EBook Gratis

APRENDIZAJE common-lisp

Free unaffiliated eBook created from
Stack Overflow contributors.

#common-lisp

Tabla de contenido

Acerca de

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.

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

Capítulo 1: Empezando con common-lisp

Observaciones

Esta es una función simple de hello world en Common Lisp. Ejemplos imprimirán el texto `Hello, World!` (sin comillas, seguido de una nueva línea) a la salida estándar.

Common Lisp es un lenguaje de programación que se usa en gran medida de forma interactiva mediante una interfaz conocida como REPL. El REPL (Leer Eval Print Loop) le permite a uno escribir código, hacer que se evalúe (ejecutar) y ver los resultados inmediatamente. `CL-USER>` indica el aviso para el REPL (en cuyo punto se escribe el código a ejecutar). A veces, algo distinto de `CL-USER` aparecerá antes del `>` pero esto sigue siendo un REPL.

Después de la solicitud, aparece algún código, generalmente una sola palabra (es decir, un nombre de variable) o un formulario (una lista de palabras / formas entre `(` y `)`) (es decir, una llamada de función o declaración, etc.). En la siguiente línea habrá cualquier salida que el programa imprima (o nada si el programa no imprime nada) y luego los valores devueltos al evaluar la expresión. Normalmente, una expresión devuelve un valor, pero si devuelve varios valores, aparecen una vez por línea.

Versiones

Versión	Fecha de lanzamiento
Lisp común	1984-01-01
ANSI Common Lisp	1994-01-01

Examples

Hola Mundo

Lo que sigue es un extracto de una sesión REPL con Common Lisp en la que aparece un "¡Hola mundo!" La función está definida y ejecutada. Vea las observaciones al final de esta página para obtener una descripción más detallada de un REPL.

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

Esto define la "función" de cero argumentos llamados `hello` , que escribirán la cadena `"Hello, World!"`

seguido de una nueva línea a la salida estándar, y devuelve `NIL` .

Para definir una función escribimos

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

En este caso, la función se llama `hello` , no toma parámetros y el código que ejecuta es para hacer una llamada a la función. El valor devuelto de una función lisp es el último bit de código en la función que se ejecutará, por lo que `hello` devuelve lo que sea `(format t "Hello, World!~%")` Devuelve.

En lisp para llamar a una función se escribe `(function-name arguments...)` donde `function-name` es el nombre de la función y `arguments...` es la lista de argumentos (separados por espacios) de la llamada. Hay algunos casos especiales que parecen llamadas a funciones pero no están, por ejemplo, en el código anterior no hay una función `defun` que se llama, se trata especialmente y define una función en su lugar.

En el segundo aviso del REPL, después de que hayamos definido la función `hello` , la llamamos sin parámetros escribiendo `(hello)` . Esto, a su vez, llamará a la función de `format` con los parámetros `t` y `"Hello, World!~%"` . La función de `format` produce una salida con formato basada en los argumentos que se le dan (un poco como una versión avanzada de `printf` en C). El primer argumento le dice a dónde enviar, con `t` significa salida estándar. El segundo argumento le dice qué imprimir (y cómo interpretar los parámetros adicionales). La directiva (código especial en el segundo argumento) `~%` le dice al formato que imprima una nueva línea (es decir, en UNIX puede escribir `\n` y en windows `\r\n`). El formato generalmente devuelve `NIL` (un poco como `NULL` en otros idiomas).

Después del segundo aviso, vemos que `Hello, World` se imprimió y en la siguiente línea que el valor devuelto era `NIL` .

Hola nombre

Este es un ejemplo un poco más avanzado que muestra algunas características más de la luz común. Empezamos con un simple `Hello, World!` Funcionar y demostrar algún desarrollo interactivo en el REPL. Tenga en cuenta que cualquier texto de un punto y coma `;` , al resto de la línea hay un comentario.

```
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>
```

Esto resalta algunas de las características avanzadas de la función de `format` de Common Lisp, así como algunas características como parámetros opcionales y argumentos de palabras clave

(por ejemplo, `:number`). Esto también da un ejemplo de desarrollo interactivo en un REPL en lisp común.

El sencillo programa Hello World en REPL.

Common Lisp REPL es un entorno interactivo. Cada formulario escrito después de la solicitud se evalúa, y su valor se imprime después como resultado de la evaluación. Así que el programa más simple posible de "Hello, World!" En Common Lisp es:

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

Lo que sucede aquí es que se da un costo de cadena en la entrada al REPL, se evalúa y se imprime el resultado. Lo que se puede ver en este ejemplo es que las cadenas, como los números, los símbolos especiales como `NIL` y `T` y algunos otros literales, son formas de *autoevaluación* : es decir, se evalúan a sí mismas.

Expresiones basicas

Probemos alguna expresión básica en el 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>
```

El bloque de construcción básico de un programa Common Lisp es la *forma* . En estos ejemplos tenemos *funciones de formas* , es decir, expresiones, escritas como una lista, en las que el primer elemento es un operador (o función) y el resto de los elementos son los operandos (esto se llama "Notación de prefijo" o "Notación polaca" ”). Escribir formularios en el REPL provoca su evaluación. En los ejemplos puede ver expresiones simples cuyos argumentos son números constantes, cadenas y símbolos (en el caso de `'string` , que es el nombre de un tipo). También puede ver que los operadores aritméticos pueden tomar cualquier número de argumentos.

Es importante tener en cuenta que los paréntesis son una parte integral de la sintaxis y no se pueden utilizar libremente como en otros lenguajes de programación. Por ejemplo, el siguiente es un error:

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

En Common Lisp, los formularios también pueden ser datos, símbolos, formularios macro,

formularios especiales y formularios lambda. Se pueden escribir para ser evaluados, devolviendo cero, uno o más valores, o se pueden dar en entrada a una macro, que los transforma en otras formas.

Suma de lista de enteros

```
(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))
```

Expresiones Lambda y Funciones Anónimas

Una función anónima se puede definir sin un nombre a través de una Expresión Lambda . Para definir este tipo de funciones, se utiliza la palabra clave `lambda` lugar de la palabra clave `defun` . Las siguientes líneas son todas equivalentes y definen funciones anónimas que dan como resultado la suma de dos números:

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

Su utilidad es notable al crear formularios Lambda , es decir, un formulario que es una lista donde el primer elemento es la expresión lambda y los elementos restantes son los argumentos de la función anónima. Ejemplos (ejecución en línea):

```
(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)
```

Recursos de aprendizaje comunes de Lisp

Libros en linea

Estos son libros que son de libre acceso en línea.

- Practical Common Lisp de Peter Seibel es una buena introducción a CL para programadores experimentados, que intenta destacar desde el principio lo que hace que CL sea diferente a otros idiomas.
- Common Lisp: una suave introducción a la computación simbólica por David S. Touretzky es una buena introducción para personas nuevas en la programación.
- Common Lisp: se utilizó un enfoque interactivo de Stuart C. Shapiro como libro de texto del curso, y las notas del curso acompañan al libro en el sitio web.
- Common Lisp, el lenguaje de Guy L. Steele es una descripción del lenguaje Common Lisp. Según el CLiki , está desactualizado, pero contiene mejores descripciones de la macro y el

formato del bucle que el Common Lisp Hyperspec.
- En Lisp por Paul Graham es un gran libro para Lispers de experiencia intermedia.
- Let Over Lambda por Doug Hoyte es un libro avanzado sobre macros de Lisp. Varias personas recomendaron que se sienta cómodo con On Lisp antes de leer este libro y que el comienzo sea lento.

Referencias en linea

- Common Lisp Hyperspec es *el* documento de referencia de idioma para Common Lisp.
- El Common Lisp Cookbook es una lista de recetas útiles de Lisp. También contiene una lista de otras fuentes en línea de información de CL.
- Referencia rápida de Common Lisp tiene hojas de referencia de Lisp imprimibles.
- Lispdoc.com busca varias fuentes de información de Lisp (Common Lisp, Lisp exitoso, Lisp, HyperSpec) para obtener documentación.
- L1sp.org es un servicio de redireccionamiento para documentación.

Libros fuera de linea

Estos son libros que probablemente tendrá que comprar o prestar en una biblioteca.

- ANSI Common Lisp por Paul Graham .
- Recetas comunes de Lisp por Edmund Weitz .
- Paradigmas de la Inteligencia Artificial La programación tiene muchas aplicaciones interesantes de Lisp, pero ya no es una buena referencia para la IA.

Comunidades Online

- El CLiki tiene una gran página de inicio . Un gran recurso para todas las cosas CL. Tiene una extensa lista de libros de Lisp .
- Common Lisp subreddit tiene muchos enlaces útiles y documentos de referencia en la barra lateral.
- IRC: #lisp, #ccl, #sbcl y otros en Freenode .
- Common-Lisp.net proporciona hospedaje para muchos proyectos lisp comunes y grupos de usuarios.

Bibliotecas

- Quicklisp es el administrador de bibliotecas para Common Lisp y tiene una larga lista de bibliotecas compatibles .
- Quickdocs alberga la documentación de la biblioteca para muchas bibliotecas CL.
- Awesome CL es una lista curada dirigida por la comunidad de bibliotecas, marcos y otros elementos brillantes ordenados por categoría.

Entornos Lisp pre-envasados

Estos son entornos de edición de Lisp que son fáciles de instalar y comenzar porque todo lo que necesita está preempaquetado y preconfigurado.

- Portacle es un entorno Common Lisp portátil y multiplataforma. Incluye un Emacs

ligeramente personalizado con Slime, SBCL (una popular implementación de Common Lisp), Quicklisp y Git. No necesita instalación, por lo que es una forma muy rápida y fácil de ponerse en marcha.
- Lispbox es un IDE (Emacs + SLIME), un entorno Common Lisp (Clozure Common Lisp) y un administrador de bibliotecas (Quicklisp), preempaquetado como archivos para Windows, Mac OSX y Linux. Descendiente de "Lisp in a Box" recomendado en el libro Practical Common Lisp.
- No está pre-empacado, pero SLIME convierte a Emacs en un IDE de Common Lisp y tiene un manual de usuario para ayudarlo a comenzar. Requiere una implementación separada de Common Lisp.

Implementaciones comunes de Lisp

Esta sección enumera algunas implementaciones de CL comunes y sus manuales. A menos que se indique lo contrario, estas son implementaciones de software libre. Consulte también la lista de aplicaciones gratuitas de Common Lisp de Cliki y la lista de implementaciones comerciales de Common Lisp de Wikipedia .

- Allegro Common Lisp (ACL) y manual . Comercial, pero tiene una edición Express gratuita y videos de entrenamiento en Youtube .
- CLISP y manual .
- Clozure Common Lisp (CCL) y manual .
- Carnegie Mellon University Common Lisp (CMUCL) , tiene un manual y otra página de información útil .
- Embeddable Common Lisp (ECL) y manual .
- LispWorks y manual . Comercial, pero tiene una Edición Personal con algunas limitaciones .
- Banco de acero Common Lisp (SBCL) y manual .
- Scieneer Common Lisp (SCL) y manual es una implementación comercial de Linux y Unix, pero tiene una versión gratuita sin restricciones de evaluación y uso no comercial .

Lea Empezando con common-lisp en línea: https://riptutorial.com/es/common-lisp/topic/534/empezando-con-common-lisp

Capítulo 2: ANSI Common Lisp, el estándar de lenguaje y su documentación.

Examples

Common Lisp HyperSpec

Common Lisp tiene un estándar, que se publicó inicialmente en 1994 como un estándar ANSI.

El Common Lisp HyperSpec , CLHS corto, proporcionado por LispWorks es una documentación HTML de uso frecuente, que se deriva del documento estándar. La HyperSpec también se puede descargar y utilizar localmente .

Los entornos de desarrollo de Common Lisp generalmente permiten la búsqueda de la documentación de HyperSpec para los símbolos de Lisp.

- Para GNU Emacs hay clhs.el.
- SLIME para GNU Emacs proporciona una versión de hyperspec.el .

Ver también: cliki en CLHS

Declaraciones de sintaxis EBNF en la documentación.

El estándar ANSI CL utiliza una notación de sintaxis EBNF extendida. La documentación duplicada en Stackoverflow debe usar la misma notación de sintaxis para reducir la confusión.

Ejemplo:

```
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] )}*] )
```

Notación:

- `[foo]` -> cero o uno `foo`
- `{foo}*` -> cero o más `foo`
- `foo | bar` -> `foo` o `bar`

Common Lisp the Language, 2ª edición, por Guy L. Steele Jr.

Este libro es conocido como CLtL2.

Esta es la segunda edición del libro Common Lisp the Language. Fue publicado en 1990, antes

de que el estándar ANSI CL fuera definitivo. Tomó la definición del idioma original de la primera edición (publicada en 1984) y describió todos los cambios en el proceso de estandarización hasta 1990 más algunas extensiones (como la función de iteración SERIES).

Nota: CLTL2 describe una versión de Common Lisp que es ligeramente diferente del estándar publicado de 1994. Por lo tanto, siempre use el estándar, y no CLtL2, como referencia.

CLtL2 todavía puede ser útil, porque proporciona información que no se encuentra en el documento de especificación Common Lisp.

Hay una versión HTML de Common Lisp the Language, 2ª edición .

CLiki - Propuestas revisiones y aclaraciones de ANSI

En CLiki, una Wiki para Common Lisp y *el* software *gratuito* Common Lisp, se mantiene una lista de Propuestas Revisiones y Aclaraciones ANSI .

Dado que el estándar Common Lisp no ha cambiado desde 1994, los usuarios han encontrado varios problemas con el documento de especificación. Estos están documentados en la página de CLiki.

Referencia rápida de Common Lisp

La Referencia rápida de Common Lisp es un documento que se puede imprimir y encuadernar como un folleto en varios diseños para tener una referencia rápida impresa para Common Lisp.

El estándar ANSI Common Lisp en formato Texinfo (especialmente útil para GNU Emacs)

GNU Emacs utiliza un formato especial para la documentación: *información* .

El estándar Common Lisp se ha convertido al formato Texinfo, que se puede usar para crear documentación navegable con el lector de *información* en GNU Emacs.

Vea aquí: dpans2texi.el convierte las fuentes TeX del borrador del estándar ANSI Common Lisp (dpANS) al formato Texinfo.

Se ha hecho otra versión para GCL: gcl.info.tgz .

Lea ANSI Common Lisp, el estándar de lenguaje y su documentación. en línea: https://riptutorial.com/es/common-lisp/topic/2900/ansi-common-lisp--el-estandar-de-lenguaje-y-su-documentacion-

Capítulo 3: ASDF - Otra facilidad de definición de sistema

Observaciones

ASDF - Otra facilidad de definición de sistema

ASDF es una herramienta para especificar cómo los sistemas del software Common Lisp se componen de componentes (subsistemas y archivos), y cómo operar estos componentes en el orden correcto para que puedan compilarse, cargarse, probarse, etc.

Examples

Sistema ASDF simple con una estructura de directorio plana.

Considere este proyecto simple con una estructura de directorio plana:

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

El archivo `example.asd` es en realidad otro archivo Lisp con poco más que una llamada de función específica de ASDF. Asumiendo que su proyecto depende de los sistemas `drakma` y `clsql` , su contenido puede ser algo como esto:

```
(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"))))
```

Cuando carga este archivo Lisp, le dice a ASDF sobre su `:example` sistema de `:example` , pero todavía no está cargando el sistema en sí. Esto se hace mediante `(asdf:require-system :example)` o `(ql:quickload :example)` .

Y cuando cargue el sistema, ASDF:

1. Cargue las dependencias, en este caso los sistemas ASDF `clsql` y `drakma`
2. *Compile y cargue* los componentes de su sistema, es decir, los archivos Lisp, según las

dependencias dadas

1. `packages` primero (sin dependencias)
2. `functions` después de los `packages` (ya que solo depende de los `packages`), pero antes de `main` (que depende de ellos)
3. `functions` `main` posteriores (ya que depende de `packages` y `functions`)
4. `tools` cualquier momento después de los `packages`

Tenga en cuenta:

- Ingrese las dependencias según sean necesarias (por ejemplo, las definiciones de macro son necesarias antes del uso). Si no lo hace, ASDF producirá un error al cargar su sistema.
- Todos los archivos enumerados terminan en `.lisp` pero este postfix debe eliminarse en el script asdf
- Si su sistema tiene el mismo nombre que su archivo `.asd` , y mueve (o enlace simbólico) su carpeta a `quicklisp/local-projects/` folder, puede cargar el proyecto usando `(ql:quickload "example")` .
- Las bibliotecas de las que depende su sistema deben ser conocidas por ASDF (a través de `ASDF:*CENTRAL-REGISTRY` variable) o Quicklisp (ya sea a través de `QUICKLISP-CLIENT:*LOCAL-PROJECT-DIRECTORIES*` variable o disponible en cualquiera de sus dists)

Cómo definir una operación de prueba para un sistema

```
(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)
```

Notas:

- Estamos asumiendo que el *sistema* : foo-tests define un *paquete* llamado "FOO-TESTS"
- `run-tests` es el punto de entrada para el corredor de prueba
- uoip: symbol-call permite definir un método que llama a una función que aún no se ha leído. El paquete en el que se define la función no existe cuando definimos el sistema

¿En qué paquete debo definir mi sistema ASDF?

ASDF proporciona el paquete `ASDF-USER` para que los desarrolladores definan sus paquetes en.

Lea ASDF - Otra facilidad de definición de sistema en línea: https://riptutorial.com/es/common-lisp/topic/670/asdf---otra-facilidad-de-definicion-de-sistema

Capítulo 4: Booleanos y booleanos generalizados

Examples

Verdadero y falso

El símbolo especial `T` representa el valor *verdadero* en Common Lisp, mientras que el símbolo especial `NIL` representa *falso* :

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

Se denominan "Variables constantes" (sic!) En el estándar, ya que son variables cuyo valor *no se* puede modificar. Como consecuencia, no puede usar sus nombres para variables normales, como en el siguiente ejemplo, incorrecto:

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

En realidad, uno puede considerarlas simplemente como constantes, o como símbolos autoevaluados. `T` y `NIL` son especiales en otros sentidos. Por ejemplo, `T` también es un tipo (el supertipo de cualquier otro tipo), mientras que `NIL` también es la lista vacía:

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

Booleanos generalizados

En realidad, cualquier valor diferente de `NIL` se considera un valor *verdadero* en Common Lisp. Por ejemplo:

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

Este hecho se puede combinar con los operadores booleanos para hacer que los programas sean más concisos. Por ejemplo, el ejemplo anterior es equivalente a:

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

La macro `OR` evalúa sus argumentos en orden de izquierda a derecha y se detiene tan pronto como encuentra un valor no NIL, devolviéndolo. Si todos ellos son `NIL` , el valor devuelto es `NIL` :

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

De manera análoga, la macro `AND` evalúa sus argumentos de izquierda a derecha y devuelve el valor del último, si todos ellos se evalúan como no NIL, de lo contrario, detiene la evaluación tan pronto como encuentra `NIL` , devolviéndolo:

```
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
```

Por estas razones, `AND` y `OR` pueden considerarse más similares a las estructuras de control de otros idiomas, en lugar de a los operadores booleanos.

Lea Booleanos y booleanos generalizados en línea: https://riptutorial.com/es/common-lisp/topic/3292/booleanos-y-booleanos-generalizados

Capítulo 5: Bucles basicos

Sintaxis

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

Examples

dotimes

`dotimes` es una macro para la iteración de enteros sobre una sola variable desde 0 debajo de algún valor de parámetro. Uno de los ejemplos simples sería:

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

0
1
2
3
4
NIL
```

Tenga en cuenta que `NIL` es el valor devuelto, ya que nosotros no proporcionamos uno; la variable comienza desde 0 y en todo el bucle se convierte en valores de 0 a N-1. Después del bucle, la variable se convierte en la 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)
```

lista de tareas

`dolist` es una macro de bucle creada para recorrer fácilmente las listas. Uno de los usos más simples sería:

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

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

Tenga en cuenta que dado que no proporcionamos un valor de retorno, se devuelve `NIL` (y A, B, C, D se imprimen en `*standard-output*`).

`dolist` también puede devolver valores:

```
;;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
```

Bucle simple

La macro de **bucle** tiene dos formas: la forma "simple" y la forma "extendida". La forma extendida se trata en otro tema de documentación, pero el bucle simple es útil para bucles muy básicos.

La forma de **bucle** simple toma varias formas y las repite hasta que se sale del bucle usando **return** o alguna otra salida (por ejemplo, **throw**).

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

0
1
2
3
4
NIL
```

Lea Bucles basicos en línea: https://riptutorial.com/es/common-lisp/topic/2053/bucles-basicos

Capítulo 6: Citar

Sintaxis

- (citar objeto) -> objeto

Observaciones

Hay algunos objetos (por ejemplo, símbolos de palabras clave) que no necesitan ser citados ya que se evalúan a sí mismos.

Examples

Ejemplo de cita simple

Quote es un **operador especial** que evita la evaluación de su argumento. Devuelve su argumento, sin evaluar.

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

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

'es un alias para el operador especial QUOTE

La `'thing` notación es igual a `(quote thing)` .

El *lector* hará la expansión:

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

Las citas se utilizan para evitar una evaluación posterior. El objeto citado se evalúa a sí mismo.

```
> 'a
A

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

Si los objetos citados son modificados destructivamente, las consecuencias son indefinidas!

Evite operaciones destructivas en objetos citados. Los objetos citados son objetos literales.

Posiblemente están incrustados en el código de alguna manera. La forma en que funciona y los efectos de las modificaciones no se especifican en el estándar Common Lisp, pero puede tener consecuencias no deseadas como modificar los datos compartidos, intentar modificar los datos protegidos contra escritura o crear efectos secundarios no deseados.

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

Cotización y autoevaluación de objetos.

Tenga en cuenta que muchos tipos de datos no necesitan ser citados, ya que se evalúan a sí mismos. `QUOTE` es especialmente útil para los símbolos y las listas, para evitar la evaluación como formularios de Lisp.

Ejemplo para otros tipos de datos que no es necesario citar para evitar la evaluación: cadenas, números, caracteres, objetos CLOS, ...

Aquí un ejemplo para cuerdas. Los resultados de la evaluación son cadenas, ya sean citados en la fuente o no.

```
> (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")
```

Citar por lo tanto los objetos es opcional.

Lea Citar en línea: https://riptutorial.com/es/common-lisp/topic/1315/citar

Capítulo 7: CLOS - el sistema de objetos Common Lisp

Examples

Creando una clase básica de CLOS sin padres

Una clase CLOS es descrita por:

- un nombre
- una lista de superclases
- una lista de ranuras
- otras opciones como documentación

Cada ranura tiene:

- un nombre
- un formulario de inicialización (opcional)
- un argumento de inicialización (opcional)
- un tipo (opcional)
- una cadena de documentación (opcional)
- Funciones de acceso, lector y / o escritor (opcional)
- otras opciones como la asignación

Ejemplo:

```
(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"))
```

Un método de impresión predeterminado:

```
(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))))
```

Creando instancias:

```
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 e interfaces

Common Lisp no tiene interfaces en el sentido en que lo hacen algunos idiomas (por ejemplo, Java), y hay menos necesidad de ese tipo de interfaz, dado que Common Lisp admite funciones de herencia múltiple y genéricas. Sin embargo, el mismo tipo de patrones se puede realizar fácilmente usando clases de mezcla. Este ejemplo muestra la especificación de una interfaz de colección con varias funciones genéricas correspondientes.

```
;; 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)))
```

Una implementación de la interfaz es solo una clase que tiene la mezcla como una de sus súper clases y definiciones de las funciones genéricas apropiadas. (En este punto, observe que la clase mixin es realmente solo para indicar la intención de que la clase implemente la "interfaz". Este ejemplo funcionaría igual de bien con algunas funciones genéricas y documentación que indica que hay métodos en la función para la clase.)

```
;; 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))))
```

Finalmente, podemos ver cómo se ve una instancia de la clase de **conjuntos ordenados** cuando se usan las funciones de "interfaz":

```
(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)
```

Lea CLOS - el sistema de objetos Common Lisp en línea: https://riptutorial.com/es/common-lisp/topic/673/clos---el-sistema-de-objetos-common-lisp

Capítulo 8: Contras celdas y listas

Examples

Listas como convención

Algunos idiomas incluyen una estructura de datos de lista. Common Lisp, y otros idiomas de la familia Lisp, hacen un uso extensivo de las listas (y el nombre Lisp se basa en la idea de un procesador LISt). Sin embargo, Common Lisp no incluye realmente un tipo de datos de lista primitiva. En cambio, las listas existen por convención. La convención depende de dos principios:

1. El símbolo **nil** es la lista vacía.
2. Una lista no vacía es una *celda de contras* cuyo *automóvil* es el primer elemento de la lista y cuyo *CDR* es el resto de la lista.

Eso es todo lo que hay en las listas. Si has leído el ejemplo llamado *¿Qué es una celda de contras?* , entonces sabes que una celda de contras cuyo coche es X y cuyo CDR es Y puede escribirse como **(X. Y)** . Eso significa que podemos escribir algunas listas basadas en los principios anteriores. La lista de los elementos 1, 2 y 3 es simplemente:

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

Sin embargo, debido a que las listas son tan comunes en la familia de lenguajes Lisp, existen convenciones de impresión especiales que van más allá de la simple notación de par de puntos para las celdas de contras.

1. El símbolo **nil** también se puede escribir como **()** .
2. Cuando el cdr de una celda de contras es otra lista (ya sea **()** o una celda de contras), en lugar de escribir la celda de una cons con la notación de par de puntos, se utiliza la "notación de lista".

La notación de la lista se muestra más claramente mediante varios ejemplos:

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

La idea es que los elementos de la lista se escriban en orden sucesivo entre paréntesis hasta que se alcance el cdr final de la lista. Si el cdr final es **nulo** (la lista vacía), entonces se escribe el paréntesis final. Si el cdr final no es **nulo** (en cuyo caso la lista se denomina *lista impropia*), se escribe un punto y luego se escribe el cdr final.

¿Qué es una célula contras?

Una celda de contras, también conocida como un par de puntos (debido a su representación

impresa), es simplemente un par de dos objetos. Una función de contras es creada por la función `cons` , y los elementos del par se extraen utilizando las funciones `car` y `cdr` .

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

Por ejemplo, esto devuelve un par cuyo primer elemento (que puede extraerse con un `car`) es `"a"` , y cuyo segundo elemento (que puede extraerse con `cdr`) es `4` .

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

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

Contras celdas se pueden imprimir en notación de *par de puntos* :

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

Las celdas de contras también se pueden leer en notación de par de puntos, para que

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

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

(La forma impresa de las celdas contras también puede ser un poco más complicada. Para más información, consulte el ejemplo sobre celdas cons en forma de listas).

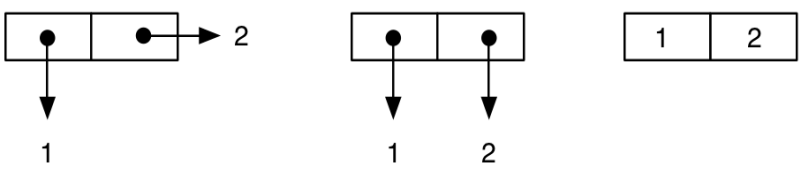
Eso es; contras celdas son solo pares de elementos creados por la función `cons` , y los elementos pueden extraerse con `car` y `cdr` . Debido a su simplicidad, las celdas contras pueden ser un bloque de construcción útil para estructuras de datos más complejas.

Dibujando células de contras

Para comprender mejor la semántica de las recomendaciones y listas, a menudo se utiliza una representación gráfica de este tipo de estructuras. Una celda de contras generalmente se representa con dos cuadros en contacto, que contienen dos flechas que apuntan a los valores de `car` y `cdr` , o directamente los valores. Por ejemplo, el resultado de:

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

Se puede representar con uno de estos dibujos:

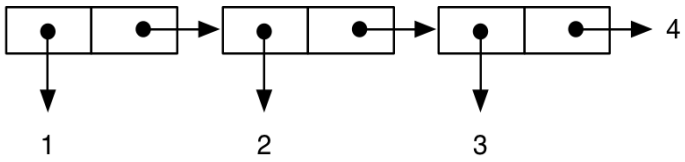


Tenga en cuenta que estas representaciones son puramente conceptuales y no denotan el hecho de que los valores están *contenidos* en la celda, o están *apuntados* desde la celda: en general, esto depende de la implementación, el tipo de los valores, el nivel de optimización, etc. En el resto del ejemplo usaremos el primer tipo de dibujo, que es el que se usa más comúnmente.

