le tableau avec les octets lus à chaque fois et renvoie le nombre d'octets lus (qui peut être inférieur à la taille du tableau lorsque la fin du fichier est atteinte). Notez que le tableau est modifié de manière destructive à chaque itération.

Copie exacte ligne par ligne d'un fichier

Le dernier exemple est une copie effectuée en lisant chaque ligne de caractères du fichier d'entrée et en l'écrivant dans le fichier de sortie. Notez que, comme nous voulons une copie exacte, nous devons vérifier si la dernière ligne du fichier d'entrée est terminée ou non par un caractère de fin de ligne. Pour cette raison, nous utilisons les deux valeurs renvoyées par `read-line` : une nouvelle chaîne contenant les caractères de la ligne suivante et une valeur booléenne qui est *vraie* si la ligne est la dernière du fichier et ne contient pas le caractère de nouvelle ligne final (s). Dans ce cas, `write-string` est utilisé à la place de `write-line` , car le premier n'ajoute pas de nouvelle ligne à la fin de la ligne.

```
(defun line-copy (infile outfile)
  (with-open-file (instream infile :direction :input :if-does-not-exist nil)
    (when instream
      (with-open-file (outstream outfile :direction :output :if-exists :supersede)
        (let (line missing-newline-p)
          (loop
             (multiple-value-setq (line missing-newline-p)
               (read-line instream nil nil))
             (cond (missing-newline-p                          ; we are at the end of file
                    (when line (write-string line outstream))  ; note `write-string`
                    (return))                                  ; exit from simple loop
                   (t (write-line line outstream)))))))))
```

Notez que ce programme est indépendant de la plate-forme, car le ou les caractères de la nouvelle ligne (variant selon les systèmes d'exploitation) sont automatiquement gérés par les fonctions `read-line` et d' `write-line` .

Lecture et écriture de fichiers entiers vers et depuis des chaînes

La fonction suivante lit un fichier entier dans une nouvelle chaîne et la renvoie:

```
(defun read-file (infile)
  (with-open-file (instream infile :direction :input :if-does-not-exist nil)
    (when instream
      (let ((string (make-string (file-length instream))))
        (read-sequence string instream)
        string))))
```

Le résultat est `NIL` si le fichier n'existe pas.

La fonction suivante écrit une chaîne dans un fichier. Un paramètre de mot-clé est utilisé pour spécifier quoi faire si le fichier existe déjà (par défaut, il provoque une erreur, les valeurs admissibles sont celles de la macro `with-open-file`).

```
(defun write-file (string outfile &key (action-if-exists :error))
   (check-type action-if-exists (member nil :error :new-version :rename :rename-and-delete
                                        :overwrite :append :supersede))
   (with-open-file (outstream outfile :direction :output :if-exists action-if-exists)
     (write-sequence string outstream)))
```

Dans ce cas, la `write-sequence` peut être remplacée par une `write-string` d' `write-string` .

Lire Ruisseaux en ligne: https://riptutorial.com/fr/common-lisp/topic/3028/ruisseaux

Chapitre 24: sequence - comment diviser une séquence

Syntaxe

1. split regex chaîne-cible & limite de fin de clé avec-registres-p omit-unmatched-p sharedp => liste
2. lispworks: séquence de séparateur-sac à séquence fractionnée et clé de test de fin de touche de fin de touche coalesce-separators => séquences
3. séquence de délimitation de séquence fractionnée et clé début fin de compte de fin remove-empty -ookqs test test-pas clé => liste des sous-séquences

Examples

Fractionner des chaînes à l'aide d'expressions régulières

La bibliothèque CL-PPCRE fournit la fonction `split` qui nous permet de diviser des chaînes dans des sous-chaînes correspondant à une expression régulière, en ignorant les parties de la chaîne qui ne le font pas.

```
(cl-ppcre:split "\\." "127.0.0.1")
;; => ("127" "0" "0" "1")
```

SPLIT-SEQUENCE dans LIspWorks

Fractionnement simple d'une chaîne de numéros IP.

```
> (lispworks:split-sequence "." "127.0.0.1")
  ("127" "0" "0" "1")
```

Division simple d'une URL:

```
> (lispworks:split-sequence ".:/" "http://127.0.0.1/foo/bar.html"
                            :coalesce-separators t)
("http" "127" "0" "0" "1" "foo" "bar" "html")
```

Utilisation de la bibliothèque de séquences fractionnées

La bibliothèque de séquence de fractionnement fournit une fonction `split-sequence` , ce qui permet de séparer des éléments d'une séquence

```
(split-sequence:split-sequence #\Space "John Doe II")
;; => ("John" "Doe" "II")
```

Lire sequence - comment diviser une séquence en ligne: https://riptutorial.com/fr/common-lisp/topic/1454/sequence---comment-diviser-une-sequence

Chapitre 25: Structures de contrôle

Examples

Constructions conditionnelles

Dans Common Lisp, `if` est la construction conditionnelle la plus simple. Il a la forme `(if test then [else])` le `then` `test` `else` `(if test then [else])` et est évaluée pour `then` si `test` est vrai et d' `else` autrement. Le reste peut être omis.

```
(if (> 3 2)
    "Three is bigger!"
    "Two is bigger!")
;;=> "Three is bigger!"
```

Une différence très importante entre `if` dans Common Lisp et `if` dans de nombreux autres langages de programmation, CL est `if` une expression, pas une déclaration. En tant que tels, `if` formulaires renvoient des valeurs pouvant être affectées à des variables, utilisées dans les listes d'arguments, etc.:

```
;; Use a different format string depending on the type of x
(format t (if (numberp x)
              "~x~%"
              "~a~%")
          x)
```

De Common Lisp `if` peut être considéré comme équivalent à l' opérateur ternaire: en C # et d' autres langues « accolade ».

