FREE eBook

LEARNING common-lisp

Free unaffiliated eBook created from
Stack Overflow contributors.

#common-lisp

Table of Contents

About

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

Chapter 1: Getting started with common-lisp

Remarks

This is a simple hello world function in Common Lisp. Examples will print the text `Hello, World!` (without quotation marks; followed by a newline) to the standard output.

Common Lisp is a programming language that is largely used interactively using an interface known as a REPL. The REPL (Read Eval Print Loop) allows one to type code, have it evaluated (run) and see the results immediately. The prompt for the REPL (at which point one types the code to be run) is indicated by `CL-USER>`. Sometimes something other than `CL-USER` will appear before the `>` but this is still a REPL.

After the prompt comes some code, usually either a single word (i.e. a variable name) or a form (a list of words/forms enclosed between `(` and `)`) (i.e. a function call or declaration, etc). On the next line will be any output that the program prints (or nothing if the program prints nothing) and then the values returned by evaluating the expression. Normally an expression returns one value but if it returns multiple values they appear once per line.

Versions

Version	Release Date
Common Lisp	1984-01-01
ANSI Common Lisp	1994-01-01

Examples

Hello World

What follows is an excerpt from a REPL session with Common Lisp in which a "Hello, World!" function is defined and executed. See the remarks at the bottom of this page for a more thorough description of a REPL.

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

This defines the "function" of zero arguments named `hello`, which will write the string `"Hello, World!"` followed by a newline to standard output, and return `NIL`.

To define a function we write

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

In this case the function is called `hello`, takes no parameters and the code it runs is to do one function call. The returned value from a lisp function is the last bit of code in the function to run so `hello` returns whatever `(format t "Hello, World!~%")` returns.

In lisp to call a function one writes `(function-name arguments...)` where `function-name` is the name of the function and `arguments...` is the (space-separated) list of arguments to the call. There are some special cases which look like function calls but are not, for example, in the above code there is no `defun` function that gets called, it gets treated specially and defines a function instead.

At the second prompt of the REPL, after we have defined the `hello` function, we call it with no parameters by writing `(hello)`. This in turn will call the `format` function with the parameters `t` and `"Hello, World!~%"`. The `format` function produces formatted output based on the arguments which it is given (a bit like an advanced version of `printf` in C). The first argument tells it where to output to, with `t` meaning standard-output. The second argument tells it what to print (and how to interpret any extra parameters). The directive (special code in the second argument) `~%` tells format to print a newline (i.e. on UNIX it might write `\n` and on windows `\r\n`). Format usually returns `NIL` (a bit like `NULL` in other languages).

After the second prompt we see that `Hello, World` has been printed and on the next line that the returned value was `NIL`.

Hello, Name

This is a slightly more advanced example that shows a few more features of common lisp. We start with a simple `Hello, World!` function and demonstrate some interactive development at the REPL. Note that any text from a semicolon, `;`, to the rest of the line is a comment.

```
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>
```

This highlights some of the advanced features of Common Lisp's `format` function as well as some features like optional parameters and keyword arguments (e.g. `:number`). This also gives an example of interactive development at a REPL in common lisp.

The simple Hello World program in REPL

Common Lisp REPL is an interactive environment. Every form written after the prompt is

evaluated, and its value is afterwards printed as result of the evaluation. So the simplest possible “Hello, World!” program in Common Lisp is:

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

What happens here is that a string costant is given in input to the REPL, it is evaluated and the result is printed. What can be seen from this example is that strings, like numbers, special symbols like `NIL` and `T` and a few other literals, are *self-evaluating* forms: that is they evaluate to themselves.

Basic expressions

Let’s try some basic expression in the 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>
```

The basic building block of a Common Lisp program is the *form*. In these examples we have *functions forms*, that is expressions, written as list, in which the first element is an operator (or function) and the rest of the elements are the operands (this is called “Prefix Notation”, or “Polish Notation”). Writing forms in the REPL causes their evaluation. In the examples you can see simple expressions whose arguments are constant numbers, strings and symbols (in the case of `'string`, which is the name of a type). You can also see that arithmetic operators can take any number of arguments.

It is important to note that parentheses are an integral part of the syntax, and cannot be used freely as in other programming languages. For instance the following is an error:

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

In Common Lisp forms can also be data, symbols, macro forms, special forms and lambda forms. They can be written to be evaluated, returning zero, one, or more values, or can be given in input to a macro, that transform them in other forms.

Sum of list of integers

```
(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))
```

Lambda Expressions and Anonymous Functions

An anonymous function can be defined without a name through a Lambda Expression. For defining these type of functions, the keyword `lambda` is used instead of the keyword `defun`. The following lines are all equivalent and define anonymous functions which output the sum of two numbers:

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

Their usefulness is noticeable when creating Lambda forms, i.e. a form that is a list where the first element is the lambda expression and the remaining elements are the anonymous function's arguments. Examples (online execution):