Entonces, por ejemplo:

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

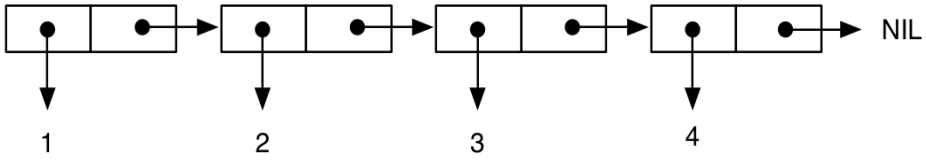
se representa como:



mientras:

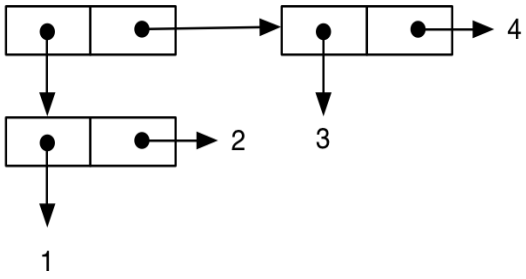
```
(cons 1 (cons 2 (cons 3 (cons 4 nil))))  ;; proper list, equivalent to: (list 1 2 3 4)
;; -> (1 2 3 4)
```

se representa como:



Aquí hay una estructura en forma de árbol:

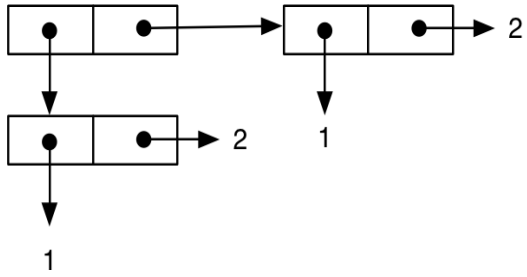
```
(cons (cons 1 2) (cons 3 4))
;; -> ((1 . 2) 3 . 4)         ; note the printing as an improper list
```



El último ejemplo muestra cómo esta notación puede ayudarnos a comprender aspectos semánticos importantes del lenguaje. Primero, escribimos una expresión similar a la anterior:

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

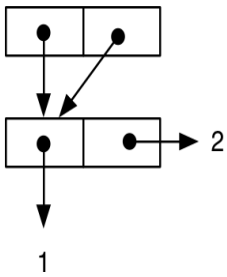
que se puede representar de la manera habitual como:



Luego, escribimos una expresión diferente, que aparentemente es equivalente a la anterior, y esto parece confirmado por la representación impresa del resultado:

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

Pero, si dibujamos el diagrama, podemos ver que la semántica de la expresión es diferente, ya que la *misma* celda es el valor *tanto* de la parte del `car` como de la parte `cdr` de los `cons` exteriores (esto es, la `cell-a` es *compartida*) :



y el hecho de que la semántica de los dos resultados es realmente diferente a nivel de idioma se puede verificar mediante las siguientes pruebas:

```
(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)
```

El primer `eq` es *falso,* ya que `car` y `cdr` de `c1` son estructuralmente iguales (es *cierto* por `equal`), pero no son "idénticos" (es decir, "la misma estructura compartida"), mientras que en la segunda prueba el resultado es *verdadero* ya que `car` y `cdr` de `c2` son *idénticos* , es decir, son *la misma estructura* .

Lea Contras celdas y listas en línea: https://riptutorial.com/es/common-lisp/topic/2622/contras-celdas-y-listas

Capítulo 9: Corrientes

Sintaxis

- `(read-char &optional stream eof-error-p eof-value recursive-p)` => carácter
- `(write-char character &optional stream)` => carácter
- `(read-line &optional stream eof-error-p eof-value recursive-p)` => line, missing-newline-p
- `(write-line line &optional stream)` => línea

Parámetros

Parámetro	Detalle
`stream`	La secuencia para leer o escribir.
`eof-error-p`	Si se señala un error si se encuentra el final del archivo.
`eof-value`	Qué valor debe devolverse si se encuentra `eof-error-p` , y `eof-error-p` es falso.
`recursive-p`	Es la operación de lectura llamada recursivamente desde `READ` . Por lo general, esto debe dejarse como `NIL` .
`character`	El carácter a escribir, o el carácter que se leyó.
`line`	La línea a escribir, o la línea que se leyó.

Examples

Creando flujos de entrada desde cadenas

La macro `WITH-INPUT-FROM-STRING` se puede utilizar para crear una secuencia a partir de una cadena.

```
(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
```

Lo mismo se puede hacer manualmente usando `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)))
```

Escritura de salida a una cadena

La macro `WITH-OUTPUT-TO-STRING` se puede usar para crear una secuencia de salida de cadena y devolver la cadena resultante al final.

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

Lo mismo se puede hacer manualmente utilizando `MAKE-STRING-OUTPUT-STREAM` y `GET-OUTPUT-STREAM-STRING` .

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

Arroyos grises

Las transmisiones en gris son una extensión no estándar que permite transmisiones definidas por el usuario. Proporciona clases y métodos que el usuario puede ampliar. Debería consultar el manual de implementación para ver si proporciona flujos de Gray.

Para un ejemplo simple, una secuencia de entrada de caracteres que devuelve caracteres aleatorios podría implementarse así:

```
(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
```

Archivo de lectura

Se puede abrir un archivo para leerlo como un flujo usando 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
```

Lo mismo se puede hacer manualmente usando `OPEN` y `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)))
```

Tenga en cuenta que `READ-LINE` crea una nueva cadena para cada línea. Esto puede ser lento. Algunas implementaciones proporcionan una variante, que puede leer una línea en un búfer de cadena. Ejemplo: `READ-LINE-INTO` para Allegro CL.

Escribiendo en un archivo

Se puede abrir un archivo para escribirlo como una secuencia usando 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)))
```

Lo mismo se puede hacer manualmente con `OPEN` y `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))
```

Copiando un archivo

Copiar byte por byte de un archivo

La siguiente función copia un archivo en otro realizando una copia exacta de byte por byte, ignorando el tipo de contenido (que puede ser cualquiera de las líneas de caracteres en algunos datos de codificación o binarios):

```
(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))))))
```

El tipo `(unsigned-byte 8)` es el tipo de bytes de 8 bits. Las funciones `read-byte` y `write-byte` funcionan en bytes, en lugar de `read-char` y `write-char` que funcionan en caracteres. `read-byte` devuelve un byte leído desde el flujo, o `NIL` al final del archivo si el segundo parámetro opcional es `NIL` (de lo contrario, indica un error).

Copia masiva

Una copia exacta, más eficiente la anterior. Se puede hacer leyendo y escribiendo los archivos con grandes porciones de datos cada vez, en lugar de bytes individuales:

```
(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` `write-sequence` se utilizan aquí con un búfer que es un vector de bytes (pueden operar en secuencias de bytes o caracteres). `read-sequence` llena la matriz con los bytes leídos cada vez, y devuelve los números de bytes leídos (que puede ser menor que el tamaño de la matriz cuando se llega al final del archivo). Tenga en cuenta que la matriz se modifica destructivamente en cada iteración.

Copia exacta línea por línea de un archivo

El ejemplo final es una copia realizada leyendo cada línea de caracteres del archivo de entrada y escribiéndola en el archivo de salida. Tenga en cuenta que, dado que queremos una copia exacta, debemos verificar si la última línea del archivo de entrada está terminada o no por uno o más caracteres de final de línea. Por esta razón, usamos los dos valores devueltos por `read-line` : una nueva cadena que contiene los caracteres de la siguiente línea y un valor booleano que es *verdadero* si la línea es la última del archivo y no contiene el carácter de nueva línea final (s). En este caso `write-string` se utiliza `write-string` lugar de `write-line` , ya que el primero no agrega una nueva línea al final de la línea.

```
(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)))))))))
```

Tenga en cuenta que este programa es independiente de la plataforma, ya que los caracteres de nueva línea (que varían en diferentes sistemas operativos) se administran automáticamente mediante las funciones de `write-line` `read-line` `write-line` .

Leyendo y escribiendo archivos enteros en y desde cadenas

La siguiente función lee un archivo completo en una nueva cadena y lo devuelve:

```
(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))))
```

El resultado es `NIL` si el archivo no existe.

La siguiente función escribe una cadena en un archivo. Un parámetro de palabra clave se usa para especificar qué hacer si el archivo ya existe (de manera predeterminada causa un error, los valores admisibles son los 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)))
```

En este caso `write-sequence` puede sustituirse `write-string` .

Lea Corrientes en línea: https://riptutorial.com/es/common-lisp/topic/3028/corrientes

Capítulo 10: Creando Binarios

Examples

Edificio buildapp

Los binarios independientes de Common Lisp se pueden construir con `buildapp` . Antes de que podamos usarlo para generar binarios, necesitamos instalarlo y compilarlo.

La forma más fácil que conozco es usar `quicklisp` y Common Lisp (este ejemplo usa [`sbcl`], pero no debería marcar la diferencia cuál tiene).

```
$ 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
$
```

Una vez que haya creado ese binario, puede usarlo para construir binarios de sus programas Common Lisp. Si pretende hacerlo mucho, probablemente también debería ponerlo en algún lugar de su `PATH` para que pueda ejecutarlo con `buildapp` desde cualquier directorio.

Buildapp Hello World

El binario más simple posible que podrías construir

1. No tiene dependencias
2. No toma argumentos de línea de comando
3. Sólo escribe "¡Hola mundo!" al `stdout`

Después de que hayas construido `buildapp` , puedes simplemente ...

```
$ 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 Hello Web World

Un ejemplo más realista implica un proyecto que está construyendo con varios archivos en el disco (en lugar de una opción `--eval` pasada a `buildapp`), y algunas dependencias para `buildapp` .

Debido a que pueden suceder cosas arbitrarias durante la búsqueda y carga de sistemas `asdf` (incluida la carga de otros sistemas potencialmente no relacionados), no es suficiente con solo inspeccionar los archivos `asd` de los proyectos en los que depende para averiguar qué necesita cargar . El enfoque general es utilizar `quicklisp` para cargar el sistema de destino, luego llamar a `ql:write-asdf-manifest-file` para escribir un manifiesto completo de todo lo que está cargado.

Aquí hay un sistema de juguete construido con `hunchentoot` para ilustrar cómo podría suceder eso en la práctica:

```
;;;; 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
```

Una vez que haya guardado esos archivos en un directorio llamado `buildapp-hello-web-world` , puede hacerlo

```
$ 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
```

Esto produce un binario que hace exactamente lo que crees que debería, dado lo anterior.

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

Entonces deberías poder disparar otro shell, hacer `curl localhost:4242` y ver la respuesta en texto sin formato de `Hello Web World!` imprimirse

Lea Creando Binarios en línea: https://riptutorial.com/es/common-lisp/topic/5457/creando-binarios

Capítulo 11: Estructuras de Control

Examples

Construcciones condicionales

En Common Lisp, `if` es la construcción condicional más simple. Tiene la forma `(if test then [else])` y se evalúa a `then` si la `test` es verdadera y de `else` contrario. La otra parte puede ser omitida.

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

Una diferencia muy importante entre `if` en Common Lisp y `if` en muchos otros lenguajes de programación es que CL de `if` es una expresión, no una afirmación. Como tal, `if` formularios devuelven valores, que pueden asignarse a variables, usarse en listas de argumentos, etc.

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

Common Lisp 's `if` se puede considerar equivalente al operador ternario?: En C # y otros lenguajes de "corsé".

Por ejemplo, la siguiente expresión de C #:

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

Es equivalente al siguiente código Common Lisp, asumiendo que ese `year` contiene un número entero:

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

`cond` es otro constructo condicional. Es algo similar a una cadena de sentencias `if` , y tiene la forma:

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

Más precisamente, `cond` tiene cero o más *cláusulas* , y cada cláusula tiene una prueba seguida de cero o más consecuentes. La construcción completa de `cond` selecciona la primera cláusula cuya

prueba no evalúa a `nil` y evalúa sus consecuentes en orden. Devuelve el valor del último formulario en los consecuentes.

```
(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!"
```

Para proporcionar una cláusula predeterminada para evaluar si ninguna otra cláusula se evalúa como `t` , puede agregar una cláusula que sea verdadera de manera predeterminada utilizando `t` . Este concepto es muy similar al de `CASE...ELSE` de SQL `CASE...ELSE` , pero utiliza un verdadero booleano literal en lugar de una palabra clave para realizar la tarea.

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

Una construcción `if` puede escribirse como una construcción `cond` . `(if test then else)` y `(cond (test then) (t else))` son equivalentes.

Si solo necesita una cláusula, use `when` o a `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
```

El bucle do

La mayoría de las construcciones condicionales y de bucle en Common Lisp son en realidad macros que ocultan construcciones más básicas. Por ejemplo, `dotimes` y `dolist` se basan en la macro `do` . La forma para `do` ve así:

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

- `varlist` se compone de las variables definidas en el bucle, sus valores iniciales y cómo cambian después de cada iteración. La parte de 'cambio' se evalúa al final del bucle.

- `endlist` contiene las condiciones finales y los valores devueltos al final del bucle. La condición final se evalúa al comienzo del bucle.

Aquí hay uno que comienza en 0 y sube hasta (no incluye) 10.

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

Y aquí hay uno que se mueve a través de una lista:

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

La parte `varlist` es similar a la de una sentencia `let` . Puede enlazar más de una variable, y solo existen dentro del bucle. Cada variable declarada está en su propio conjunto de paréntesis. Aquí hay uno que cuenta cuántos 1 y 2 están en una lista.

```
(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)
```

Y si una macro loop de while no ha sido implementada:

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

Para las aplicaciones más comunes, las macros `dotimes` y `doloop` más específicas son mucho más sucintas.

Lea Estructuras de Control en línea: https://riptutorial.com/es/common-lisp/topic/3229/estructuras-de-control

Capítulo 12: Examen de la unidad

Examples

Usando FiveAM

Cargando la biblioteca

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

Definir un caso de prueba.