Par exemple, l'expression C # suivante:

```
year == 1990 ? "Hammertime" : "Not Hammertime"
```

Est équivalent au code Common Lisp suivant, en supposant que cette `year` contient un entier:

```
(if (eql year 1990) "Hammertime" "Not Hammertime")
```

`cond` est une autre construction conditionnelle. Il est quelque peu similaire à une chaîne de déclarations `if` et a la forme:

```
(cond (test-1 consequent-1-1 consequent-2-1 ...)
      (test-2)
      (test-3 consequent-3-1 ...)
      ... )
```

Plus précisément, `cond` a zéro ou plusieurs *clauses* , et chaque clause a un test suivi de zéro ou plusieurs conséquences. L'ensemble de la construction `cond` sélectionne la première clause dont

le test n'est pas évalué à `nil` et évalue ses conséquences dans l'ordre. Il renvoie la valeur de la dernière forme dans les conséquents.

```
(cond ((> 3 4) "Three is bigger than four!")
      ((> 3 3) "Three is bigger than three!")
      ((> 3 2) "Three is bigger than two!")
      ((> 3 1) "Three is bigger than one!"))
;;=> "Three is bigger than two!"
```

Pour fournir une clause par défaut à évaluer si aucune autre clause n'équivaut à `t` , vous pouvez ajouter une clause qui est vraie par défaut à l'aide de `t` . Ce concept est très similaire au concept `CASE...ELSE` de SQL, mais il utilise un booléen littéral true plutôt qu'un mot-clé pour accomplir la tâche.

```
(cond
    ((= n 1) "N equals 1")
    (t "N doesn't equal 1")
)
```

Une construction `if` peut être écrite comme une construction `cond` . `(if test then else)` et `(cond (test then) (t else))` sont équivalents.

Si vous n'avez besoin que d'une clause, utilisez `when` ou à `unless` :

```
(when (> 3 4)
  "Three is bigger than four.")
;;=> NIL

(when (< 2 5)
  "Two is smaller than five.")
;;=> "Two is smaller than five."

(unless (> 3 4)
  "Three is bigger than four.")
;;=> "Three is bigger than four."

(unless (< 2 5)
  "Two is smaller than five.")
;;=> NIL
```

La boucle do

La plupart des constructions en boucle et conditionnelles dans Common Lisp sont en fait des macros qui cachent des constructions plus basiques. Par exemple, les `dotimes` et `dolist` sont construits sur la macro `do` . Le formulaire pour `do` ressemble à ceci:

```
(do (varlist)
    (endlist)
   &body)
```

- `varlist` est composée des variables définies dans la boucle, de leurs valeurs initiales et de leur modification après chaque itération. La partie 'change' est évaluée à la fin de la boucle.

- `endlist` contient les conditions de fin et les valeurs renvoyées à la fin de la boucle. La condition de fin est évaluée au début de la boucle.

En voici un qui commence à 0 et va jusqu'à (non compris) 10.

```
;;same as (dotimes (i 10))
(do (( i (+ 1 i))
    ((< i 10) i)
   (print i))
```

Et en voici un qui se déplace dans une liste:

```
;;same as (dolist (item given-list)
(do ((item (car given-list))
     (temp list (cdr temp))
   (print item))
```

La partie `varlist` est similaire à celle d'une instruction `let` . Vous pouvez lier plusieurs variables et elles n'existent que dans la boucle. Chaque variable déclarée est dans son propre ensemble de parenthèses. En voici un qui compte combien de 1 et 2 sont dans une liste.

```
(let ((vars (list 1 2 3 2 2 1)))
  (do ((ones 0)
       (twos 0)
       (temp vars (cdr temp)))
      ((not temp) (list ones twos))
    (when (= (car temp) 1)
      (setf ones (+ 1 ones)))
    (when (= (car temp) 2)
      (setf twos (+ 1 twos)))))
-> (2 3)
```

Et si une macro de boucle while n'a pas été implémentée:

```
(do ()
    (t)
  (when task-done
    (break)))
```

Pour les applications les plus courantes, les macros `dotimes` et `doloop` plus spécifiques sont beaucoup plus succinctes.

Lire Structures de contrôle en ligne: https://riptutorial.com/fr/common-lisp/topic/3229/structures-de-controle

Chapitre 26: Tables de hachage

Examples

Créer une table de hachage

Les tables de hachage sont créées par `make-hash-table` :

```
(defvar *my-table* (make-hash-table))
```

La fonction peut prendre des paramètres de mot clé pour spécifier davantage le comportement de la table de hachage résultante:

- `test` : Sélectionne la fonction utilisée pour comparer les clés pour l'égalité. Peut-être un désignateur pour l'une des fonctions `eq` , `eql` , `equal` ou `equalp` . La valeur par défaut est `eq` .
- `size` : un indice sur l'implémentation de l'espace éventuellement requis.
- `rehash-size` : Si un entier (> = 1), la table de hachage augmentera sa capacité par le nombre spécifié lors de la reprise. Si sinon un float (> 1.0), la table de hachage augmentera sa capacité au produit de la `rehash-size` et de la capacité précédente.
- `rehash-threshold` : Spécifie le niveau de remplissage de la table de hachage afin de déclencher une rehash.