```
(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)
```

Common Lisp Learning Resources

Online Books

These are books that are freely accessible online.

- Practical Common Lisp by Peter Seibel is a good introduction to CL for experienced programmers, which tries to highlight from the very beginning what makes CL different to other languages.
- Common Lisp: A Gentle Introduction to Symbolic Computation by David S. Touretzky is a good introduction for people new to programming.
- Common Lisp: An interactive approach by Stuart C. Shapiro was used as a course textbook, and course notes accompany the book on the website.
- Common Lisp, the Language by Guy L. Steele is a description of the Common Lisp language. According to the CLiki it is outdated, but it contains better descriptions of the loop macro and format than the Common Lisp Hyperspec does.
- On Lisp by Paul Graham is a great book for intermediately experienced Lispers.
- Let Over Lambda by Doug Hoyte is an advanced book on Lisp Macros. Several people recommended that you be comfortable with On Lisp before reading this book, and that the start is slow.

Online References

- The Common Lisp Hyperspec is *the* language reference document for Common Lisp.
- The Common Lisp Cookbook is a list of useful Lisp recipes. Also contains a list of other online sources of CL information.
- Common Lisp Quick Reference has printable Lisp reference sheets.
- Lispdoc.com searches several sources of Lisp information (Practical Common Lisp, Successful Lisp, On Lisp, the HyperSpec) for documentation.
- L1sp.org is a redirect service for documentation.

Offline Books

These are books that you'll likely have to buy, or lend from a library.

- ANSI Common Lisp by Paul Graham.
- Common Lisp Recipes by Edmund Weitz.
- Paradigms of Artificial Intelligence Programming has many interesting applications of Lisp, but is not a good reference for AI any more.

Online Communities

- The CLiki has a great Getting Started Page. A great resource for all things CL. Has an extensive list of Lisp books.
- Common Lisp subreddit has loads of useful links and reference documents in the sidebar.
- IRC: #lisp, #ccl, #sbcl and others on Freenode.
- Common-Lisp.net provides hosting for many common lisp projects and user groups.

Libraries

- Quicklisp is library manager for Common Lisp, and has a long list of supported libraries.
- Quickdocs hosts library documentation for many CL libraries.
- Awesome CL is a community-driven curated list of libraries, frameworks and other shiny stuff sorted by category.

Pre-packaged Lisp Environments

These are Lisp editing environments that are easy to install and get started with because everything you need is pre-packaged and pre-configured.

- Portacle is a portable and multiplatform Common Lisp environment. It ships a slightly customized Emacs with Slime, SBCL (a popular Common Lisp implementation), Quicklisp and Git. No installation needed, so it's a very quick and easy way to get going.
- Lispbox is an IDE (Emacs + SLIME), Common Lisp environment (Clozure Common Lisp) and library manager (Quicklisp), pre-packaged as archives for Windows, Mac OSX and Linux. Descendant of "Lisp in a Box" Recommended in the Practical Common Lisp book.
- Not pre-packed, but SLIME turns Emacs into a Common Lisp IDE, and has a user manual to help you get started. Requires a separate Common Lisp implementation.

Common Lisp Implementations

This section lists some common CL implementations and their manuals. Unless otherwise noted,

these are free software implementations. See also the Cliki's list of free software Common Lisp Implementations, and Wikipedia's list of commercial Common Lisp Implementations.

- Allegro Common Lisp (ACL) and manual. Commercial, but has a free Express Edition and training videos on Youtube.
- CLISP and manual.
- Clozure Common Lisp (CCL) and manual.
- Carnegie Mellon University Common Lisp (CMUCL), has a manual and other useful information page.
- Embeddable Common Lisp (ECL) and manual.
- LispWorks and manual. Commercial, but has a Personal Edition with some limitations.
- Steel Bank Common Lisp (SBCL) and manual.
- Scieneer Common Lisp (SCL) and manual is a commercial Linux and Unix implementation, but has an unrestricted free evaluation and non-commercial use version.

Read Getting started with common-lisp online: https://riptutorial.com/common-lisp/topic/534/getting-started-with-common-lisp

Chapter 2: ANSI Common Lisp, the language standard and its documentation

Examples

Common Lisp HyperSpec

Common Lisp has a standard, which was initially published in 1994 as an ANSI standard.

The Common Lisp HyperSpec, short CLHS, provided by LispWorks is an often used HTML documentation, which is derived from the standard document. The HyperSpec can also be downloaded and locally used.

Common Lisp development environments usually allow lookup of the HyperSpec documentation for Lisp symbols.

- For GNU Emacs there is clhs.el.
- SLIME for GNU Emacs provides a version of hyperspec.el.

See also: cliki on CLHS

EBNF syntax declarations in documentation

The ANSI CL standard uses an extended EBNF syntax notation. The documentation duplicated on Stackoverflow should use the same syntax notation to reduce confusion.

Example:

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

Notation:

- `[foo]` -> zero or one `foo`
- `{foo}*` -> zero or more `foo`
- `foo | bar` ->`foo` or `bar`

Common Lisp the Language, 2nd Edition, by Guy L. Steele Jr.

This book is known as CLtL2.

This is the second edition of the book Common Lisp the Language. It was published in 1990,

before the ANSI CL standard was final. It took the original language definition from the first edition (published in 1984) and described all changes in the standardization process up to 1990 plus some extensions (like the SERIES iteration facility).

Note: CLTL2 describes a version of Common Lisp which is slightly different from the published standard from 1994. Thus always use the standard, and not CLtL2, as a reference.

CLtL2 still can be useful, because it provides information not found in the Common Lisp specification document.

There is a HTML version of Common Lisp the Language, 2nd Edition.

CLiki - Proposed ANSI Revisions and Clarifications

On CLiki, a Wiki for Common Lisp and *free* Common Lisp software, a list of Proposed ANSI Revisions and Clarifications is being maintained.

Since the Common Lisp standard has not changed since 1994, users have found several problems with the specification document. These are documented on the CLiki page.

Common Lisp Quick Reference

The Common Lisp Quick Reference is a document which can be printed and bound as a booklet in various layouts to have a printed quick reference for Common Lisp.

The ANSI Common Lisp standard in Texinfo format (especially useful for GNU Emacs)

GNU Emacs uses a special format for documentation: *info*.

The Common Lisp standard has been converted to the Texinfo format, which can be used to create documentation browsable with the *info* reader in GNU Emacs.

See here: dpans2texi.el converts the TeX sources of the draft ANSI Common Lisp standard (dpANS) to the Texinfo format.

Another version has been done for for GCL: gcl.info.tgz.

Read ANSI Common Lisp, the language standard and its documentation online: https://riptutorial.com/common-lisp/topic/2900/ansi-common-lisp--the-language-standard-and-its-documentation

Chapter 3: ASDF - Another System Definition Facility

Remarks

ASDF - Another System Definition Facility

ASDF is a tool for specifying how systems of Common Lisp software are made up of components (sub-systems and files), and how to operate on these components in the right order so that they can be compiled, loaded, tested, etc.

Examples

Simple ASDF system with a flat directory structure

Consider this simple project with a flat directory structure:

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

The `example.asd` file is really just another Lisp file with little more than an ASDF-specific function call. Assuming your project depends on the `drakma` and `clsql` systems, its contents can be something like this:

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

When you load this Lisp file, you tell ASDF about your `:example` system, but you're not loading the system itself yet. That is done either by `(asdf:require-system :example)` or `(ql:quickload :example)`.

And when you load the system, ASDF will:

1. Load the dependencies - in this case the ASDF systems `clsql` and `drakma`
2. *Compile and load* the components of your system, i.e. the Lisp files, based on the given dependencies

1. `packages` first (no dependencies)
2. `functions` after `packages` (as it only depends on `packages`), but before `main` (which depends on it)
3. `main` after `functions` (as it depends on `packages` and `functions`)
4. `tools` anytime after `packages`

Keep in mind:

- Enter the dependencies as they are needed (e.g. macro definitions are needed before usage). If you don't, ASDF will error when loading your system.
- All files listed end on `.lisp` but this postfix should be dropped in the asdf script
- If your system is named the same as its `.asd` file, and you move (or symlink) its folder into `quicklisp/local-projects/` folder, you can then load the project using `(ql:quickload "example")`.
- Libraries your system depends on have to be known to either ASDF (via the `ASDF:*CENTRAL-REGISTRY` variable) or Quicklisp (either via the `QUICKLISP-CLIENT:*LOCAL-PROJECT-DIRECTORIES*` variable or available in any of its dists)

How to define a test operation for a system