```
(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))))
```

Ejecutar pruebas

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

que informa

```
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
```

Notas

- Las pruebas están agrupadas por suites de prueba.
- Por defecto, las pruebas se agregan al conjunto de pruebas global.

Introducción

Hay algunas bibliotecas para pruebas de unidad en Common Lisp

- Cincoam
- Pruebe , con algunas características únicas, como extensos reportes de prueba, salida en color, informe de duración de la prueba e integración asdf.
- Lisp-Unit2 , similar a JUnit
- Fiasco , centrándose en proporcionar una buena experiencia de prueba de la REPL. Sucesor de hu.dwim.stefil

Lea Examen de la unidad en línea: https://riptutorial.com/es/common-lisp/topic/2349/examen-de-la-unidad

Capítulo 13: Expresiones regulares

Examples

Usando con la coincidencia de patrones para unir grupos capturados

La trivia de la biblioteca de coincidencia de patrones proporciona un sistema `trivia.ppcre` que permite `trivia.ppcre` grupos capturados mediante la coincidencia de patrones

```
(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")
```

- Nota: la biblioteca Optima proporciona una facilidad similar en el sistema `optima.ppcre`

Encuadernación de grupos de registro con CL-PPCRE.

`CL-PPCRE:REGISTER-GROUPS-BIND` hará coincidir una cadena con una expresión regular, y si coincide, vinculará los grupos de registros en la expresión regular a las variables. Si la cadena no coincide, se devuelve `NIL` .

```
(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
```

Lea Expresiones regulares en línea: https://riptutorial.com/es/common-lisp/topic/2897/expresiones-regulares

Capítulo 14: Formas de agrupación

Examples

¿Cuándo es necesaria la agrupación?

En algunos lugares de Common Lisp, una serie de formularios se evalúan en orden. Por ejemplo, en el cuerpo de un **defun** o **lambda** , o el cuerpo de un **dotimes** . En esos casos, escribir múltiples formularios en orden funciona como se espera. Sin embargo, en algunos lugares, como las partes *then* y *else* de las expresiones **if** , solo se permite una única forma. Por supuesto, uno puede querer evaluar múltiples expresiones en esos lugares. Para esas situaciones, se necesita algún tipo de forma implícita de agrupación explícita.

Progreso

El operador especial de propósito general **progn** se utiliza para la evaluación de cero o más formas. Se devuelve el valor del último formulario. Por ejemplo, en el siguiente, se evalúa **(print 'hola)** (y su resultado es ignorado), y luego se evalúa **42** y **(42)** se devuelve su resultado:

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

Si no hay formularios dentro del **programa** , se devuelve **nil** :

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

Además de agrupar una serie de formularios, **progn** también tiene la importante propiedad de que si el formulario **progn** es un *formulario de nivel superior* , todos los formularios dentro de él se procesan como formularios de nivel superior. Esto puede ser importante cuando se escriben macros que se expanden en múltiples formularios y todos deben procesarse como formularios de nivel superior.

Progn también es valioso ya que devuelve *todos* los valores del último formulario. Por ejemplo,

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

En contraste, algunas expresiones de agrupación solo devuelven el *valor primario* de la forma que produce el resultado.

Avances implícitos

Algunas formas usan programas *implícitos* para describir su comportamiento. Por ejemplo, el **cuándo** y **menos** macros, que son esencialmente de un solo **lado, si** formas, describen su comportamiento en términos de un *progn implícita.* Esto significa que una forma como

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

se evalúa y la condición **(foo-p foo)** es verdadera, entonces *form1* y *form2* se agrupan como si estuvieran contenidos dentro de un **progn** . La expansión de la macro **cuando** es esencialmente:

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

Prog1 y Prog2

Muchas veces, es útil evaluar múltiples expresiones y devolver el resultado de la primera o segunda forma en lugar de la última. Esto es fácil de lograr usando **let** y, por ejemplo:

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

Debido a que esta forma es común en algunas aplicaciones, Common Lisp incluye **prog1** y **prog2** que son como **progn** , pero devuelven el resultado de la primera y la segunda forma, respectivamente. Por ejemplo:

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

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

Sin embargo, una distinción importante entre **prog1** / **prog2** y **progn** es que **progn** devuelve *todos* los valores de la última forma, mientras que **prog1** y **prog2** solo devuelven el valor primario de la primera y la segunda forma. Por ejemplo:

```
(progn
```

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

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

Para valores múltiples con **una** evaluación de estilo **prog1** , use **multiples valor-prog1** en **su** lugar. No existe un **prog2 de valor múltiple** similar, pero no es difícil de implementar si lo necesita.

Bloquear

El especial operador **bloque** permite agrupación de varias formas Lisp (como un implícito `progn`) y también toma un *nombre* para nombrar el bloque. Cuando se evalúan los formularios dentro del bloque, el operador especial de **devolución** puede usarse para abandonar el bloque. Por ejemplo:

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

return-from también se puede proporcionar con un valor de retorno:

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

Los bloques con nombre son útiles cuando una porción de código tiene un nombre significativo, o cuando los bloques están anidados. En algún contexto, solo la capacidad de regresar de un bloque temprano es importante. En ese caso, puede usar **nil** como nombre de bloque y **volver** . **La devolución** es igual que la **devolución** , excepto que el nombre del bloque siempre es **nulo** .

Nota: los formularios adjuntos no son formularios de nivel superior. Eso es diferente de `progn` , donde las formas adjuntas de una forma de `progn` nivel `progn` todavía se consideran formas de *nivel superior* .

Tagbody

Para una gran cantidad de control en los formularios de un grupo, el operador especial de **tagbody** puede ser muy útil. Los formularios dentro de un formulario de **tagbody** son *etiquetas go* (que son solo símbolos o enteros) o formularios para ejecutar. Dentro de un **tagbody** , el operador especial **go** se utiliza para transferir la ejecución a una nueva ubicación. Este tipo de programación puede considerarse de nivel bastante bajo, ya que permite rutas de ejecución

arbitrarias. El siguiente es un ejemplo detallado de cómo se vería un for-loop cuando se implementa como un **cuerpo de etiqueta** :

```
(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))
```

Mientras que **Tagbody** and **Go** no se usa comúnmente, tal vez debido a que "GOTO se considera dañino", pero puede ser útil al implementar estructuras de control complejas como las máquinas de estado. Muchas construcciones de iteración también se expanden en un *cuerpo de etiqueta implícito* . Por ejemplo, el cuerpo de un **punto** se especifica como una serie de etiquetas y formas.

¿Qué forma utilizar?

Cuando se escriben macros que se expanden en formas que podrían involucrar agrupación, vale la pena dedicar un tiempo a considerar en qué agrupación se expandirá la construcción.

Para las formas de estilo definición, por ejemplo, un **define-widget de** macro que generalmente aparecerá como un formulario de nivel superior, y que varios **defun** s, **defstruct** s, etc., por lo general tiene sentido utilizar un **progn,** de modo que los formularios secundarios son Procesados como formularios de nivel superior. Para las formas de iteración, un **tagbody** implícito es más común.

Por ejemplo, el cuerpo de **dotimes** , **dolist** , y **hacer** cada expanden en un **tagbody** implícita.

Para los formularios que definen una "parte" del código, un **bloque** implícito suele ser útil. Por ejemplo, mientras que el cuerpo de un **defun** está dentro de un **progn** implícito, ese **progn** implícito está dentro de un bloque que comparte el nombre de la función. Eso significa que se puede utilizar **return-from** para salir de la función. Tal comp

Lea Formas de agrupación en línea: https://riptutorial.com/es/common-lisp/topic/4892/formas-de-agrupacion

Capítulo 15: formato

Parámetros

Lambda-List	`(format DESTINATION CONTROL-STRING &REST FORMAT-ARGUMENTS)`
`DESTINATION`	la cosa para escribir. Puede ser una secuencia de salida, `t` (abreviatura para `*standard-output*`), o `nil` (que crea una cadena para escribir)
`CONTROL-STRING`	la cadena de la plantilla. Puede ser una cadena primitiva o puede contener directivas de comandos con prefijo de tilde que especifiquen y de alguna manera transformen argumentos adicionales.
`FORMAT-ARGUMENTS`	posibles argumentos adicionales requeridos por el `CONTROL-STRING` dado.

Observaciones

La documentación de CLHS para las directivas `FORMAT` se puede encontrar en la Sección 22.3 . Con SLIME, puede escribir `Cc Cd ~` para buscar la documentación de CLHS para una directiva de formato específica.

Examples

Uso básico y directivas simples

Los primeros dos argumentos para dar formato son una secuencia de salida y una cadena de control. El uso básico no requiere argumentos adicionales. Al pasar `t` como la secuencia se escribe en `*standard-output*` .

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

Esa expresión escribirá el `Basic Message` en la salida estándar y devolverá `nil` .

Al pasar `nil` como la secuencia crea una nueva cadena y la devuelve.

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

La mayoría de las directivas de cadenas de control requieren argumentos adicionales. La directiva `~a` ("estética") imprimirá cualquier argumento como por el procedimiento de `princ` . Esto imprime el formulario sin caracteres de escape (las palabras clave se imprimen sin los dos puntos iniciales,

las cadenas sin las comillas circundantes, 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` opcionalmente derecha o izquierda de pads basada en entradas adicionales.

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

El `~s` directiva es como `~a` , pero imprime caracteres de escape.

```
> (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\""
```

Iterando sobre una lista

Uno puede iterar sobre una lista usando las directivas `~{` y `~}` .

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

`~^` puede usarse para escapar si no quedan más elementos.

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

Se puede dar un argumento numérico a `~{` para limitar cuántas iteraciones se pueden hacer:

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

`~@{` iterará sobre los argumentos restantes, en lugar de una lista:

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

Las sublistas pueden ser iteradas usando `~:{` :

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

Expresiones condicionales

Las expresiones condicionales se pueden hacer con `~[` y `~]` . Las cláusulas de la expresión se separan usando `~;` .

De forma predeterminada, `~[` toma un número entero de la lista de argumentos y selecciona la cláusula correspondiente. Las cláusulas comienzan en cero.

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

La última cláusula se puede separar con `~:;` en su lugar para que sea la cláusula else.

```
(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 la expresión condicional comienza con `~:[` , esperará un booleano generalizado en lugar de un entero. Solo puede tener dos cláusulas; el primero se imprime si el valor booleano es `NIL` , y la segunda cláusula si es veraz.

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

Si la expresión condicional comienza con `~@[` , solo debería haber una cláusula, que se imprime si la entrada, un booleano generalizado, era veraz. El booleano no se consumirá si es veraz.

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

Lea formato en línea: https://riptutorial.com/es/common-lisp/topic/687/formato

Capítulo 16: Funciones

Observaciones

Se pueden crear funciones anónimas usando `LAMBDA` . Las funciones locales se pueden definir utilizando `LABELS` o `FLET` . Sus parámetros se definen igual que en las funciones nombradas globales.

Examples

Parámetros requeridos

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

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

Parámetros opcionales

Los parámetros opcionales se pueden especificar después de los parámetros requeridos, utilizando la palabra clave `&OPTIONAL` . Puede haber múltiples parámetros opcionales después de ello.

```
(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
```

Alor por defecto

Se puede dar un valor predeterminado para los parámetros opcionales especificando el parámetro con una lista; El segundo valor es el predeterminado. La forma del valor predeterminado solo se evaluará si se proporcionó el argumento, por lo que se puede usar para

efectos secundarios, como señalar un error.

```
(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
```

Compruebe si el argumento opcional fue dado

Se puede agregar un tercer miembro a la lista después del valor predeterminado; un nombre de variable que es verdadero si se proporcionó el argumento, o `NIL` si no se dio (y se usa el valor predeterminado).

```
(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
```

Función sin parámetros

Las funciones nombradas globales se definen con `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
```

El cuerpo de la función puede contener cualquier número de formas. Los valores del último formulario serán devueltos desde la función.

Parámetro de descanso

Se puede dar un único parámetro de descanso con la palabra clave `&REST` después de los argumentos requeridos. Si tal parámetro existe, la función puede tomar varios argumentos, que se agruparán en una lista en el parámetro resto. Tenga en cuenta que la variable `CALL-ARGUMENTS-LIMIT` determina el número máximo de argumentos que se pueden usar en una llamada de función, por lo tanto, el número de argumentos se limita a un valor específico de implementación de un mínimo de 50 o más argumentos.

```
(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
```

Resto y parámetros de palabras clave juntos

El parámetro resto puede estar antes de los parámetros de palabras clave. En ese caso contendrá la lista de propiedades dada por el usuario. Los valores de palabra clave seguirán vinculados al parámetro de palabra clave correspondiente.

```
(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
```

Las palabras clave `&ALLOW-OTHER-KEYS` se pueden agregar al final de la lista lambda para permitir al usuario dar argumentos de palabras clave no definidos como parámetros. Ellos irán en la lista de descanso.

```
(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 auxiliares

La palabra clave `&AUX` se puede usar para definir variables locales para la función. No son parámetros; El usuario no puede suministrarlos.

`&AUX` variables `&AUX` son raramente usadas. Siempre puede utilizar `LET` lugar, o alguna otra forma de definir variables locales en el cuerpo de la función.

`&AUX` variables `&AUX` tienen las ventajas de que las variables locales de todo el cuerpo de la función se mueven a la parte superior y hacen que un nivel de sangría (por ejemplo, introducido por un LET) sea innecesario.

```
(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
```

Un uso típico puede ser resolver los parámetros de "designador". De nuevo, no necesitas hacerlo de esta manera; Usar `let` es igual de idiomático.

```
(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, salir de un bloque o una función

Las funciones siempre establecen un bloque alrededor del cuerpo. Este bloque tiene el mismo nombre que el nombre de la función. Esto significa que puede usar `RETURN-FROM` con este nombre de bloque para regresar de la función y devolver los valores.

Debe evitar regresar temprano siempre que sea posible.

```
(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"
```

Parámetros de palabras clave

Los parámetros de palabras clave se pueden definir con la palabra clave `&KEY` . Siempre son opcionales (consulte el ejemplo de Parámetros opcionales para obtener detalles de la definición). Puede haber múltiples parámetros de palabras clave.