Itération sur les entrées d'une table de hachage avec maphash

```
(defun print-entry (key value)
  (format t "~A => ~A~%" key value))

(maphash #'print-entry *my-table*) ;; => NIL
```

L'utilisation de `maphash` permet de parcourir les entrées d'une table de hachage. L'ordre d'itération n'est pas spécifié. Le premier argument est une fonction acceptant deux paramètres: la clé et la valeur de l'entrée en cours.

`maphash` renvoie toujours `NIL` .

Itération sur les entrées d'une table de hachage avec boucle

La macro de **boucle** prend en charge l'itération sur les clés, les valeurs ou les clés et les valeurs d'une table de hachage. Les exemples suivants montrent des possibilités, mais la syntaxe en **boucle** complète permet plus de combinaisons et de variantes.

Plus de clés et de valeurs

```
(let ((ht (make-hash-table)))
  (setf (gethash 'a ht) 1
        (gethash 'b ht) 2)
```

```
  (loop for k being each hash-key of ht
     using (hash-value v)
     collect (cons k v)))
;;=> ((A . 1) (B . 2))
```

```
(let ((ht (make-hash-table)))
  (setf (gethash 'a ht) 1
        (gethash 'b ht) 2)
  (loop for v being each hash-value of ht
     using (hash-key k)
     collect (cons k v)))
;;=> ((A . 1) (B . 2))
```

Plus de clés

```
(let ((ht (make-hash-table)))
  (setf (gethash 'a ht) 1
        (gethash 'b ht) 2)
  (loop for k being each hash-key of ht
       collect k))
;;=> (A B)
```

Survaleurs

```
(let ((ht (make-hash-table)))
  (setf (gethash 'a ht) 1
        (gethash 'b ht) 2)
  (loop for v being each hash-value of ht
       collect v))
;;=> (1 2)
```

Itération sur les entrées d'une table de hachage avec un itérateur de table de hachage

Les clés et les valeurs d'une table de hachage peuvent être répétées à l'aide de la macro **avec table de calcul-itérateur** . Cela peut être un peu plus complexe que **maphash** ou **loop** , mais il pourrait être utilisé pour implémenter les constructions d'itération utilisées dans ces méthodes. **with-hash-table-iterator** prend un nom et une table de hachage et lie le nom dans un corps tel que les appels successifs au nom produisent plusieurs valeurs: (i) un booléen indiquant si une valeur est présente; (ii) la clé de l'entrée; et (iii) la valeur de l'entrée.

```
(let ((ht (make-hash-table)))
  (setf (gethash 'a ht) 1
        (gethash 'b ht) 2)
  (with-hash-table-iterator (iterator ht)
    (print (multiple-value-list (iterator)))
    (print (multiple-value-list (iterator)))
    (print (multiple-value-list (iterator)))))

;; (T A 1)
;; (T B 2)
;; (NIL)
```

Lire Tables de hachage en ligne: https://riptutorial.com/fr/common-lisp/topic/4482/tables-de-hachage

Chapitre 27: Tests unitaires

Examples

Utiliser FiveAM

Chargement de la bibliothèque

```
(ql:quickload "fiveam")
```

Définir un cas de test

```
(fiveam:test sum-1
  (fiveam:is (= 3 (+ 1 2))))

;; We'll also add a failing test case
(fiveam:test sum2
  (fiveam:is (= 4 (+ 1 2))))
```

Exécuter des tests

```
(fiveam:run!)
```

qui rapporte

```
Running test suite NIL
 Running test SUM2 f
 Running test SUM1 .
 Did 2 checks.
    Pass: 1 (50%)
    Skip: 0 ( 0%)
    Fail: 1 (50%)
 Failure Details:
 --------------------------------
 SUM2 []:

(+ 1 2)

 evaluated to

3

 which is not

=
```

```
 to

4

..
 --------------------------------
NIL
```

Remarques

- Les tests sont regroupés par suites de test
- Par défaut, les tests sont ajoutés à la suite de tests globale

introduction

Il existe quelques bibliothèques pour les tests unitaires dans Common Lisp

- FiveAM
- Prouvez , avec quelques fonctionnalités uniques comme des rapports de test étendus, des sorties en couleur, un rapport sur la durée des tests et une intégration asdf.
- Lisp-Unit2 , similaire à JUnit
- Fiasco , en mettant l'accent sur la fourniture d'une bonne expérience de test du REPL. Successeur de hu.dwim.stefil

Lire Tests unitaires en ligne: https://riptutorial.com/fr/common-lisp/topic/2349/tests-unitaires

Chapitre 28: Travailler avec des bases de données

Examples

Utilisation simple de PostgreSQL avec Postmodern

Postmodern est une bibliothèque pour interfacer la base de données relationnelle PostgreSQL . Il offre plusieurs niveaux d'accès à PostgreSQL, de l'exécution de requêtes SQL représentées sous forme de chaînes, ou de listes, à un mappage objet-relationnel.