```
(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)
```

Notes:

- We are assuming that the *system* :foo-tests defines a *package* named "FOO-TESTS"
- `run-tests` is the entry point for the test runner
- uoip:symbol-call allows as to define a method that calls a function that hasn't been read yet. The package the function is defined in doesn't exist when we define the system

In what package should I define my ASDF system?

ASDF provides the package `ASDF-USER` for developers to define their packages in.

Read ASDF - Another System Definition Facility online: https://riptutorial.com/common-lisp/topic/670/asdf---another-system-definition-facility

Chapter 4: Basic loops

Syntax

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

Examples

dotimes

`dotimes` is a macro for integer iteration over a single variable from 0 below some parameter value. One of the simples examples would be:

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

0
1
2
3
4
NIL
```

Note that `NIL` is the returned value, since we did not provide one ourselves; the variable starts from 0 and throughout the loop becomes values from 0 to N-1. After the loop, the variable becomes the 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)
```

dolist

`dolist` is a looping macro created to easily loop through the lists. One of the simplest uses would be:

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

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

Note that since we did not provide return value, `NIL` is returned (and A,B,C,D are printed to `*standard-output*`).

`dolist` can also return values:

```
;;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
```

Simple loop

The **loop** macro has two forms: the "simple" form and the "extended" form. The extended form is covered in another documentation topic, but the simple loop is useful for very basic loop.

The simple **loop** form takes a number of forms and repeats them until the loop is exited using **return** or some other exit (e.g., **throw**).

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

0
1
2
3
4
NIL
```

Read Basic loops online: https://riptutorial.com/common-lisp/topic/2053/basic-loops

Chapter 5: Booleans and Generalized Booleans

Examples

True and False

The special symbol `T` represents the value *true* in Common Lisp, while the special symbol `NIL` represents *false*:

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

They are called “Constant Variables” (sic!) in the standard, since they are variables whose value *cannot* be modified. As a consequence, you cannot use their names for normal variables, like in the following, incorrect, example:

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

Actually, one can consider them simply as constants, or as self-evaluated symbols. `T` and `NIL` are specials in other senses, too. For instance, `T` is also a type (the supertype of any other type), while `NIL` is also the empty list:

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

Generalized Booleans

Actually any value different from `NIL` is considered a *true* value in Common Lisp. For instance:

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

This fact can be combined with the boolean operators to make programs more concise. For instance, the above example is equivalent to:

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

The macro `OR` evaluates its arguments in order from left to right and stops as soon as it finds a non-NIL value, returning it. If all of them are `NIL`, the value returned is `NIL`:

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

Analogously, the macro `AND` evaluates its arguments from left to right and returns the value of the last, if all of them are evaluated to non-NIL, otherwise stops the evaluation as soon as it finds `NIL`, returning it:

```
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
```

For these reasons, `AND` and `OR` can be considered more similar to control structures of other languages, rather than to boolean operators.