```
(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
```

Lea Funciones en línea: https://riptutorial.com/es/common-lisp/topic/2126/funciones

Capítulo 17: Funciones como valores de primera clase.

Sintaxis

- (nombre de la función) ; recupera la función objeto de ese nombre
- #'nombre ; azúcar sintáctica para (nombre de la función)
- (símbolo de función de símbolo); devuelve la función vinculada al símbolo
- (función funcs args ...); llamar a la función con args
- (aplicar la función arglista); Función de llamada con argumentos dados en una lista.
- (aplicar la función arg1 arg2 ... argn arglist); Función de llamada con argumentos dados por arg1, arg2, ..., argn y el resto en la lista arglist

Parámetros

Parámetro	Detalles
nombre	algún símbolo (no evaluado) que nombra una función
símbolo	un símbolo
función	una función que se va a llamar
args ...	cero o más argumentos (*no* una lista de argumentos)
arglista	una lista que contiene argumentos para pasar a una función
arg1, arg2, ..., argn	Cada uno es un único argumento que se pasa a una función.

Observaciones

Cuando se habla de lenguajes similares a Lisp, existe una distinción común entre lo que se conoce como Lisp-1 y Lisp-2. En un Lisp-1, los símbolos solo tienen un valor y si un símbolo se refiere a una función, entonces el valor de ese símbolo será esa función. En un Lisp-2, los símbolos pueden tener valores y funciones asociadas por separado, por lo que se requiere una forma especial para referirse a la función almacenada en un símbolo en lugar del valor.

Common Lisp es básicamente un Lisp-2, sin embargo, en realidad hay más de 2 espacios de nombres (cosas a las que los símbolos pueden referirse): los símbolos pueden referirse a valores, funciones, tipos y etiquetas, por ejemplo.

Examples

Definiendo funciones anónimas.

Las funciones en Common Lisp son *valores de primera clase* . Se puede crear una función anónima usando `lambda` . Por ejemplo, aquí hay una función de 3 argumentos que luego llamamos usando `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
```

Las funciones anónimas también se pueden utilizar directamente. Common Lisp proporciona una sintaxis para ello.

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

Las funciones anónimas también se pueden almacenar como funciones 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
```

Las expresiones lambda citadas no son funciones

Tenga en cuenta que las expresiones lambda citadas no son funciones en Common Lisp. Esto **no** funciona:

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

Para convertir una expresión lambda entrecomillada a una función, use `coerce` , `eval` o `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>
```

Haciendo referencia a las funciones existentes

Cualquier símbolo en Common Lisp tiene una ranura para una variable a vincular y una ranura separada para una función a vincular.

Tenga en cuenta que la denominación en este ejemplo es solo para ilustración. Las variables globales no deberían llamarse `foo` , sino `*foo*` . La última notación es una convención para dejar en claro que la variable es una variable *especial* que utiliza *el enlace dinámico* .

```
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
```

Funciones de orden superior

Common Lisp contiene muchas funciones de orden superior que son funciones pasadas para los argumentos y las llaman. Tal vez los más fundamentales sean `funcall` y de `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)
```

Hay muchas otras funciones de orden superior que, por ejemplo, aplican una función muchas veces a los elementos de una lista.

```
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)
```

Sumando una lista

La función de **reducción** se puede utilizar para sumar los elementos de una lista.

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

Por defecto, **reducir** realiza una reducción *asociativa a la izquierda* , lo que significa que la suma 10 se calcula como

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

Los primeros dos elementos se suman primero, y luego ese resultado (3) se agrega al siguiente elemento (3) para producir 6, que a su vez se agrega a 4, para producir el resultado final.

Esto es más seguro que usar **aplicar** (por ejemplo, en **(aplicar '+' (1 2 3 4))** porque la longitud de la lista de argumentos que se puede pasar a **aplicar** es limitada (consulte el **límite de argumentos de llamadas**), y la **reducción** funcionará Con funciones que solo toman dos argumentos.

Al especificar el argumento de la palabra clave **desde el final** , **reducir** procesará la lista en la otra dirección, lo que significa que la suma se computa en el orden inverso. Es decir

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

está computando

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

Implementación de reversa y reventa.

Common Lisp ya tiene una función **inversa** , pero si no lo tenía, entonces podría implementarse fácilmente usando **reducir** . Dada una lista como

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

la lista invertida es

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

Puede que no sea un uso obvio de **reducir** , pero si tenemos una función de "contras invertida", digamos **xcons** , de modo que

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

Entonces

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

Que es una reducción.

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

Common Lisp tiene otra función útil, **revappend** , que es una combinación de **reversa** y **anexa** . Conceptualmente, invierte una lista y la agrega a alguna cola:

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

Esto también se puede implementar con **reducir** . De hecho, es lo mismo que la implementación del **reverso** anterior, excepto que el valor inicial debería ser **(4 5 6) en** lugar de la lista vacía.

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

Cierres

Las funciones recuerdan el ámbito léxico en el que se definieron. Debido a esto, podemos incluir un lambda en un let para definir los cierres.

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

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

En el ejemplo anterior, la variable de contador solo es accesible a la función anónima. Esto se ve más claramente en el siguiente ejemplo.

```
(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
```

Definiendo funciones que toman funciones y devuelven funciones.

Un ejemplo 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
```

El ejemplo clásico de la composición de la función : $(f \circ g \circ h) (x) = f (g (h (x))$:

```
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)))
```

Lea Funciones como valores de primera clase. en línea: https://riptutorial.com/es/common-lisp/topic/1259/funciones-como-valores-de-primera-clase-

Capítulo 18: Funciones de mapeo sobre listas

Examples

Visión general

Un conjunto de funciones de mapeo de alto nivel está disponible en Common Lisp, para aplicar una función a los elementos de una o más listas. Se diferencian en la forma en que se aplica la función a las listas y en cómo se obtiene el resultado final. La siguiente tabla resume las diferencias y muestra para cada una de ellas el formulario de LOOP equivalente. *f* es la función que debe aplicarse, que debe tener un número de argumentos igual al número de listas; "Aplicado al auto" significa que se aplica a su vez a los elementos de las listas, "aplicado a cdr" significa que se aplica a su vez a las listas, su cdr, su cddr, etc .; la columna "devoluciones" muestra si el resultado global es el obtenido al enumerar los resultados, concatenarlos (¡por lo tanto deben ser listas!), o simplemente usarse para efectos secundarios (y en este caso se devuelve la primera lista).

Función	Aplicado a	Devoluciones	Bucle equivalente
(mapcar fl $_1$ ... l $_n$)	coche	lista de resultados	(bucle para x $_1$ en l $_1$ … para x $_n$ en l $_n$ collect (fx $_1$ … x $_n$))
(maplist fl $_1$ ... l $_n$)	cdr	lista de resultados	(bucle para x $_1$ en l $_1$ … para x $_n$ en l $_n$ collect (fx $_1$ … x $_n$))
(mapcan fl $_1$ ... l $_n$)	coche	concatenación de resultados	(bucle para x $_1$ en l $_1$ ... para x $_n$ en l $_n$ nconc (fx $_1$ ... x $_n$))
(mapcon fl $_1$ ... l $_n$)	cdr	concatenación de resultados	(bucle para x $_1$ en l $_1$ … para x $_n$ en l $_n$ nconc (fx $_1$ … x $_n$))
(mapc fl $_1$ … l $_n$)	coche	l $_1$	(bucle para x $_1$ en l $_1$ ... para x $_n$ en l $_n$ do (fx $_1$ ... x $_n$) finalmente (retorno l $_1$))
(mapl fl $_1$ ... l $_n$)	cdr	l $_1$	(bucle para x $_1$ en l $_1$ ... para x $_n$ en l $_n$ do (fx $_1$ ... x $_n$) finalmente (retorno l $_1$))

Tenga en cuenta que, en todos los casos, las listas pueden ser de diferentes longitudes, y la aplicación termina cuando se termina la lista más corta.

Hay otras dos funciones de mapas disponibles: `map` , que se puede aplicar a secuencias (cadenas, vectores, listas), análogo a `mapcar` , y que puede devolver cualquier tipo de secuencia, especificado como primer argumento, y `map-into` , análogo a `map` , pero que modifica destructivamente su primer argumento de secuencia para mantener los resultados de la aplicación de la función.

Ejemplos de MAPCAR

MAPCAR es la función más utilizada de la familia:

```
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)
```

Un uso idiomático de `mapcar` es transponer una matriz representada como una lista de listas:

```
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 |
;  +---+---+---+               +---+---+---+
```

Para una explicación, vea esta respuesta .

Ejemplos 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))
```

Ejemplos de MAPCAN y 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)
```

Uno de los usos de MAPCAN es crear una lista de resultados sin valores 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)
```

Ejemplos de MAPC y 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)
```

Lea Funciones de mapeo sobre listas en línea: https://riptutorial.com/es/common-lisp/topic/6064/funciones-de-mapeo-sobre-listas

Capítulo 19: Igualdad y otros predicados de comparación.

Examples

La diferencia entre EQ y EQL.

1. `EQ` verifica si dos valores tienen la misma dirección de memoria: en otras palabras, verifica si los dos valores son en realidad el *mismo* objeto *idéntico* . Por lo tanto, se puede considerar la prueba de identidad y se debe aplicar *solo* a estructuras: conses, arrays, estructuras, objetos, generalmente para ver si se trata de hecho con el mismo objeto "alcanzado" a través de diferentes caminos, o alias a través de diferentes variables

2. `EQL` verifica si dos estructuras son el mismo objeto (como `EQ`) *o* si son los mismos valores no estructurados (es decir, los mismos valores numéricos para números del mismo tipo o los valores de caracteres). Dado que incluye el operador `EQ` y se puede usar también en valores no estructurados, es el operador más importante y más utilizado, y casi todas las funciones primitivas que requieren una comparación de igualdad, como `MEMBER` , *usan este operador de forma predeterminada* .

Entonces, siempre es cierto que `(EQ XY)` implica `(EQL XY)` , mientras que viceversa no se cumple.

Algunos ejemplos pueden aclarar la diferencia entre los dos operadores:

```
(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
```

Probemos los mismos ejemplos con `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
```

A partir de los ejemplos que podemos ver por qué la `EQL` operador debe utilizar para comprobar de forma portátil para la “identidad” para todos los valores, estructurado y no estructurado, y por qué en realidad muchos expertos desaconsejan el uso de `EQ` en general.

Igualdad estructural con EQUAL, EQUALP, TREE-EQUAL

Estos tres operadores implementan la equivalencia estructural, es decir, comprueban si diferentes objetos complejos tienen una estructura equivalente con un componente equivalente.

`EQUAL` comporta como `EQL` para datos no estructurados, mientras que para las estructuras construidas por pares (listas y árboles), y los dos tipos especiales de matrices, cadenas y vectores de bits, realiza *una equivalencia estructural* , volviéndose verdadera en dos estructuras que son isomorfas y cuyos Los componentes elementales son igualmente iguales por `EQUAL` . Por ejemplo:

```
(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` devuelve verdadero en todos los casos en que `EQUAL` es verdadero, pero también usa equivalencia estructural para arreglos de cualquier tipo y dimensión, para estructuras y para tablas hash (¡pero no para instancias de clase!). Por otra parte, utiliza equivalencia insensible a mayúsculas y minúsculas para cadenas.

```
(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
```

Finalmente, `TREE-EQUAL` se puede aplicar a las estructuras creadas a través de `cons` y verifica si son isomorfas, como `EQUAL` , pero dejando al usuario la opción de qué función usar para comparar las hojas, es decir, los no contras (átomo) encontrados. que puede ser de cualquier otro tipo de datos (de forma predeterminada, la prueba utilizada en el átomo es `EQL`). Por ejemplo:

```
(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
```

Operadores de comparación en valores numéricos

Los valores numéricos pueden compararse con `=` y los otros operadores de comparación numéricos (`/=` , `<` , `<=` , `>` , `>=`) que ignoran la diferencia en la representación física de los diferentes tipos de números, y realizan la comparación de los valores matemáticos correspondientes . Por ejemplo:

```
(= 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
```

A partir de estos ejemplos, podemos concluir que `=` es el operador que normalmente se debe usar para realizar la comparación entre valores numéricos, a menos que queramos ser estrictos en el hecho de que dos valores numéricos son iguales *solo* si tienen el mismo tipo numérico, en En qué caso se debe utilizar `EQL` .

Operadores de comparación en caracteres y cadenas.

Common Lisp tiene 12 operadores de tipo específico para comparar dos caracteres, 6 de ellos sensibles a las mayúsculas y los otros no. Sus nombres tienen un patrón simple para hacer fácil recordar su significado:

Distingue mayúsculas y minúsculas	Caso Insensible
CHAR =	CHAR-IGUAL
CHAR / =	CHAR-NO-IGUAL
CHAR <	CHAR-LESSP
CHAR <=	CHAR-NOT-GREATERP
CHAR>	CHAR-GREATERP
CHAR> =	CHAR-NOT-LESSP

Dos caracteres del mismo caso están en el mismo orden que los códigos correspondientes obtenidos por `CHAR-CODE` , mientras que para comparaciones no sensibles al caso, el orden relativo entre dos caracteres cualquiera de los dos rangos `a..z` , `A..Z` depende de la implementación . Ejemplos:

```
(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
```

Para cadenas, los operadores específicos son `STRING=` , `STRING-EQUAL` , etc. con la palabra STRING en lugar de CHAR. Dos cadenas son iguales si tienen el mismo número de caracteres *y* los caracteres correspondientes son iguales según `CHAR=` o `CHAR-EQUAL` si la prueba distingue entre

mayúsculas y minúsculas.

El orden entre las cadenas es el orden lexicográfico de los caracteres de las dos cadenas. Cuando una comparación de ordenación tiene éxito, el resultado no es `T` , pero el índice del primer carácter en el que difieren las dos cadenas (lo que equivale a verdadero, ya que cada objeto no NIL es un "booleano generalizado" en Common Lisp).

Una cosa importante es que *todos* los operadores de comparación en la cadena aceptan cuatro parámetros de palabras clave: `start1` , `end1` , `start2` , `end2` , que pueden usarse para restringir la comparación a solo una serie de caracteres contiguos dentro de una o ambas cadenas. El índice de inicio si se omite es 0, el índice final se omite es igual a la longitud de la cadena, y la comparación se realiza en la subcadena comenzando en el carácter con índice `:start` y termina con el carácter con índice `:end - 1` incluido.

Finalmente, tenga en cuenta que una cadena, incluso con un solo carácter, no se puede comparar con un carácter.

Ejemplos:

```
(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
```

Como un caso especial, los operadores de comparación de cadenas también se pueden aplicar a los símbolos, y la comparación se realiza en el `SYMBOL-NAME` del símbolo. Por ejemplo:

```
(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
```

Como nota final, `EQL` en los caracteres es equivalente a `CHAR=` ; `EQUAL` en cadenas es equivalente a `STRING=` , mientras que `EQUALP` en cadenas es equivalente a `STRING-EQUAL` .

Overwiew

En Common Lisp hay muchos predicados diferentes para comparar valores. Se pueden clasificar en las siguientes categorías:

1. Operadores de igualdad genéricos: EQ, EQL, EQUAL, EQUALP. Se pueden usar para valores de cualquier tipo y devolver siempre un valor booleano T o NIL.