La base de données utilisée dans les exemples suivants peut être créée avec ces instructions SQL:

```
create table employees
  (empid integer not null primary key,
   name text not null,
   birthdate date not null,
   skills text[] not null);
insert into employees (empid, name, birthdate, skills) values
  (1, 'John Orange', '1991-07-26', '{C, Java}'),
  (2, 'Mary Red', '1989-04-14', '{C, Common Lisp, Hunchentoot}'),
  (3, 'Ron Blue', '1974-01-17', '{JavaScript, Common Lisp}'),
  (4, 'Lucy Green', '1968-02-02', '{Java, JavaScript}');
```

Le premier exemple montre le résultat d'une simple requête renvoyant une relation:

```
CL-USER> (ql:quickload "postmodern")   ; load the system postmodern (nickname: pomo)
("postmodern")
CL-USER> (let ((parameters '("database" "dbuser" "dbpass" "localhost")))
           (pomo:with-connection parameters
             (pomo:query "select name, skills from employees")))
(("John Orange" #("C" "Java"))                  ; output manually edited!
 ("Mary Red" #("C" "Common Lisp" "Hunchentoot"))
 ("Ron Blue" #("JavaScript" "Common Lisp"))
 ("Lucy Green" #("Java" "JavaScript")))
4                                               ; the second value is the size of the result
```

Notez que le résultat peut être renvoyé sous la forme d'une liste d'alist ou de plists ajoutant les paramètres facultatifs `:alists` ou `:plists` à la fonction de requête.

Une alternative à la `query` est `doquery` , pour parcourir les résultats d'une requête. Ses paramètres sont `query (&rest names) &body body` , où les noms sont liés aux valeurs de la ligne à chaque itération:

```
CL-USER> (let ((parameters '("database" "dbuser" "dbpass" "localhost")))
           (pomo:with-connection parameters
             (format t "The employees that knows Java are:~%")
             (pomo:doquery "select empid, name from employees where skills @> '{Java}'" (i n)
```

```
               (format t "~a (id = ~a)~%" n i))))
The employees that knows Java are:
John Orange (id = 1)
Lucy Green (id = 4)
NIL
2
```

Lorsque la requête nécessite des paramètres, on peut utiliser des instructions préparées:

```
CL-USER> (let ((parameters '("database" "dbuser" "dbpass" "localhost")))
           (pomo:with-connection parameters
             (funcall
               (pomo:prepare "select name, skills from employees where skills @> $1")
               #("Common Lisp"))))    ; find employees with skills including Common Lisp
(("Mary Red" #("C" "Common Lisp" "Hunchentoot"))
 ("Ron Blue" #("JavaScript" "Common Lisp")))
2
```

La fonction `prepare` reçoit une requête avec des espaces réservés `$1` , `$2` , etc.

En cas de mises à jour, la fonction `exec` renvoie le nombre de tuples modifiés (les deux instructions DDL sont incluses dans une transaction):

```
CL-USER> (let ((parameters '("database" "dbuser" "dbpass" "localhost")))
           (pomo:with-connection parameters
             (pomo:ensure-transaction
               (values
                 (pomo:execute "alter table employees add column salary integer")
                 (pomo:execute "update employees set salary =
                                       case when skills @> '{Common Lisp}'
                                       then 100000 else 50000 end")))))
0
4
```

En plus d'écrire des requêtes SQL sous forme de chaînes, on peut utiliser des listes de mots-clés, de symboles et de constantes, avec une syntaxe rappelant celle de lisp (S-SQL):

```
CL-USER> (let ((parameters '("database" "dbuser" "dbpass" "localhost")))
           (pomo:with-connection parameters
             (pomo:query (:select 'name :from 'employees :where (:> 'salary 60000)))))
(("Mary Red") ("Ron Blue"))
2
```

Lire Travailler avec des bases de données en ligne: https://riptutorial.com/fr/common-lisp/topic/4558/travailler-avec-des-bases-de-donnees

Chapitre 29: Travailler avec SLIME

Examples

Installation

Il est préférable d'utiliser la dernière version de SLIME depuis le dépôt MELPA d'Emacs: les paquets peuvent être un peu instables, mais vous obtenez les dernières fonctionnalités.

Portail et multiplateforme Emacs, Slime, Quicklisp, SBCL et Git

Vous pouvez télécharger une version portable et multiplateforme d'Emacs25 déjà configurée avec Slime, SBCL, Quicklisp et Git: Portacle . C'est un moyen rapide et facile de démarrer. Si vous voulez apprendre à installer tout vous-même, lisez la suite.