Read Booleans and Generalized Booleans online: https://riptutorial.com/common-lisp/topic/3292/booleans-and-generalized-booleans

Chapter 6: CLOS - the Common Lisp Object System

Examples

Creating a basic CLOS class without parents

A CLOS class is described by:

- a name
- a list of superclasses
- a list of slots
- further options like documentation

Each slot has:

- a name
- an initialization form (optional)
- an initialization argument (optional)
- a type (optional)
- a documentation string (optional)
- accessor, reader and/or writer functions (optional)
- further options like allocation

Example:

```
(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"))
```

A default print method:

```
(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))))
```

Creating instances:

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

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

Mixins and Interfaces

Common Lisp does not have interfaces in the sense that some languages (e.g., Java) do, and there is less need for that type of interface given that Common Lisp supports multiple inheritance and generic functions. However, the same type of patterns can be realized easily using mixin classes. This example shows the specification of a collection interface with several corresponding generic functions.

```
;; 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)))
```

An implementation of the interface is just a class that has the mixin as one of its super classes, and definitions of the appropriate generic functions. (At this point, notice that the mixin class is really only for signalling the intent that the class implements the "interface". This example would work just as well with a few generic functions and documentation that states that there are methods on the function for the class.)

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

Finally, we can see what using an instance of the **sorted-set** class looks like when using the "interface" functions:

```
(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)
```

Read CLOS - the Common Lisp Object System online: https://riptutorial.com/common-lisp/topic/673/clos---the-common-lisp-object-system

Chapter 7: CLOS Meta-Object Protocol

Examples

Obtain the slot names of a Class

Lets say we have a class as

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

To obtain the names of the slots of the class we use the function class-slots. This can be found in the closer-mop package, provided by the closer-mop system. To load it the running lisp image we use `(ql:quickload :closer-mop)`. We also have to make sure the class is finalized before calling class-slots.

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

which returns a list of *effective slot definition* objects:

```
(#<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>)
```