2. Operadores de igualdad específicos de tipo: = y = para números, CHAR = CHAR = CHAR-EQUAL CHAR-NOT-EQUAL para los caracteres, STRING = STRING = STRING-IGUAL STRING-NOT-IGUAL para las cadenas, TREE-IGUAL para las cuentas.
3. Operadores de comparación para valores numéricos: <, <=,>,> =. Se pueden aplicar a cualquier tipo de número y comparar el valor matemático del número, independientemente del tipo real.
4. Operadores de comparación para caracteres, como CHAR <, CHAR-LESSP, etc., que comparan caracteres de una manera sensible a las mayúsculas y minúsculas o de una manera no sensible a las mayúsculas y minúsculas, de acuerdo con un orden de implementación que preserva el orden alfabético natural.
5. Operadores de comparación para cadenas, como STRING <, STRING-LESSP, etc., que comparan cadenas por lexicografía, ya sea de forma sensible a las mayúsculas y minúsculas, mediante el uso de operadores de comparación de caracteres.

Lea Igualdad y otros predicados de comparación. en línea: https://riptutorial.com/es/common-lisp/topic/10064/igualdad-y-otros-predicados-de-comparacion-

Capítulo 20: La coincidencia de patrones

Examples

Visión general

Las dos bibliotecas principales que proporcionan la coincidencia de patrones en Common Lisp son Optima y Trivia . Ambos proporcionan una API y una sintaxis similares. Sin embargo, trivia proporciona una interfaz unificada para extender la coincidencia, `defpattern` .

Despachando solicitudes de Clack

Debido a que una solicitud de clack se representa como una lista, podemos usar la coincidencia de patrones como punto de entrada a la aplicación de clack como una forma de enrutar la solicitud a sus controladores apropiados

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

Nota: Para iniciar `*app*` lo pasamos a clackup. ej `(clack:clackup *app*)`

defun-match

El uso de la coincidencia de patrones permite vincular la definición de la función y la coincidencia de patrones, similar a 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
```

Patrones de constructor

Las celdas, estructuras, vectores, listas y demás pueden combinarse con patrones de constructor.

```
(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
```

Patrón de guardia

Los patrones de guardia se pueden usar para verificar que un valor satisfaga una forma de prueba arbitraria.

```
(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!
```

Lea La coincidencia de patrones en línea: https://riptutorial.com/es/common-lisp/topic/2933/la-coincidencia-de-patrones

Capítulo 21: Léxico vs variables especiales

Examples

Las variables especiales globales son especiales en todas partes

Por lo tanto estas variables utilizarán enlace dinámico.

```
(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
```

Asigne nombres especiales a las variables especiales para evitar este problema:

```
(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
```

Nota 1: no es posible hacer una variable global no especial en un determinado ámbito. No hay declaración para hacer una variable *léxica* .

Nota 2: es posible declarar una variable *especial* en un contexto local utilizando la declaración `special` . Si no hay una declaración especial global para esa variable, la declaración es solo local y puede ser sombreada.

```
(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)
```

Lea Léxico vs variables especiales en línea: https://riptutorial.com/es/common-lisp/topic/3362/lexico-vs-variables-especiales

Capítulo 22: LOOP, una macro de Common Lisp para iteración

Examples

Bucles limitados

Podemos repetir una acción varias veces usando 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)
```

Bucle sobre secuencias

```
(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
```

Aquí hay un resumen de las palabras clave.

Palabra clave	Tipo de secuencia	Tipo variable
en	lista	elemento de la lista

Palabra clave	Tipo de secuencia	Tipo variable
en	lista	algunos cdr de lista
a través de	vector	elemento de vector

Bucle sobre tablas de hash

```
(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))
```

Formulario de bucle simple

Formulario de LOOP simple sin palabras clave especiales:

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

Para salir del bucle podemos usar `(return <return value>)` `

Algunos ejemplos:

```
(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!~%")))))
```

Bucle sobre paquetes

```
(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))
```

Bucles aritméticos

```
(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
```

Desestructuración en declaraciones FOR

Podemos destruir listas de objetos compuestos.

```
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)
```

También podemos desestructurar una lista en sí.

```
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)
```

Esto es útil cuando queremos recorrer solo ciertos elementos

```
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)
```

Usando `NIL` para ignorar un término:

```
(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
```

LOOP como expresión

A diferencia de los bucles en casi todos los demás lenguajes de programación en uso hoy en día, el `LOOP` en Common Lisp se puede usar como una expresión:

```
(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` hace que el `LOOP` devuelva el valor más grande que se evaluó. `MINIMIZE` es lo opuesto a `MAXIMIZE` .

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

`COUNT` te dice cuántas veces una expresión se evaluó como no `NIL` durante el ciclo:

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

`LOOP` también tiene equivalentes de las funciones de `some` , `every` y no a `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))
```

... excepto que no se limitan a iterar sobre secuencias:

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

Los verbos generadores de valor `LOOP` también se pueden escribir con un sufijo -ing:

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

También es posible capturar el valor generado por estos verbos en variables (creadas implícitamente por la macro `LOOP`), para que pueda generar más de un valor a la vez:

```
 (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)))
```

Puede tener más de una cláusula de `collect` , `count` , etc. que se acumule en el mismo valor de salida. Serán ejecutados en secuencia.

Lo siguiente convierte una lista de asociaciones (que puede usar con `assoc`) en una lista de propiedades (que puede usar con `getf`):

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

Aunque este es mejor estilo:

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

Condicionalmente ejecutando cláusulas LOOP.

`LOOP` tiene su propia declaración `IF` que puede controlar cómo se ejecutan las cláusulas:

```
(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 combinación de múltiples cláusulas en un cuerpo de IF requiere una sintaxis especial:

```
 (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)))
```

Iteración paralela

Se permiten múltiples cláusulas `FOR` en un `LOOP` . El bucle termina cuando la primera de estas cláusulas termina:

```
(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))
```

Otras cláusulas que determinan si el bucle debe continuar pueden combinarse:

```
(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)
```

Determine qué lista es más larga, eliminando la iteración tan pronto como se conozca la respuesta:

```
(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)))
```

Numerar los elementos de una lista:

```
(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))
```

Asegúrese de que todos los números en una lista sean pares, pero solo para los primeros 100 elementos:

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

Iteración anidada

La sintaxis especial de `LOOP NAMED foo` te permite crear un bucle desde el que puedes salir antes. La salida se realiza utilizando `return-from` , y se puede usar desde dentro de bucles anidados.

Lo siguiente utiliza un bucle anidado para buscar un número complejo en una matriz 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))))
```

Cláusula de RETORNO versus formulario de RETORNO.

Dentro de un `LOOP` , puede utilizar el formulario Common Lisp `(return)` en cualquier expresión, lo que hará que el formulario `LOOP` evalúe inmediatamente el valor dado para `return` .

`LOOP` también tiene una cláusula de `return` que funciona de manera casi idéntica, y la única diferencia es que no la rodeas con paréntesis. La cláusula se usa dentro del DSL de `LOOP` , mientras que la forma se usa dentro de las expresiones.

```
(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 cosa después de `finally` debe ser una expresión, por lo que se debe usar el formulario `(return)` y no la cláusula 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.
```

Bucle sobre una ventana de una lista

Algunos ejemplos para una ventana de tamaño 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
```

Lea LOOP, una macro de Common Lisp para iteración en línea:
https://riptutorial.com/es/common-lisp/topic/1369/loop--una-macro-de-common-lisp-para-iteracion

Capítulo 23: macros

Observaciones

El propósito de las macros

Las macros están destinadas a generar código, transformar código y proporcionar nuevas notaciones. Estas nuevas notaciones pueden ser más adecuadas para expresar mejor el programa, por ejemplo, proporcionando construcciones a nivel de dominio o nuevos lenguajes incorporados.

Las macros pueden hacer que el código fuente sea más autoexplicativo, pero la depuración puede ser más difícil. Como regla general, uno no debe usar macros cuando una función normal funcionará. Cuando los use, evite los escollos habituales, trate de atenerse a los patrones de uso común y las convenciones de nombres.

Orden de Macroexpansion

En comparación con las funciones, las macros se expanden en orden inverso; Primero lo primero, lo último lo último. Esto significa que, de forma predeterminada, no se puede usar una macro interna para generar la sintaxis necesaria para una macro externa.

Orden de evaluación

A veces las macros necesitan mover los formularios proporcionados por el usuario. Uno debe asegurarse de no cambiar el orden en que se evalúan. El usuario puede estar confiando en los efectos secundarios que suceden en orden.

Evaluar una sola vez

La expansión de una macro a menudo necesita usar el valor del mismo formulario proporcionado por el usuario más de una vez. Es posible que la forma tenga efectos secundarios o que esté llamando a una función costosa. Por lo tanto, la macro debe asegurarse de evaluar solo tales formas una vez. Por lo general, esto se hará asignando el valor a una variable local (cuyo nombre es `GENSYM` ed).

Funciones utilizadas por las macros, utilizando EVAL-WHEN

Las macros complejas a menudo tienen partes de su lógica implementada en funciones separadas. Sin embargo, hay que recordar que las macros se expanden antes de compilar el código real. Al compilar un archivo, entonces, por defecto, las funciones y variables definidas en el mismo archivo no estarán disponibles durante la ejecución de la macro. Todas las definiciones de funciones y variables, en el mismo archivo, utilizadas por una macro deben estar envueltas dentro de una forma `EVAL-WHEN` . `EVAL-WHEN` debe tener las tres veces especificadas, cuando el código adjunto también debe evaluarse durante la carga y el tiempo de ejecución.

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

Esto no se aplica a las funciones llamadas desde la expansión de la macro, solo a las llamadas por la macro misma.

Examples

Patrones de macro comunes

TODO: Tal vez mover las explicaciones a los comentarios y agregar ejemplos por separado

FOOF

En Common Lisp, hay un concepto de referencias generalizadas . Permiten a un programador establecer valores en varios "lugares" como si fueran variables. Las macros que hacen uso de esta habilidad a menudo tienen un prefijo `F` en el nombre. El lugar suele ser el primer argumento de la macro.

Ejemplos de la norma: `INCF` , `DECF` , `ROTATEF` , `SHIFTF` , `REMF` .

Un ejemplo tonto, una macro que voltea el signo de un almacén de números en un lugar:

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

CON FOO

Macros que bloquean y liberan de forma segura un recurso generalmente se nombran con un `WITH-` -prefix. La macro usualmente debe usar una sintaxis como:

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

Ejemplos de la norma: `WITH-OPEN-FILE` , `WITH-OPEN-STREAM` , `WITH-INPUT-FROM-STRING` , `WITH-OUTPUT-TO-STRING` .

Un enfoque para implementar este tipo de macro que puede evitar algunos de los inconvenientes de la contaminación de nombres y la evaluación múltiple no deseada es implementar primero una versión funcional. Por ejemplo, el primer paso en la implementación de una macro `with-widget` que crea de forma segura un widget y luego se limpia puede ser una función:

```
(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
```

Debido a que esta es una función, no hay preocupaciones sobre el alcance de los nombres dentro de la **función** o el **proveedor** , y facilita la escritura de la macro correspondiente:

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

DO-FOO

Las macros que se repiten sobre algo a menudo se nombran con un prefijo `DO` . La sintaxis de las macros debería estar normalmente en forma.

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

Ejemplos de la norma: `DOTIMES` , `DOLIST` , `DO-SYMBOLS` .

FOOCASE, EFOOCASE, CFOOCASE

Las macros que coinciden con una entrada en ciertos casos a menudo se denominan con un `CASE` -postfix. A menudo hay una variante `E...CASE` , que señala un error si la entrada no coincide con ninguno de los casos, y `C...CASE` , que señala un error continuo. Deben tener sintaxis como

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

Ejemplos de la norma: `CASE` , `TYPECASE` , `HANDLER-CASE` .

Por ejemplo, una macro que hace coincidir una cadena con expresiones regulares y vincula los grupos de registro a las variables. Utiliza CL-PPCRE para expresiones regulares.

```
(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

Las macros que definen cosas generalmente se nombran con `DEFINE-` o `DEF` -prefix.

Ejemplos de la norma: `DEFUN` , `DEFMACRO` , `DEFINE-CONDITION` .

Macros anafóricas

Una macro anafórica es una macro que introduce una variable (a menudo `IT`) que captura el resultado de un formulario proporcionado por el usuario. Un ejemplo común es el If anafórico, que es como un `IF` regular, pero también define la variable `IT` para referirse al resultado del formulario de prueba.

```
(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

La expansión de macros es el proceso de convertir macros en código real. Esto suele suceder como parte del proceso de compilación. El compilador expandirá todas las formas de macro antes de compilar realmente el código. La expansión de macros también ocurre durante la *interpretación* del código Lisp.

Uno puede llamar a `MACROEXPAND` manualmente para ver a qué se expande una forma 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` es el mismo, pero solo se expande una vez. Esto es útil cuando se trata de dar sentido a una forma macro que se expande a otra forma macro.

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

Tenga en cuenta que ni `MACROEXPAND` ni `MACROEXPAND-1` expanden el código Lisp en todos los niveles. Sólo expanden la forma de macro de nivel superior. Para expandir una forma completamente en todos los niveles, se necesita un *caminante de código* para hacerlo. Esta instalación no se proporciona en el estándar Common Lisp.

Backquote - escribiendo plantillas de código para macros

Código de retorno de macros. Dado que el código en Lisp consta de listas, se pueden usar las funciones de manipulación de listas normales para generarlas.

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

Esto suele ser muy difícil de leer, especialmente en macros más largas. La macro del lector de Backquote permite escribir plantillas entre comillas que se completan mediante la evaluación selectiva de los elementos.

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

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

Esta versión se parece casi a un código regular. Las comas se utilizan para evaluar `FORM` ; Todo lo demás se devuelve como es. Tenga en cuenta que en `',form` la comilla simple está fuera de la

coma, por lo que se devolverá.

También se puede usar `,@` para empalmar una lista en la posición.

```
(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 puede ser usado fuera de macros también.

Símbolos únicos para evitar choques de nombre en macros

La expansión de una macro a menudo necesita usar símbolos que no fueron pasados como argumentos por el usuario (como nombres para variables locales, por ejemplo). Uno debe asegurarse de que tales símbolos no puedan entrar en conflicto con un símbolo que el usuario está usando en el código circundante.

Esto generalmente se logra mediante el uso de `GENSYM` , una función que devuelve un nuevo símbolo sin entramar.