Installation manuelle

Dans le fichier d'initialisation GNU Emacs (> = 24.5) (`~/.emacs` ou `~/.emacs.d/init.el`), ajoutez ce qui suit:

```
;; Use Emacs package system
(require 'package)
;; Add MELPA repository
(add-to-list 'package-archives
             '("melpa" . "http://melpa.milkbox.net/packages/") t)
;; Reload package list
(package-initialize)
(unless package-archive-contents
  (package-refresh-contents))
;; List of packages to install:
(setq package-list
      '(magit                    ; git interface (OPTIONAL)
        auto-complete            ; auto complete (RECOMMENDED)
        auto-complete-pcmp       ; programmable completion
        idle-highlight-mode      ; highlight words in programming buffer (OPTIONAL)
        rainbow-delimiters       ; highlight parenthesis (OPTIONAL)
        ac-slime                 ; auto-complete for SLIME
        slime                    ; SLIME itself
        eval-sexp-fu             ; Highlight evaluated form (OPTIONAL)
        smartparens              ; Help with many parentheses (OPTIONAL)
        ))

;; Install if are not installed
(dolist (package package-list)
  (unless (package-installed-p package)
    (package-install package)))

;; Parenthesis - OPTIONAL but recommended
(show-paren-mode t)
(require 'smartparens-config)
(sp-use-paredit-bindings)
(sp-pair "(" ")" :wrap "M-(")
```

```
(define-key smartparens-mode-map (kbd "C-<right>") 'sp-forward-slurp-sexp)
(define-key smartparens-mode-map (kbd "C-<left>") 'sp-backward-slurp-sexp)
(define-key smartparens-mode-map (kbd "C-S-<right>") 'sp-forward-barf-sexp)
(define-key smartparens-mode-map (kbd "C-S-<left>") 'sp-backward-barf-sexp)

(define-key smartparens-mode-map (kbd "C-)") 'sp-forward-slurp-sexp)
(define-key smartparens-mode-map (kbd "C-(") 'sp-backward-slurp-sexp)
(define-key smartparens-mode-map (kbd "C-}") 'sp-forward-barf-sexp)
(define-key smartparens-mode-map (kbd "C-{") 'sp-backward-barf-sexp)

(sp-pair "(" ")" :wrap "M-(")
(sp-pair "[" "]" :wrap "M-[")
(sp-pair "{" "}" :wrap "M-{")

;; MAIN Slime setup
;; Choose lisp implementation:
;; The first option uses roswell with default sbcl
;; the second option - uses ccl directly
(setq slime-lisp-implementations
      '((roswell ("ros" "-L" "sbcl-bin" "run"))
        (ccl ("ccl64"
              "-K" "utf-8"))))
;; Other settings...
```

SLIME seul est correct, mais il fonctionne mieux avec le gestionnaire de paquets Quicklisp . Pour installer Quicklisp, suivez les instructions sur le site Web (si vous utilisez Roswell , suivez les instructions de Roswell). Une fois installé, dans votre lisp invoquer:

```
(ql:quickload :quicklisp-slime-helper)
```

et ajoutez les lignes suivantes au fichier init d'Emacs:

```
;; Find where quicklisp is installed to
;; Add your own location if quicklisp is installed somewhere else
(defvar quicklisp-directories
  '("~/.roswell/lisp/quicklisp/"           ;; default roswell location for quicklisp
    "~/quicklisp/")                        ;; default quicklisp location
  "Possible locations of QUICKLISP")

;; Load slime-helper
(let ((continue-p t)
      (dirs quicklisp-directories))
  (while continue-p
    (cond ((null dirs) (message "Cannot find slime-helper.el"))
          ((file-directory-p (expand-file-name (car dirs)))
           (message "Loading slime-helper.el from %s" (car dirs))
           (load (expand-file-name "slime-helper.el" (car dirs)))
           (setq continue-p nil))
          (t (setq dirs (cdr dirs))))))

;; Autocomplete in SLIME
(require 'slime-autoloads)
(slime-setup '(slime-fancy))

;; (require 'ac-slime)
 (add-hook 'slime-mode-hook 'set-up-slime-ac)
 (add-hook 'slime-repl-mode-hook 'set-up-slime-ac)
 (eval-after-load "auto-complete"
```

```
   '(add-to-list 'ac-modes 'slime-repl-mode))

(eval-after-load "auto-complete"
  '(add-to-list 'ac-modes 'slime-repl-mode))

;; Hooks
(add-hook 'lisp-mode-hook (lambda ()
                            (rainbow-delimiters-mode t)
                            (smartparens-strict-mode t)
                            (idle-highlight-mode t)
                            (auto-complete-mode)))

(add-hook 'slime-mode-hook (lambda ()
                             (set-up-slime-ac)
                             (auto-complete-mode)))

(add-hook 'slime-repl-mode-hook (lambda ()
                                  (rainbow-delimiters-mode t)
                                  (smartparens-strict-mode t)
                                  (set-up-slime-ac)
                                  (auto-complete-mode)))
```

Après le redémarrage, GNU Emacs installera et configurera tous les paquets nécessaires.

Démarrer et terminer SLIME, commandes spéciales (virgule) REPL

Dans Emacs, `Mx slime` démarrera slime avec la première implémentation Common Lisp par défaut. Si plusieurs implémentations sont fournies (via des implémentations variables `slime-lisp-implementations`), d'autres implémentations sont accessibles via `M-- Mx slime` , qui offrira le choix des implémentations disponibles dans le mini-tampon.

`Mx slime` ouvrira le tampon REPL qui ressemblera à ceci:

```
; SLIME 2016-04-19
CL-USER>
```

Le tampon SLIME REPL accepte quelques commandes spéciales. Elles commencent toutes avec `,` . Une fois `,` est tapé, la liste des options sera affiché dans le mini-tampon. Ils comprennent:

- `,quit`
- `,restart-inferior-lisp`
- `,pwd` - imprime le répertoire courant à partir duquel Lisp est exécuté
- `,cd` - va changer le répertoire courant

Utiliser REPL

```
CL-USER> (+ 2 3)
5
CL-USER> (sin 1.5)
0.997495
CL-USER> (mapcar (lambda (x) (+ x 2)) '(1 2 3))
(3 4 5)
```

Le résultat imprimé après l'évaluation n'est pas seulement une chaîne: il y a un objet Lisp complet qui peut être inspecté en cliquant dessus avec le bouton droit de la souris et en choisissant Inspecter.