Update a slot when another slot is modified

The CLOS MOP provides the hook slot-value-using-class, that is called when a slot is value is accessed, read or modified. Because we only care for modifications in this case we define a method for `(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))))
```

Note that because at instance creation `slot-value` is not called it may be necessary to duplicate the code in the `initialize-instance :after` method

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

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

Read CLOS Meta-Object Protocol online: https://riptutorial.com/common-lisp/topic/2901/clos-meta-object-protocol

Chapter 8: Cons cells and lists

Examples

Lists as a convention

Some languages include a list data structure. Common Lisp, and other languages in the Lisp family, make extensive use of lists (and the name Lisp is based on the idea of a LISt Processor). However, Common Lisp doesn't actually include a primitive list datatype. Instead, lists exist by convention. The convention depends on two principles:

1. The symbol **nil** is the empty list.
2. A non empty list is a *cons cell* whose *car* is the first element of the list, and whose *cdr* is the rest of the list.

That's all that there is to lists. If you've read the example called *What is a cons cell?*, then you know that a cons cell whose car is X and whose cdr is Y can be written as **(X . Y)**. That means that we can write some lists based on the principles above. The list of the elements 1, 2, and 3 is simply:

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

However, because lists are so common in the Lisp family of languages, there are special printing conventions beyond the simple dotted pair notation for cons cells.

1. The symbol **nil** can also be written as **()**.
2. When the cdr of one cons cell is another list (either **()** or a cons cell), instead of writing the one cons cell with the dotted pair notation, the "list notation" is used.

The list notation is shown most clearly by several examples:

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

The idea is that the elements of the list are written in successive order within parenthesis until the final cdr in the list is reached. If the final cdr is **nil** (the empty list), then the final parenthesis is written. If the final cdr is not **nil** (in which case the list is called an *improper list*), then a dot is written, and then that final cdr is written.

What is a cons cell?

A cons cell, also known as a dotted pair (because of its printed representation), is simply a pair of two objects. A cons cell is created by the function `cons`, and elements in the pair are extracted using the functions `car` and `cdr`.

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

For instance, this returns a pair whose first element (which can be extracted with `car`) is `"a"`, and whose second element (which can be extracted with `cdr`), is `4`.

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

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

Cons cells can be printed in *dotted pair* notation:

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

Cons cells can also be read in dotted pair notation, so that

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

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

(The printed form of cons cells can be a bit more complicated, too. For more about that, see the example about cons cells as lists.)

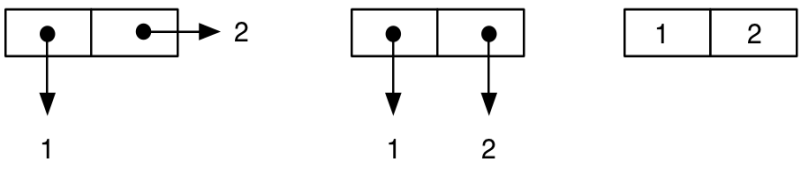
That's it; cons cells are just pairs of elements created by the function `cons`, and the elements can be extracted with `car` and `cdr`. Because of their simplicity, cons cells can be a useful building block for more complex data structures.

Sketching cons cells

To better understand the semantics of conses and lists, a graphical representation of this kind of structures is often used. A cons cell is usually represented with two boxes in contact, that contain either two arrows that point to the `car` and `cdr` values, or directly the values. For instance, the result of:

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

can be represented with one of these drawings:



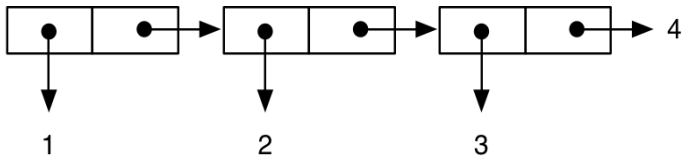
Note that these representations are purely conceptual, and do not denote the fact that the values are *contained* into the cell, or are *pointed* from the cell: in general this depends on the

implementation, the type of the values, the level of optimization, etc. In the rest of the example we will use the first kind of drawing, which is the one more commonly used.

So, for instance:

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

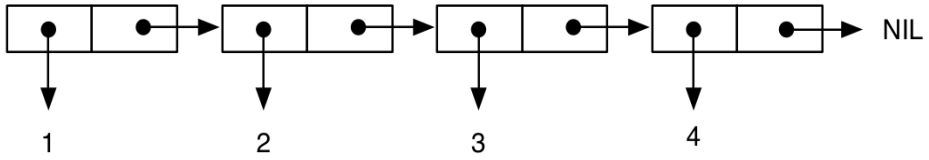
is represented as:



while:

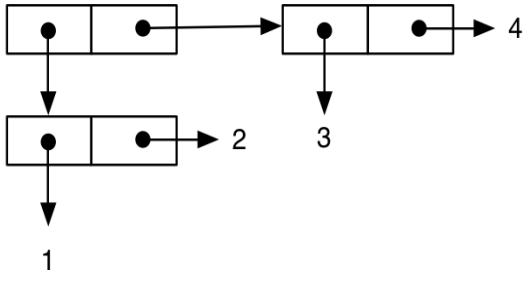
```
(cons 1 (cons 2 (cons 3 (cons 4 nil))))  ;; proper list, equivalent to: (list 1 2 3 4)
;; -> (1 2 3 4)
```

is represented as:



Here is a tree-like structure:

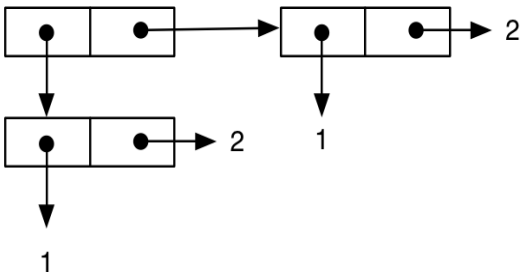
```
(cons (cons 1 2) (cons 3 4))
;; -> ((1 . 2) 3 . 4)         ; note the printing as an improper list
```



The final example shows how this notation can help us to understand important semantics aspects of the language. First, we write an expression similar to the previous one:

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

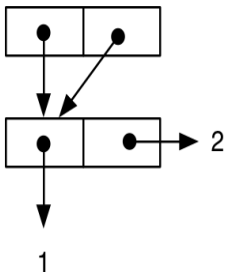
that can be represented in the usual way as:



Then, we write a different expression, which is apparently equivalent to the previous one, and this seems confirmed by printed representation of the result:

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

But, if we draw the diagram, we can see that the semantics of the expression is different, since the *same* cell is the value *both* of the `car` part and the `cdr` part of the outer `cons` (this is, `cell-a` is *shared*):



and the fact that the semantics of the two results is actually different at the language level can be verified by the following tests:

```
(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)
```

The first `eq` is *false* since the `car` and `cdr` of `c1` are structurally equal (that is *true* by `equal`), but are not “identical” (i.e. “the same shared structure”), while in the second test the result is *true* since the `car` and `cdr` of `c2` are *identical*, that is they are *the same structure*.

Read Cons cells and lists online: https://riptutorial.com/common-lisp/topic/2622/cons-cells-and-lists

Chapter 9: Control Structures

Examples

Conditional Constructs

In Common Lisp, `if` is the simplest conditional construct. It has the form `(if test then [else])` and is evaluated to `then` if `test` is true and `else` otherwise. The else part can be omitted.

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

One very important difference between `if` in Common Lisp and `if` in many other programming languages is that CL's `if` is an expression, not a statement. As such, `if` forms return values, which can be assigned to variables, used in argument lists, etc:

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

Common Lisp's `if` can be considered equivalent to the ternary operator ?: in C# and other "curly brace" languages.

For example, the following C# expression:

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

Is equivalent to the following Common Lisp code, assuming that `year` holds an integer:

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

`cond` is another conditional construct. It is somewhat similar to a chain of `if` statements, and has the form:

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

More precisely, `cond` has zero or more *clauses*, and each clause has one test followed by zero or more consequents. The entire `cond` construct selects the first clause whose test does not evaluate to `nil` and evaluates its consequents in order. It returns the value of the last form in the consequents.