Malo

Considere la macro a continuación. Hace un `DOTIMES` -loop que también recopila el resultado del cuerpo en una lista, que se devuelve al final.

```
(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)
```

Esto parece funcionar en este caso, pero si el usuario tiene un nombre de resultado `RESULT` , que utiliza en el cuerpo, los resultados probablemente no sean lo que el usuario espera. Considere

este intento de escribir una función que recopile una lista de sumas de todos los enteros hasta `N` :

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

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

Bueno

Para solucionar el problema, necesitamos usar `GENSYM` para generar un nombre único para la variable `RESULT` en la expansión de macros.

```
(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: Cómo hacer símbolos a partir de cuerdas.

TODO: Evitar problemas con símbolos en diferentes paquetes.

si-vamos, cuando-dejemos macros de entrada

Estas macros combinan flujo de control y enlace. Son una mejora con respecto a las macros anafóricas anafóricas porque permiten que el desarrollador comunique el significado a través de nombres. Como tal, se recomienda su uso sobre sus contrapartes anafóricas.

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

`FOO-LET` macros `FOO-LET` unen una o más variables, y luego usan esas variables como la forma de prueba para el condicional correspondiente (`IF` , `WHEN`). Las variables múltiples se combinan con `AND` . La rama elegida se ejecuta con los enlaces en vigor. Una implementación simple de una variable de `IF-LET` podría ser algo como:

```
(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.~%")))
```

Una versión que admite múltiples variables está disponible en la biblioteca de Alexandria .

Usando macros para definir estructuras de datos

Un uso común de las macros es crear plantillas para estructuras de datos que obedecen a reglas comunes pero que pueden contener campos diferentes. Al escribir una macro, puede permitir que se especifique la configuración detallada de la estructura de datos sin necesidad de repetir el código repetitivo, ni usar una estructura menos eficiente (como un hash) en la memoria simplemente para simplificar la programación.

Por ejemplo, supongamos que deseamos definir un número de clases que tienen un rango de propiedades diferentes, cada una con un captador y definidor. Además, para algunas (pero no todas) de estas propiedades, deseamos que el setter llame un método en el objeto que le notifique que la propiedad ha sido cambiada. Aunque Common LISP ya tiene una forma abreviada para escribir getters y setters, escribir un setter personalizado estándar de esta manera normalmente requeriría la duplicación del código que llama al método de notificación en cada setter, lo que podría ser una molestia si hay una gran cantidad de propiedades involucradas. . Sin embargo, al definir una macro es mucho más fácil:

```
(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)))
```

Ahora podemos escribir `(defclass-notifier-slots foo (bar baz qux) (waldo))` e inmediatamente definir una clase `foo` con un slot regular `waldo` (creado por la segunda parte de la macro con la especificación `(waldo :accessor waldo)`) , y `bar` ranuras, `baz` y `qux` con configuradores que llaman al método `changed` (donde el captador se define por la primera parte de la macro, `(bar :reader bar)` , y el definidor por la macro del `notifier` invocado).

Además de permitirnos definir rápidamente múltiples clases que se comportan de esta manera, con un gran número de propiedades, sin repetición, tenemos el beneficio habitual de la reutilización de código: si más tarde decidimos cambiar la forma en que funcionan los métodos de notificación, simplemente podemos cambiar la macro, y la estructura de cada clase que lo use cambiará.

Lea macros en línea: https://riptutorial.com/es/common-lisp/topic/1257/macros

Capítulo 24: Mesas de hash

Examples

Creando una tabla hash

Las tablas hash son creadas por `make-hash-table` :

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

La función puede tomar parámetros de palabras clave para especificar aún más el comportamiento de la tabla hash resultante:

- `test` : selecciona la función que se usa para comparar claves para la igualdad. Tal vez un designador para una de las funciones `eq` , `eql` , `equal` o `equalp` . El valor predeterminado es `eq` .
- `size` : una sugerencia para la implementación sobre el espacio que puede requerirse inicialmente.
- `rehash-size` : si es un número entero (> = 1), al hacer un rehash, la tabla hash aumentará su capacidad en el número especificado. De lo contrario, un flotador (> 1.0), entonces la tabla hash aumentará su capacidad al producto del `rehash-size` y la capacidad anterior.
- `rehash-threshold` : especifica qué tan completa debe estar la tabla hash para activar un rehash.

Iterando sobre las entradas de una tabla hash con maphash

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

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

El uso de `maphash` permite iterar sobre las entradas de una tabla hash. El orden de iteración no está especificado. El primer argumento es una función que acepta dos parámetros: la clave y el valor de la entrada actual.

`maphash` siempre devuelve `NIL` .

Iterando sobre las entradas de una tabla hash con bucle

La macro de **bucle** admite la iteración sobre las claves, los valores o las claves y valores de una tabla hash. Los siguientes ejemplos muestran posibilidades, pero la sintaxis de **bucle** completo permite más combinaciones y variantes.

Sobre llaves y valores.

```
(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))
```

Sobre llaves

```
(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)
```

Sobre valores

```
(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)
```

Iterando sobre las entradas de una tabla hash con un iterador de tabla hash

Las claves y los valores de una tabla hash pueden repetirse utilizando la macro **with-hash-table-iterator** . Esto puede ser un poco más complejo que el **maphash** o el **bucle** , pero podría usarse para implementar las construcciones de iteración utilizadas en esos métodos. **with-hash-table-iterator** toma un nombre y una tabla hash y vincula el nombre dentro de un cuerpo de modo que las sucesivas llamadas al nombre produzcan múltiples valores: (i) un valor booleano que indica si un valor está presente; (ii) la clave de la entrada; y (iii) el valor de la entrada.

```
(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)
```

Lea Mesas de hash en línea: https://riptutorial.com/es/common-lisp/topic/4482/mesas-de-hash

Capítulo 25: Personalización

Examples

Más características para el Read-Eval-Print-Loop (REPL) en un terminal

CLISP tiene una integración con GNU Readline.

Para mejoras para otras implementaciones, consulte: Cómo personalizar el REPL SBCL .

Archivos de inicialización

La mayoría de las implementaciones de Common Lisp intentarán cargar un *archivo* de inicio al inicio:

Implementación	Archivo inicial	Archivo de inicio de sitio / sistema
ABCL	`$HOME/.abclrc`	
Allegro cl	`$HOME/.clinit.cl`	
ECL	`$HOME/.eclrc`	
Corchete	`$HOME/.clasprc`	
Apretar	`$HOME/.clisprc.lisp`	
Clozure CL	`home:ccl-init.lisp` o `home:ccl-init.fasl` o `home:.ccl-init.lisp`	
CMUCL	`$HOME/.cmucl-init.lisp`	
LispWorks	`$HOME/.lispworks`	
MKCL	`$HOME/.mkclrc`	
SBCL	`$HOME/.sbclrc`	`$SBCL_HOME/sbclrc` o `/etc/sbclrc`
SCL	`$HOME/.scl-init.lisp`	

Archivos de inicialización de muestra:

Implementación	Archivo inicial de muestra
LispWorks	`Library/lib/7-0-0-0/config/a-dot-lispworks.lisp`

Ajustes de optimización

Common Lisp tiene una forma de influir en las estrategias de compilación. Tiene sentido definir sus valores preferidos.

Los valores de optimización están entre 0 (sin importancia) y 3 (extremadamente importante). **1 es el valor neutral.**

Es útil usar siempre el código de seguridad (safety = 3) con todas las comprobaciones de tiempo de ejecución habilitadas.

Tenga en cuenta que la interpretación de los valores es específica de la implementación. La mayoría de las implementaciones de Lisp comunes hacen uso de estos valores.

Ajuste	Explicación	valor predeterminado útil	valor de entrega útil
`compilation-speed`	Velocidad del proceso de compilación.	2	0
`debug`	facilidad de depuración	2	1 o 0
`safety`	comprobación de errores en tiempo de ejecución	3	2
`space`	Tamaño de código y espacio de tiempo de ejecución	2	2
`speed`	velocidad del código objeto	2	3

Una declaración de `optimize` para usar con `declaim` , `declare` y `proclaim` :

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

Tenga en cuenta que también puede aplicar configuraciones de optimización especiales a partes del código en una función usando la macro `LOCALLY` .

Lea Personalización en línea: https://riptutorial.com/es/common-lisp/topic/5679/personalizacion

Capítulo 26: Protocolo de metaobjetos CLOS

Examples

Obtener los nombres de tragamonedas de una clase

Digamos que tenemos una clase como

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

Para obtener los nombres de las ranuras de la clase usamos la función class-slots. Esto se puede encontrar en el paquete de fregona más cercana, proporcionado por el sistema de fregona más cercana. Para cargar la imagen lisp que utilizamos `(ql:quickload :closer-mop)` . También debemos asegurarnos de que la clase esté finalizada antes de llamar a los espacios de clase.

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

que devuelve una lista de objetos de *definición de ranura efectivos* :

```
(#<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>)
```

Actualizar una ranura cuando se modifica otra ranura

CLOS MOP proporciona la clase de uso de valor de ranura de gancho, que se llama cuando se accede, lee o modifica una ranura. Debido a que solo nos preocupamos por las modificaciones en este caso, definimos un método para `(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))))
```

Tenga en cuenta que debido a que en la creación de la instancia no se llama `slot-value` , puede ser necesario duplicar el código en la `initialize-instance :after` método

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

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

Lea Protocolo de metaobjetos CLOS en línea: https://riptutorial.com/es/common-lisp/topic/2901/protocolo-de-metaobjetos-clos

Capítulo 27: Recursion

Observaciones

Lisp se usa a menudo en contextos educativos, donde los estudiantes aprenden a comprender e implementar algoritmos recursivos.

El código de producción escrito en Common Lisp o código portátil tiene varios problemas con la recursión: no hacen uso de las características específicas de la implementación, como la *optimización de llamadas de cola* , lo que a menudo hace que sea necesario evitar la recursión por completo. En estos casos, las implementaciones:

- Generalmente tienen un *límite de profundidad de recursión* debido a los límites en los tamaños de pila. Por lo tanto, los algoritmos recursivos solo funcionarán para datos de tamaño limitado.
- No siempre proporcione optimización de llamadas de cola, especialmente en combinación con operaciones de alcance dinámico.
- Solo proporciona optimización de llamadas de cola en ciertos niveles de optimización.
- No suele proporcionar *optimización de llamada de cola* .
- Por lo general, no proporcionan la *optimización de llamadas de cola* en ciertas plataformas. Por ejemplo, las implementaciones en JVM pueden no hacerlo, ya que la JVM en sí no admite la *optimización de llamadas de cola* .

Reemplazar llamadas de la cola por saltos generalmente hace que la depuración sea más difícil; Agregar saltos hará que los marcos de pila no estén disponibles en un depurador. Como alternativas, Common Lisp ofrece:

- Construcciones de iteración, como `DO` , `DOTIMES` , `LOOP` y otras.
- Funciones de orden superior, como `MAP` , `REDUCE` y otras.
- Varias estructuras de control, incluyendo bajo nivel `go to`

Examples

Recursión de plantilla 2 de múltiples condiciones.

```
(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
```

Recursión plantilla 1 sola condición recursión de cola única

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

Calcular el número 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.
            )
        )
    )
)
```

Imprimir recursivamente los elementos de una lista

```
;;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))
        )
    )
)
```

Para probar esto, ejecute:

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

El resultado será:

```
1
2
3
4
```

Calcular el factorial de un número entero

Un algoritmo fácil de implementar como función recursiva es factorial.

```
;;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)))
        )
    )
)
```

Lea Recursion en línea: https://riptutorial.com/es/common-lisp/topic/3190/recursion

Capítulo 28: secuencia - cómo dividir una secuencia

Sintaxis

1. dividir regex target-string y key start end limit with-registers-p omit-unmatched-p sharedp => list
2. lispworks: secuencia separadora-bolsa de secuencia dividida y clave inicio final prueba clave coalescencia-separadores => secuencias
3. secuencia de delimitador de secuencia dividida y clave inicio final desde el final conteo remove-empty-subseqs test test-not key => list of subsences

Examples

Dividir cadenas usando expresiones regulares

La biblioteca CL-PPCRE proporciona la `split` funciones que nos permite dividir cadenas en subcadenas que coinciden con una expresión regular, descartando las partes de la cadena que no lo hacen.

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

SPLIT-SEQUENCE en LIspWorks

División simple de una cadena de número de IP.

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

Simple división de una URL:

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

Usando la biblioteca de secuencia dividida

La biblioteca de secuencia dividida proporciona una función `split-sequence` , que permite dividir en elementos de una secuencia

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

Lea secuencia - cómo dividir una secuencia en línea: https://riptutorial.com/es/common-lisp/topic/1454/secuencia---como-dividir-una-secuencia

Capítulo 29: Tipos de listas

Examples

Listas simples

Las listas simples son el tipo de lista más simple en Common Lisp. Son una secuencia ordenada de elementos. Admiten operaciones básicas como obtener el primer elemento de una lista y el resto de una lista en tiempo constante, admiten acceso aleatorio en tiempo lineal.