L'entrée multi-lignes est également possible: utilisez `Cj` pour mettre une nouvelle ligne. `Enter` -key provoquera l'évaluation du formulaire entré et si le formulaire n'est pas terminé, entraînera probablement une erreur:

```
CL-USER> (mapcar (lambda (x y)
                   (declare (ignore y))
                   (* x 2))
                 '(1 2 3)
                 '(:a :b :c))
(2 4 6)
```

La gestion des erreurs

Si l'évaluation provoque une erreur:

```
CL-USER> (/ 3 0)
```

Cela affichera un tampon de débogueur avec le contenu suivant (dans lisp SBCL):

```
arithmetic error DIVISION-BY-ZERO signalled
Operation was /, operands (3 0).
   [Condition of type DIVISION-BY-ZERO]

Restarts:
 0: [RETRY] Retry SLIME REPL evaluation request.
 1: [*ABORT] Return to SLIME's top level.
 2: [ABORT] abort thread (#<THREAD "repl-thread" RUNNING {1004FA8033}>)

Backtrace:
  0: (SB-KERNEL::INTEGER-/-INTEGER 3 0)
  1: (/ 3 0)
  2: (SB-INT:SIMPLE-EVAL-IN-LEXENV (/ 3 0) #<NULL-LEXENV>)
  3: (EVAL (/ 3 0))
  4: (SWANK::EVAL-REGION "(/ 3 0) ..)
  5: ((LAMBDA NIL :IN SWANK-REPL::REPL-EVAL))
--- more ---
```

En déplaçant le curseur vers le bas `--- more ---` passerez `--- more ---` provoquera l'extension de la trace arrière.

À chaque ligne du backtrace, appuyez sur `Enter` pour afficher plus d'informations sur un appel particulier (si disponible).

Appuyez sur `Enter` sur la ligne de redémarrage pour provoquer un redémarrage particulier. Alternativement, le redémarrage peut être choisi par le nombre `0` , `1` ou `2` (appuyez sur la touche correspondante n'importe où dans le tampon). Le redémarrage par défaut est marqué par une étoile et peut être appelé en appuyant sur la touche `q` (pour "quitter"). Appuyez sur `q` pour fermer le

débogueur et afficher les informations suivantes dans REPL

```
; Evaluation aborted on #<DIVISION-BY-ZERO {10064CCE43}>.
CL-USER>
```

Enfin, très rarement, mais Lisp peut rencontrer une erreur qui ne peut pas être traitée par le débogueur Lisp, auquel cas il tombera dans un débogueur de bas niveau ou finira anormalement. Pour voir la cause de ce type d'erreur, passez au tampon `*inferior-lisp*` .

Configuration d'un serveur SWANK sur un tunnel SSH.

1. Installez une implémentation Common Lisp sur le serveur. (Par exemple , `sbcl` , `clisp` , etc ...)
2. Installez `quicklisp` sur le serveur.
3. Charger SWANK avec `(ql:quickload :swank)`
4. Démarrer le serveur avec `(swank:create-server)` . Le port par défaut est `4005` .
5. [Sur votre machine locale] Créez un tunnel SSH avec `ssh -L4005:127.0.0.1:4005 [remote machine]`
6. Connectez-vous au serveur swank distant en cours d'exécution avec `Mx slime-connect` . L'hôte doit être `127.0.0.1` et le port `4005` .

Lire Travailler avec SLIME en ligne: https://riptutorial.com/fr/common-lisp/topic/4097/travailler-avec-slime

Chapitre 30: Types de listes

Examples

Listes simples

Les listes simples sont le type de liste le plus simple dans Common Lisp. Ils sont une séquence ordonnée d'éléments. Ils prennent en charge les opérations de base telles que l'obtention du premier élément d'une liste et le reste d'une liste en temps constant, prennent en charge l'accès aléatoire en temps linéaire.

```
(list 1 2 3)
;=> (1 2 3)

(first (list 1 2 3))
;=> 1

(rest (list 1 2 3))
;=> (2 3)
```

Il existe de nombreuses fonctions qui fonctionnent sur des listes "simples", dans la mesure où elles ne concernent que les éléments de la liste. Celles-ci incluent **find** , **mapcar** et bien d'autres. (Bon nombre de ces fonctions fonctionneront également sur les concepts de séquence 17.1 pour certaines de ces fonctions.

Listes d'association

Les listes simples sont utiles pour représenter une séquence d'éléments, mais il est parfois plus utile de représenter une sorte de clé pour valoriser le mappage. Common Lisp propose plusieurs méthodes pour y parvenir, y compris de véritables tables de hachage (voir 18.1 Concepts de table de hachage). Il existe deux manières principales ou représentant des mappages de valeur dans Common Lisp: les listes de propriétés et les listes d'associations . Cet exemple décrit les listes d'associations.

Une liste d'associations, ou *alist* est une liste "simple" dont les éléments sont des paires en pointillés dans lesquelles la *voiture* de chaque paire est la clé et le *cdr* de chaque paire est la valeur associée. Par exemple,

```
(defparameter *ages* (list (cons 'john 34) (cons 'mary 23) (cons 'tim 72)))
```

peut être considéré comme une liste d'associations qui mappe des symboles indiquant un nom personnel avec un entier indiquant l'âge. Il est possible d'implémenter des fonctions de récupération en utilisant des fonctions de liste simples, comme **member** . Par exemple, pour retrouver l'âge de **John** , on pourrait écrire

```
(cdr (first (member 'mary *age* :key 'car)))
;=> 23
```

La fonction **membre** retourne la queue de la liste en commençant par une cellule cons dont la *voiture* est **Marie,** qui est, **((Mary. 23) (tim. 72)), premier** renvoie le premier élément de la liste, qui est **(Marie. 23)** , et **cdr** renvoie le côté droit de cette paire, qui est **23** . Bien qu'il s'agisse d'un moyen d'accéder à des valeurs dans une liste d'associations, une convention comme les listes d'associations a pour but de s'abstraire de la représentation sous-jacente (une liste) et de fournir des fonctions de plus haut niveau.