```
(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!"
```

To provide a default clause to evaluate if no other clause evaluates to `t`, you can add a clause that is true by default using `t`. This is very similar in concept to SQL's `CASE...ELSE`, but it uses a literal boolean true rather than a keyword to accomplish the task.

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

An `if` construct can be written as a `cond` construct. `(if test then else)` and `(cond (test then) (t else))` are equivalent.

If you only need one clause, use `when` or `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
```

The do loop

Most looping and conditional constructs in Common Lisp are actually macros that hide away more basic constructs. For example, `dotimes` and `dolist` are built upon the `do` macro. The form for `do` looks like this:

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

- `varlist` is composed of the variables defined in the loop, their initial values, and how they change after each iteration. The 'change' portion is evaluated at the end of the loop.
- `endlist` contains the end conditions and the values returned at the end of the loop. The end condition is evaluated at the beginning of the loop.

Here's one that starts at 0 and goes upto (not including) 10.

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

And here's one that moves through a list:

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

The `varlist` portion is similar the one in a `let` statement. You can bind more than one variable, and they only exist inside the loop. Each variable declared is in its own set of parenthesis. Here's one that counts how many 1's and 2's are in a list.

```
(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)
```

And if a while loop macro hasn't been implemented:

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

For the most common applications, the more specific `dotimes` and `doloop` macros are much more succinct.

Read Control Structures online: https://riptutorial.com/common-lisp/topic/3229/control-structures

Chapter 10: Creating Binaries

Examples

Building Buildapp

Standalone Common Lisp binaries can be built with `buildapp`. Before we can use it to generate binaries, we need to install and build it.

The easiest way I know how is using `quicklisp` and a Common Lisp (this example uses [`sbcl`], but it shouldn't make a difference which one you've got).

```
$ 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
$
```

Once you have that binary built, you can use it to construct binaries of your Common Lisp programs. If you intend to do this a lot, you should also probably put it somewhere on your `PATH` so that you can just run it with `buildapp` from any directory.

Buildapp Hello World

The simplest possible binary you could build

1. Has no dependencies
2. Takes no command line arguments
3. Just writes "Hello world!" to `stdout`

After you've built `buildapp`, you can just...

```
$ 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

A more realistic example involves a project you're building with multiple files on disk (rather than an `--eval` option passed to `buildapp`), and some dependencies to pull in.

Because arbitrary things can happen during the finding and loading of `asdf` systems (including loading other, potentially unrelated systems), it's not enough to just inspect the `asd` files of the projects you're depending on in order to find out what you need to load. The general approach is to use `quicklisp` to load the target system, then call `ql:write-asdf-manifest-file` to write out a full manifest of everything that's loaded.

Here's a toy system built with `hunchentoot` to illustrate how that might happen in practice:

```
;;;; 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
```

Once you have those files saved in a directory named `buildapp-hello-web-world`, you can do

```
$ 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
```

This produces a binary that does exactly what you think it should, given the above.

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

You should then be able to fire up another shell, do `curl localhost:4242` and see the plaintext response of `Hello Web World!` get printed out.

Read Creating Binaries online: https://riptutorial.com/common-lisp/topic/5457/creating-binaries

Chapter 11: Customization

Examples

More features for the Read-Eval-Print-Loop (REPL) in a terminal

CLISP has an integration with GNU Readline.

For improvements for other implementations see: How to customize the SBCL REPL.

Initialization Files

Most Common Lisp implementations will try to load an *init file* on startup:

Implementation	Init file	Site/System Init file
ABCL	`$HOME/.abclrc`	
Allegro CL	`$HOME/.clinit.cl`	
ECL	`$HOME/.eclrc`	
Clasp	`$HOME/.clasprc`	
CLISP	`$HOME/.clisprc.lisp`	
Clozure CL	`home:ccl-init.lisp` or `home:ccl-init.fasl` or `home:.ccl-init.lisp`	
CMUCL	`$HOME/.cmucl-init.lisp`	
LispWorks	`$HOME/.lispworks`	
MKCL	`$HOME/.mkclrc`	
SBCL	`$HOME/.sbclrc`	`$SBCL_HOME/sbclrc` or `/etc/sbclrc`
SCL	`$HOME/.scl-init.lisp`	

Sample Initialization files:

Implementation	Sample Init file
LispWorks	`Library/lib/7-0-0-0/config/a-dot-lispworks.lisp`

Optimization settings

Common Lisp has a way to influence the compilation strategies. It makes sense to define your preferred values.

Optimization values are between 0 (unimportant) and 3 (extremely important). **1 is the neutral value.**

It's useful to always use safe code (safety = 3) with all runtime checks enabled.

Note that the interpretation of values is implementation specific. Most Common Lisp implementations make some use of these values.

Setting	Explanation	useful default value	useful delivery value
`compilation-speed`	speed of the compilation process	2	0
`debug`	ease of debugging	2	1 or 0
`safety`	run-time error checking	3	2
`space`	both code size and run-time space	2	2
`speed`	speed of the object code	2	3

An `optimize` declaration for use with `declaim`, `declare` and `proclaim`:

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

Note that you can also apply special optimization settings to portions of the code in a function using the macro `LOCALLY`.

Read Customization online: https://riptutorial.com/common-lisp/topic/5679/customization

Chapter 12: Equality and other comparison predicates

Examples

The difference between EQ and EQL

1. `EQ` checks if two values have the same address of memory: in other words, it checks if the two values are are actually the *same*, *identical* object. So, it is can be considered the identity test, and should be applied *only* to structures: conses, arrays, structures, objects, typically to see if you are dealing in fact with the same object "reached" through different paths, or aliased through different variables.

2. `EQL` checks if two structures are the same object (like `EQ`) *or* if they are the same non-structured values (that is, the same numeric values for numbers of the same type or the character values). Since it includes the `EQ` operator and can be used also on non-structured values, is the most important and most commonly used operator, and almost all the primitive functions that require an equality comparison, like `MEMBER`, *use by default this operator*.

So, it is always true that `(EQ X Y)` implies `(EQL X Y)`, while the viceversa does not hold.

A few examples can clear the difference between the two operators:

```
(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
```

Let's try the same examples with `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
```

From the examples we can see why the `EQL` operator should be used to portably check for “sameness” for all the values, structured and non-structured, and why actually many experts advise against the use of `EQ` in general.