```
(list 1 2 3)
;=> (1 2 3)

(first (list 1 2 3))
;=> 1

(rest (list 1 2 3))
;=> (2 3)
```

Hay muchas funciones que operan en listas "simples", en la medida en que solo se preocupan por los elementos de la lista. Estos incluyen **encontrar** , **mapcar** , y muchos otros. (Muchas de esas funciones también funcionarán en 17.1 Conceptos de secuencia para algunas de estas funciones.

Listas de asociaciones

Las listas simples son útiles para representar una secuencia de elementos, pero a veces es más útil representar un tipo de clave para la asignación de valores. Common Lisp proporciona varias formas de hacer esto, incluyendo tablas hash genuinas (ver 18.1 Hash Table Concepts). Hay dos formas principales o que representan la clave para las asignaciones de valor en Common Lisp: listas de propiedades y listas de asociación . Este ejemplo describe las listas de asociaciones.

Una lista de asociaciones, o *alista,* es una *lista* "simple" cuyos elementos son pares de puntos en los que el *carro* de cada par es la clave y el valor asociado es la *CDR* de cada par. Por ejemplo,

```
(defparameter *ages* (list (cons 'john 34) (cons 'mary 23) (cons 'tim 72)))
```

se puede considerar como una lista de asociaciones que asigna símbolos que indican un nombre personal con un número entero que indica la edad. Es posible implementar algunas funciones de recuperación utilizando funciones de lista simple, como **miembro** . Por ejemplo, para recuperar la edad de **John** , uno podría escribir

```
(cdr (first (member 'mary *age* :key 'car)))
;=> 23
```

La función **miembro** devuelve la cola de la lista que comienza con una celda de contras cuyo *auto* es **mary** , es decir, **((mary. 23) (tim. 72))** , **primero** devuelve el primer elemento de esa lista, que es **(mary. 23)** , y **cdr** devuelve el lado derecho de ese par, que es **23** . Si bien esta es una

forma de acceder a los valores en una lista de asociación, el propósito de una convención como las listas de asociación es abstraerse de la representación subyacente (una lista) y proporcionar funciones de nivel superior para trabajar con la estructura de datos.

Para las listas de asociación, la función de recuperación es **assoc** , que toma una clave, una lista de asociación y palabras clave de prueba opcionales (clave, prueba, prueba-no) y devuelve el par para la clave correspondiente:

```
(assoc 'tim *ages*)
;=> (tim . 72)
```

Dado que el resultado siempre será una celda de contras si un elemento está presente, si **assoc** devuelve **nil** , entonces el elemento no estaba en la lista:

```
(assoc 'bob *ages*)
;=> nil
```

Para actualizar los valores en una lista de asociaciones, **setf** puede usarse junto con **cdr** . Por ejemplo, cuando llega el cumpleaños de **John** y aumenta su edad, se puede realizar una de las siguientes acciones:

```
(setf (cdr (assoc 'john *ages*) 35)

(incf (cdr (assoc 'john *ages*)))
```

incf funciona en este caso porque se basa en **setf** .

Las listas de asociación también se pueden usar como un tipo de mapa bidireccional, ya que las asignaciones de clave a valor se recuperan según el valor mediante la función assoc invertida, **rassoc** .

En este ejemplo, la lista de asociaciones se creó mediante el uso de **listas** y **contras** explícitamente, pero las listas de asociaciones también se pueden crear utilizando **pairlis** , que toma una lista de claves y datos y crea una lista de asociaciones basada en ellas:

```
(pairlis '(john mary tim) '(23 67 82))
;=> ((john . 23) (mary . 67) (tim . 82))
```

Se puede agregar un solo par de clave y valor a una lista de asociación usando un **acons** :

```
(acons 'john 23 '((mary . 67) (tim . 82)))
;=> ((john . 23) (mary . 67) (tim . 82))
```

La función **assoc** busca en la lista de izquierda a derecha, lo que significa que es posible "enmascarar" los valores en una lista de asociación sin eliminarlos de una lista o actualizar la estructura de la lista, simplemente agregando nuevos elementos a la lista. comienzo de la lista. La función de **acons** se proporciona para esto:

```
(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))
```

Y ahora, una búsqueda de **Mary** devolverá la primera entrada:

```
(assoc 'mary *new-ages*)
;=> 29
```

Listas de propiedades

Las listas simples son útiles para representar una secuencia de elementos, pero a veces es más útil representar un tipo de clave para la asignación de valores. Common Lisp proporciona varias formas de hacer esto, incluyendo tablas hash genuinas (ver 18.1 Hash Table Concepts). Hay dos formas principales o que representan la clave para las asignaciones de valor en Common Lisp: listas de propiedades y listas de asociación . Este ejemplo describe las listas de propiedades.

Una lista de propiedades, o *plist* , es una lista "simple" en la que los valores alternos se interpretan como claves y sus valores asociados. Por ejemplo:

```
(defparameter *ages* (list 'john 34 'mary 23 'tim 72))
```

se puede considerar como una lista de propiedades que asigna símbolos que indican un nombre personal con un número entero que indica la edad. Es posible implementar algunas funciones de recuperación utilizando funciones de lista simple, como **miembro** . Por ejemplo, para recuperar la edad de **John** , uno podría escribir

```
(second (member 'mary *age*))
;=> 23
```

La función **miembro** devuelve la cola de la lista que comienza con **mary** , es decir, **(mary 23 tim 72)** , y la **segunda** devuelve el segundo elemento de esa lista, que es **23** . Si bien esta es una forma de acceder a los valores en una lista de propiedades, el propósito de una convención como las listas de propiedades es abstraerse de la representación subyacente (una lista) y proporcionar funciones de nivel superior para trabajar con la estructura de datos.

Para las listas de propiedades, la función de recuperación es **getf** , que toma la lista de propiedades, una clave (más comúnmente llamada *indicador*) y un valor predeterminado opcional para devolver en caso de que la lista de propiedades no contenga un valor para la clave.

```
(getf *ages* 'tim)
;=> 72

(getf *ages* 'bob -1)
;=> -1
```

Para actualizar los valores en una lista de propiedades, se puede usar **setf** . Por ejemplo, cuando llega el cumpleaños de **John** y aumenta su edad, se puede realizar una de las siguientes acciones:

```
(setf (getf *ages* 'john) 35)

(incf (getf *ages* 'john))
```

incf funciona en este caso porque se basa en **setf** .

Para buscar varias propiedades en una lista de propiedades una vez, use **get-properties** .

La función **getf** busca en la lista de izquierda a derecha, lo que significa que es posible "enmascarar" los valores en una lista de propiedades sin eliminarlos de una lista o actualizar la estructura de la lista. Por ejemplo, usando la **lista *** :

```
(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)
```

Y ahora, una búsqueda de **Mary** devolverá la primera entrada:

```
(getf *new-ages* 'mary)
;=> 29
```

Lea Tipos de listas en línea: https://riptutorial.com/es/common-lisp/topic/3744/tipos-de-listas

Capítulo 30: Trabajando con bases de datos

Examples

Uso simple de PostgreSQL con Postmodern

Postmodern es una biblioteca para interactuar con la base de datos relacional PostgreSQL . Ofrece varios niveles de acceso a PostgreSQL, desde la ejecución de consultas SQL representadas como cadenas, o como listas, hasta un mapeo objeto-relacional.

La base de datos utilizada en los siguientes ejemplos se puede crear con estas sentencias de 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}');
```

El primer ejemplo muestra el resultado de una consulta simple que devuelve una relación:

```
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
```

Tenga en cuenta que el resultado se puede devolver como una lista de listas o listas que agregan los parámetros opcionales `:alists` o `:plists` a la función de consulta.

Una alternativa a la `query` es `doquery` , para iterar sobre los resultados de una consulta. Sus parámetros son `query (&rest names) &body body` , donde los nombres están vinculados a los valores en la fila en cada iteración:

```
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
```

Cuando la consulta requiere parámetros, se pueden usar instrucciones preparadas:

```
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 función `prepare` recibe una consulta con marcadores `$1` posición `$1` , `$2` , etc. y devuelve una nueva función que requiere un parámetro para cada marcador de posición y ejecuta la consulta cuando se le llama con el número correcto de argumentos.

En caso de actualizaciones, la función `exec` devuelve el número de tuplas modificadas (las dos declaraciones DDL se incluyen en una transacción):

```
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
```

Además de escribir consultas SQL como cadenas, se pueden usar listas de palabras clave, símbolos y constantes, con una sintaxis que recuerda a 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
```

Lea Trabajando con bases de datos en línea: https://riptutorial.com/es/common-lisp/topic/4558/trabajando-con-bases-de-datos

Capítulo 31: Trabajando con SLIME

Examples

Instalación

Es mejor usar el último SLIME del repositorio de Emacs MELPA: los paquetes pueden ser un poco inestables, pero usted obtiene las últimas características.

Portales y multiplataforma Emacs, Slime, Quicklisp, SBCL y Git

Puede descargar una versión portátil y multiplataforma de Emacs25 ya configurada con Slime, SBCL, Quicklisp y Git: Portacle . Es una forma rápida y fácil de ponerse en marcha. Si quieres aprender a instalar todo tú mismo, sigue leyendo.

Manual de instalación

En el archivo de inicialización de GNU Emacs (> = 24.5) (`~/.emacs` o `~/.emacs.d/init.el`) agregue lo siguiente:

```
;; 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 por sí solo está bien, pero funciona mejor con el gestor de paquetes Quicklisp . Para instalar Quicklisp, siga las instrucciones del sitio web (si utiliza roswell , siga las instrucciones de roswell). Una vez instalado, en su invocación lisp:

```
(ql:quickload :quicklisp-slime-helper)
```

y agregue las siguientes líneas al archivo de inicio de 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)))
```

Después del reinicio, GNU Emacs instalará y configurará todos los paquetes necesarios.

Iniciar y finalizar SLIME, comandos REPL especiales (comas)

En Emacs `Mx slime` iniciará slime con la implementación predeterminada (primera) Common Lisp. Si se proporcionan múltiples implementaciones (a través de las implementaciones de `slime-lisp-implementations` variable), se puede acceder a otras implementaciones a través de `M-- Mx slime` , que ofrecerá la opción de implementaciones disponibles en mini-buffer.

`Mx slime` abrirá el búfer REPL que se verá como sigue:

```
; SLIME 2016-04-19
CL-USER>
```

El búfer SLIME REPL acepta algunos comandos especiales. Todos ellos comienzan con `,` . Una vez `,` se escribe, la lista de opciones se mostrará en mini-buffer. Incluyen:

- `,quit`
- `,restart-inferior-lisp`
- `,pwd` : imprime el directorio actual desde donde se ejecuta Lisp
- `,cd` - cambiará el directorio actual

Usando 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)
```

El resultado que se imprime después de la evaluación no es solo una cadena: hay un objeto Lisp completo detrás que se puede inspeccionar haciendo clic derecho en él y seleccionando Inspeccionar.

La entrada multilínea también es posible: use `Cj` para poner una nueva línea. `Enter` enter hará que se evalúe el formulario ingresado y, si no se completa, es probable que cause un error:

```
CL-USER> (mapcar (lambda (x y)
                   (declare (ignore y))
                   (* x 2))
                 '(1 2 3)
                 '(:a :b :c))
(2 4 6)
```

Manejo de errores

Si la evaluación causa un error:

```
CL-USER> (/ 3 0)
```

Esto abrirá un búfer de depuración con el siguiente contenido (en SBCL lisp):

```
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 ---
```

Mover el cursor hacia abajo pasado `--- more ---` hará que el retroceso se expanda más.

En cada línea del retroceso, al presionar `Enter` se mostrará más información sobre una llamada en particular (si está disponible).

Si presiona `Enter` en la línea de reinicios, se invocará un reinicio en particular. Alternativamente, el reinicio se puede elegir con el número `0` , `1` o `2` (presione la tecla correspondiente en cualquier lugar del búfer). El reinicio predeterminado está marcado con una estrella y se puede invocar presionando la tecla `q` (para "salir"). Presionando `q` cerrará el depurador y mostrará lo siguiente en REPL

```
; Evaluation aborted on #<DIVISION-BY-ZERO {10064CCE43}>.
CL-USER>
```

Finalmente, muy raramente, pero Lisp puede encontrar un error que no puede ser manejado por el depurador de Lisp, en cuyo caso caerá en el depurador de bajo nivel o finalizará de manera anormal. Para ver la causa de este tipo de error, cambie al búfer `*inferior-lisp*` .

Configuración de un servidor SWANK sobre un túnel SSH.

1. Instale una implementación de Common Lisp en el servidor. (Por `sbcl` , `sbcl` , `clisp` , etc ...)
2. Instalar `quicklisp` en el servidor.
3. Cargue SWANK con `(ql:quickload :swank)`
4. Inicie el servidor con `(swank:create-server)` . El puerto predeterminado es `4005` .
5. [En su máquina local] Cree un túnel SSH con `ssh -L4005:127.0.0.1:4005 [remote machine]`
6. Conéctese al servidor swank remoto en ejecución con `Mx slime-connect` . El host debe ser `127.0.0.1` y el puerto `4005` .

Lea Trabajando con SLIME en línea: https://riptutorial.com/es/common-lisp/topic/4097/trabajando-con-slime

Creditos

S. No	Capítulos	Contributors
1	Empezando con common-lisp	blambert, Community, CPHPython, Dan Robertson, Ehvince, Gustav Bertram, Inaimathi, JAL, Rainer Joswig, Renzo, Robert Columbia, WarFox
2	ANSI Common Lisp, el estándar de lenguaje y su documentación.	Rainer Joswig, sds
3	ASDF - Otra facilidad de definición de sistema	Inaimathi, jkiiski, Joao Tavora, PuercoPop, Rainer Joswig, Sim, Svante
4	Booleanos y booleanos generalizados	Rainer Joswig, Renzo, Terje D.
5	Bucles basicos	Joshua Taylor, MatthewRock, Rainer Joswig, sadfaf, Svante
6	Citar	MatthewRock, Rainer Joswig, Svante
7	CLOS - el sistema de objetos Common Lisp	Joshua Taylor, PuercoPop, Rainer Joswig, Sim
8	Contras celdas y listas	eyqs, Joshua Taylor, Rainer Joswig, Renzo
9	Corrientes	jkiiski, Rainer Joswig, Renzo, Svante
10	Creando Binarios	Inaimathi
11	Estructuras de Control	eyqs, Rainer Joswig, Robert Columbia, Soupy, Svante, Throwaway Account 3 Million
12	Examen de la unidad	Ehvince, PuercoPop, Rainer Joswig, sadfaf
13	Expresiones regulares	jkiiski, PuercoPop
14	Formas de agrupación	Joshua Taylor, Rainer Joswig

15	formato	Dan Robertson, Inaimathi, jkiiski, otyn, Renzo
16	Funciones	jkiiski, Rainer Joswig, Svante
17	Funciones como valores de primera clase.	Dan Robertson, Joshua Taylor, PuercoPop, Rainer Joswig, Renzo
18	Funciones de mapeo sobre listas	Aaron, Rainer Joswig, Renzo
19	Igualdad y otros predicados de comparación.	Renzo
20	La coincidencia de patrones	jkiiski, PuercoPop
21	Léxico vs variables especiales	Rainer Joswig, Terje D.
22	LOOP, una macro de Common Lisp para iteración	Dan Robertson, Elias Mårtenson, Inaimathi, PuercoPop, Rainer Joswig, RamenChef, Renzo, Throwaway Account 3 Million
23	macros	JAL, jkiiski, Joshua Taylor, Mark Green, PuercoPop, Rainer Joswig
24	Mesas de hash	Daniel Jour, Joshua Taylor
25	Personalización	Daniel Kochmański, Rainer Joswig
26	Protocolo de metaobjetos CLOS	PuercoPop, Rainer Joswig
27	Recursion	4444, Rainer Joswig, Robert Columbia, sadfaf
28	secuencia - cómo dividir una secuencia	PuercoPop, Rainer Joswig, sadfaf
29	Tipos de listas	jkiiski, Joshua Taylor
30	Trabajando con bases de datos	Renzo
31	Trabajando con SLIME	Ehvince, mobiuseng, tsikov