Pour les listes d'association, la fonction de récupération est **assoc** , qui prend une clé, une liste d'association et des mots-clés de test facultatifs (clé, test, test-not) et renvoie la paire pour la clé correspondante:

```
(assoc 'tim *ages*)
;=> (tim . 72)
```

Étant donné que le résultat sera toujours une contre-cellule si un élément est présent, si **assoc** renvoie **un** résultat **nul** , l'élément ne figurait pas dans la liste:

```
(assoc 'bob *ages*)
;=> nil
```

Pour mettre à jour les valeurs d'une liste d'associations, **setf** peut être utilisé avec **cdr** . Par exemple, lorsque l'anniversaire de **John** arrive et que son âge augmente, l'une des actions suivantes peut être effectuée:

```
(setf (cdr (assoc 'john *ages*) 35)

(incf (cdr (assoc 'john *ages*)))
```

incf fonctionne dans ce cas car il est basé sur **setf** .

Les listes d'association peuvent également être utilisées comme type de carte bidirectionnelle, car les mappages de clé à valeur doivent être récupérés en fonction de la valeur à l'aide de la fonction d'assoc inversée, **rassoc** .

Dans cet exemple, la liste des associations a été créée en utilisant **list** et **cons** explicitement, mais les listes d'associations peuvent également être créées à l'aide de **pairlis** , qui prend une liste de clés et de données et crée une liste d'associations:

```
(pairlis '(john mary tim) '(23 67 82))
;=> ((john . 23) (mary . 67) (tim . 82))
```

Une seule clé et une paire de valeurs peuvent être ajoutées à une liste d'associations à l'aide d' **acons** :

```
(acons 'john 23 '((mary . 67) (tim . 82)))
;=> ((john . 23) (mary . 67) (tim . 82))
```

La fonction **assoc** parcourt la liste de gauche à droite, ce qui signifie qu'il est possible de

"masquer" des valeurs dans une liste d'association sans les supprimer d'une liste ni mettre à jour la structure de la liste, simplement en ajoutant de nouveaux éléments à la liste. début de la liste. La fonction **acons** est fournie pour cela:

```
(defvar *ages* (pairlis '(john mary tim) '(34 23 72)))

(defvar *new-ages* (acons 'mary 29 *ages*))

*new-ages*
;=> ((mary . 29) (john . 34) (mary . 23) (tim . 72))
```

Et maintenant, une recherche pour **Mary** retournera la première entrée:

```
(assoc 'mary *new-ages*)
;=> 29
```

Listes de propriété

Les listes simples sont utiles pour représenter une séquence d'éléments, mais il est parfois plus utile de représenter une sorte de clé pour valoriser le mappage. Common Lisp propose plusieurs méthodes pour y parvenir, y compris de véritables tables de hachage (voir 18.1 Concepts de table de hachage). Il existe deux manières principales ou représentant des mappages de valeur dans Common Lisp: les listes de propriétés et les listes d'associations . Cet exemple décrit des listes de propriétés.

Une liste de propriétés, ou *plist* , est une liste "simple" dans laquelle les valeurs alternées sont interprétées comme des clés et leurs valeurs associées. Par exemple:

```
(defparameter *ages* (list 'john 34 'mary 23 'tim 72))
```

peut être considéré comme une liste de propriétés qui mappe des symboles indiquant un nom personnel avec un entier indiquant l'âge. Il est possible d'implémenter des fonctions de récupération en utilisant des fonctions de liste simples, comme **member** . Par exemple, pour retrouver l'âge de **John** , on pourrait écrire

```
(second (member 'mary *age*))
;=> 23
```

La fonction **membre** renvoie la fin de la liste en commençant par **mary** , c'est-à-dire **(Mary 23 tim 72)** , et **second** renvoie le deuxième élément de cette liste, à savoir **23** . Bien qu'il s'agisse d'une façon d'accéder aux valeurs dans une liste de propriétés, le but d'une convention comme les listes de propriétés est de s'abstraire de la représentation sous-jacente (une liste) et de fournir des fonctions de plus haut niveau.

Pour les listes de propriétés, la fonction de récupération est **getf** , qui prend la liste de propriétés, une clé (plus communément appelée *indicateur*) et une valeur par défaut facultative à renvoyer si la liste de propriétés ne contient pas de valeur pour la clé.

```
(getf *ages* 'tim)
;=> 72

(getf *ages* 'bob -1)
;=> -1
```

Pour mettre à jour les valeurs dans une liste de propriétés, **setf** peut être utilisé. Par exemple, lorsque l'anniversaire de **John** arrive et que son âge augmente, l'une des actions suivantes peut être effectuée:

```
(setf (getf *ages* 'john) 35)

(incf (getf *ages* 'john))
```

incf fonctionne dans ce cas car il est basé sur **setf** .

Pour rechercher plusieurs propriétés dans une liste de propriétés, utilisez **get-properties** .