Structural equality with EQUAL, EQUALP, TREE-EQUAL

These three operators implement structural equivalence, that is they check if different, complex objects have equivalent structure with equivalent component.

`EQUAL` behaves like `EQL` for non-structured data, while for structures built by conses (lists and trees), and the two special types of arrays, strings and bit vectors, it performs *structural equivalence*, returning true on two structures that are isomorphic and whose elementary components are correspondingly equal by `EQUAL`. For instance:

```
(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` returns true on all cases in which `EQUAL` is true, but it uses also structural equivalence for arrays of any kind and dimension, for structures and for hash tables (but not for class instances!). Moreover, it uses case insensitive equivalence for strings.

```
(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
```

Finally, `TREE-EQUAL` can be applied to structures built through `cons`and checks if they are isomorphic, like `EQUAL`, but leaving to the user the choice of which function to use to compare the leafs, i.e. the non-cons (atom) encountered, that can be of any other data type (by default, the test used on atom is `EQL`). For instance:

```
(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
```

Comparison operators on numeric values

Numeric values can compared with `=` and the other numeric comparison operators (`/=`, `<`, `<=`, `>`, `>=`) that ignore the difference in the physical representation of the different types of numbers, and perform the comparison of the corresponding mathematical values. For instance:

```
(= 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
```

From these examples, we can conclude that `=` is the operator that should normally be used to perform comparison between numeric values, unless we want to be strict on the fact that two numeric values are equal *only* if they have also the same numeric type, in which case `EQL` should be used.

Comparison operators on characters and strings

Common Lisp has 12 type specific operators to compare two characters, 6 of them case sensitives and the others case insensitives. Their names have a simple pattern to make easy to remember their meaning:

Case Sensitive	Case Insensitive
CHAR=	CHAR-EQUAL
CHAR/=	CHAR-NOT-EQUAL
CHAR<	CHAR-LESSP
CHAR<=	CHAR-NOT-GREATERP
CHAR>	CHAR-GREATERP
CHAR>=	CHAR-NOT-LESSP

Two characters of the same case are in the same order as the corresponding codes obtained by `CHAR-CODE`, while for case insensitive comparisons the relative order between any two characters taken from the two ranges `a..z`, `A..Z` is implementation dependent. Examples:

```
(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
```

For strings the specific operators are `STRING=`, `STRING-EQUAL`, etc. with the word STRING instead of CHAR. Two strings are equal if they have the same number of characters *and* the corresponding characters are equal according to `CHAR=` or `CHAR-EQUAL` if the test is case sensitive or not.

The ordering between strings is tje lexicographic order on the characters of the two strings. When an ordering comparison succeeds, the result is not `T`, but the index of the first character in which the two strings differ (which is equivalent to true, since every non-NIL object is a “generalized

boolean” in Common Lisp).

An important thing is that *all* the comparison operators on string accept four keywords parameters: `start1`, `end1`, `start2`, `end2`, that can be used to restrict the comparison to only a contiguous run of characters inside one or both strings. The start index if omitted is 0, the end index is omitted is equal to the length of the string, and the comparison in performed on the substring starting at character with index `:start` and terminating with the character with index `:end - 1` included.

Finally, note that a string, even with a single character, cannot be compared to a character.

Examples:

```
(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
```

As a special case, the string comparison operators can also be applied to symbols, and the comparison is made on the `SYMBOL-NAME` of the symbol. For instance:

```
(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
```

As final note, `EQL` on characters is equivalent to `CHAR=`; `EQUAL` on strings is equivalent to `STRING=`, while `EQUALP` on strings is equivalent to `STRING-EQUAL`.

Overwiew

In Common Lisp there are many different predicates for comparing values. They can be classified in the following categories:

1. Generic equality operators: EQ, EQL, EQUAL, EQUALP. They can be used for values of any type and return always a boolean value T or NIL.
2. Type specific equality operators: = and = for numbers, CHAR= CHAR= CHAR-EQUAL CHAR-NOT-EQUAL for characters, STRING= STRING= STRING-EQUAL STRING-NOT-EQUAL for strings, TREE-EQUAL for conses.
3. Comparison operators for numeric values: <, <=, >, >=. They can be applied to any type of number and compare the mathematical value of the number, independently from the actual type.
4. Comparison operators for characters, like CHAR<, CHAR-LESSP, etc., that compare

characters either in a case sensitive way or in a case insensitive way, according to an implementation depending order that preserves the natural alphabetical ordering.

5. Comparison operators for strings, like STRING<, STRING-LESSP, etc., that compare strings lexicographically, either in a case sensitive way or in a case insensitive way, by using the character comparison operators.

Read Equality and other comparison predicates online: https://riptutorial.com/common-lisp/topic/10064/equality-and-other-comparison-predicates

Chapter 13: format

Parameters

Lambda-List	`(format DESTINATION CONTROL-STRING &REST FORMAT-ARGUMENTS)`
`DESTINATION`	the thing to write to. This can be an output stream, `t` (shorthand for `*standard-output*`), or `nil` (which creates a string to write to)
`CONTROL-STRING`	the template string. It might be a primitive string, or it might contain tilde-prefixed command directives that specify, and somehow transform additional arguments.
`FORMAT-ARGUMENTS`	potential additional arguments required by the given `CONTROL-STRING`.

Remarks

The CLHS documentation for `FORMAT` directives can be found in Section 22.3. With SLIME, you can type `C-c C-d ~` to look up the CLHS documentation for a specific format directive.

Examples

Basic Usage and Simple Directives

The first two arguments to format are an output stream and a control string. Basic use does not require additional arguments. Passing `t` as the stream writes to `*standard-output*`.

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

That expression will write `Basic Message` to standard output, and return `nil`.

Passing `nil` as the stream creates a new string, and returns it.

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

Most control string directives require additional arguments. The `~a` directive ("aesthetic") will print any argument as though by the `princ` procedure. This prints the form without any escape characters (keywords are printed without the leading colon, strings without their surrounding quotes and so forth).