La fonction **getf** parcourt la liste de gauche à droite, ce qui signifie qu'il est possible de "masquer" des valeurs dans une liste de propriétés sans les supprimer d'une liste ni mettre à jour la structure de la liste. Par exemple, en utilisant la **liste *** :

```
(defvar *ages* '(john 34 mary 23 tim 72))

(defvar *new-ages* (list* 'mary 29 *ages*))

*new-ages*
;=> (mary 29 john 34 mary 23 tim 72)
```

Et maintenant, une recherche pour **Mary** retournera la première entrée:

```
(getf *new-ages* 'mary)
;=> 29
```

Lire Types de listes en ligne: https://riptutorial.com/fr/common-lisp/topic/3744/types-de-listes

Chapitre 31: Variables lexicales vs spéciales

Examples

Les variables spéciales globales sont spéciales partout

Ainsi, ces variables utiliseront une liaison dynamique.

```
(defparameter count 0)
;; All uses of count will refer to this one

(defun handle-number (number)
  (incf count)
  (format t "~&~d~%" number))

(dotimes (count 4)
  ;; count is shadowed, but still special
  (handle-number count))

(format t "~&Calls: ~d~%" count)
==>
0
2
Calls: 0
```

Attribuez des noms distincts aux variables spéciales pour éviter ce problème:

```
(defparameter *count* 0)

(defun handle-number (number)
  (incf *count*)
  (format t "~&~d~%" number))

(dotimes (count 4)
  (handle-number count))

(format t "~&Calls: ~d~%" *count*)
==>
0
1
2
3
Calls: 4
```

Note 1: il est impossible de rendre une variable globale non spéciale dans une certaine portée. Il n'y a pas de déclaration pour faire une variable *lexicale* .

Note 2: il est possible de déclarer une variable *spéciale* dans un contexte local en utilisant la déclaration `special` . S'il n'y a pas de déclaration spéciale globale pour cette variable, la déclaration est uniquement localisée et peut être ombrée.

```
(defun bar ()
```

```
  (declare (special a))
  a)                        ; value of A is looked up from the dynamic binding

(defun foo ()
  (let ((a 42))             ; <- this variable A is special and
                            ;    dynamically bound
    (declare (special a))
    (list (bar)
          (let ((a 0))      ; <- this variable A is lexical
            (bar)))))


> (foo)
(42 42)
```

Lire Variables lexicales vs spéciales en ligne: https://riptutorial.com/fr/common-lisp/topic/3362/variables-lexicales-vs-speciales

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec Common-Lisp	blambert, Community, CPHPython, Dan Robertson, Ehvince, Gustav Bertram, Inaimathi, JAL, Rainer Joswig, Renzo, Robert Columbia, WarFox
2	ANSI Common Lisp, le standard de langage et sa documentation	Rainer Joswig, sds
3	ASDF - Une autre installation de définition de système	Inaimathi, jkiiski, Joao Tavora, PuercoPop, Rainer Joswig, Sim, Svante
4	Boucles de base	Joshua Taylor, MatthewRock, Rainer Joswig, sadfaf, Svante
5	Citation	MatthewRock, Rainer Joswig, Svante
6	CLOS - le système d'objet Lisp commun	Joshua Taylor, PuercoPop, Rainer Joswig, Sim
7	Contre les cellules et les listes	eyqs, Joshua Taylor, Rainer Joswig, Renzo
8	Correspondance de motif	jkiiski, PuercoPop
9	Créer des fichiers binaires	Inaimathi
10	Egalité et autres prédicats de comparaison	Renzo
11	Expressions régulières	jkiiski, PuercoPop
12	Fonctionne comme des valeurs de première classe	Dan Robertson, Joshua Taylor, PuercoPop, Rainer Joswig, Renzo
13	format	Dan Robertson, Inaimathi, jkiiski, otyn, Renzo
14	Les booléens et les	Rainer Joswig, Renzo, Terje D.

	booléens généralisés	
15	Les fonctions	jkiiski, Rainer Joswig, Svante
16	les macros	JAL, jkiiski, Joshua Taylor, Mark Green, PuercoPop, Rainer Joswig
17	LOOP, une macro Common Lisp pour itération	Dan Robertson, Elias Mårtenson, Inaimathi, PuercoPop, Rainer Joswig, RamenChef, Renzo, Throwaway Account 3 Million
18	Mappage des fonctions sur les listes	Aaron, Rainer Joswig, Renzo
19	Personnalisation	Daniel Kochmański, Rainer Joswig
20	Protocole de méta-objet CLOS	PuercoPop, Rainer Joswig
21	Récursivité	4444, Rainer Joswig, Robert Columbia, sadfaf
22	Regroupement des formulaires	Joshua Taylor, Rainer Joswig
23	Ruisseaux	jkiiski, Rainer Joswig, Renzo, Svante
24	sequence - comment diviser une séquence	PuercoPop, Rainer Joswig, sadfaf
25	Structures de contrôle	eyqs, Rainer Joswig, Robert Columbia, Soupy, Svante, Throwaway Account 3 Million
26	Tables de hachage	Daniel Jour, Joshua Taylor
27	Tests unitaires	Ehvince, PuercoPop, Rainer Joswig, sadfaf
28	Travailler avec des bases de données	Renzo
29	Travailler avec SLIME	Ehvince, mobiuseng, tsikov
30	Types de listes	jkiiski, Joshua Taylor
31	Variables lexicales vs spéciales	Rainer Joswig, Terje D.