```
> (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` optionally right or left-pads input based on additional inputs.

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

The `~s` directive is like `~a`, but it prints escape characters.

```
> (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\""
```

Iterating over a list

One can iterate over a list using `~{` and `~}` directives.

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

`~^` can be used to escape if there are no more elements left.

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

A numeric argument can be given to `~{` to limit how many iterations can be done:

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

`~@{` will iterate over remaining arguments, instead of a list:

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

Sublists can be iterated over by using `~:{`:

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

```
(1, 2) (3, 4) (5, 6)
```

Conditional expressions

Conditional expressions can be done with `~[` and `~]`. The clauses of the expression are separated using `~;`.

By default, `~[` takes an integer from the argument list, and picks the corresponding clause. The clauses start at zero.

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

The last clause can be separated with `~:;` instead to make it the else-clause.

```
(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!
```

If the conditional expression starts with `~:[`, it will expect a generalized boolean instead of an integer. It can only have two clauses; the first one is printed if the boolean was `NIL`, and the second clause if it was truthy.

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

If the conditional expression starts with `~@[`, there should only be one clause, which is printed if the input, a generalized boolean, was truthy. The boolean will not be consumed if it is truthy.

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

Read format online: https://riptutorial.com/common-lisp/topic/687/format

Chapter 14: Functions

Remarks

Anonymous functions can be created using `LAMBDA`. Local functions can be defined using `LABELS` or `FLET`. Their parameters are defined the same was as in global named functions.

Examples

Required Parameters

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

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

Optional Parameters

Optional parameters can be specified after required parameters, by using `&OPTIONAL` keyword. There may be multiple optional parameters after it.

```
(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
```

Default alue

A default value can be given for optional parameters by specifying the parameter with a list; the second value is the default. The default value form will only be evaluated if the argument was given, so it can be used for side-effects, such as signalling an 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
```

Check if optional argument was given

A third member can be added to the list after the default value; a variable name that is true if the argument was given, or `NIL` if it wasn't given (and the default is used).

```
(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
```

Function without Parameters

Global named functions are defined with `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
```

The function body may contain any number of forms. The values from the last form will be returned from the function.

Rest Parameter

A single rest-parameter can be given with the keyword `&REST` after the required arguments. If such a parameter exists, the function can take a number of arguments, which will be grouped into a list in the rest-parameter. Note that the variable `CALL-ARGUMENTS-LIMIT` determines the maximum number of arguments which can be used in a function call, thus the number of arguments is limited to an implementation specific value of minimum 50 or more arguments.

```
(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
```

Rest and Keyword Parameters together

The rest-parameter may be before keyword parameters. In that case it will contain the property list given by the user. The keyword values will still be bound to the corresponding keyword parameter.

```
(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
```

Keyword `&ALLOW-OTHER-KEYS` can be added at the end of the lambda-list to allow the user to give keyword arguments not defined as parameters. They will go in the rest-list.

```
(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
```

Auxiliary Variables

The `&AUX` keyword can be used to define local variables for the function. They are not parameters; the user cannot supply them.

`&AUX` variables are seldomly used. You can always use `LET` instead, or some other way of defining local variables in the function body.

`&AUX` variables have the advantages that local variables of the whole function body move to the top and it makes one indentation level (for example introduced by a LET) unnecessary.

```
(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
```

One typical usage may be resolving "designator" parameters. Again, you need not do it this way; using `let` is just as idiomatic.

```
(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, exit from a block or a function

Functions always establish a block around the body. This block has the same name as the

function name. This means you can use `RETURN-FROM` with this block name to return from the function and return values.

You should avoid returning early whenever possible.

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

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

Keyword Parameters

Keyword parameters can be defined with the `&KEY` keyword. They are always optional (see the Optional Parameters example for details of the definition). There may be multiple keyword parameters.

```
(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
```

Read Functions online: https://riptutorial.com/common-lisp/topic/2126/functions

Chapter 15: Functions as first class values

Syntax

- (function name) ; retrieves the function object of that name
- #'name ; syntactic sugar for (function name)
- (symbol-function symbol) ; returns the function bound to symbol
- (funcall function args...) ; call function with args
- (apply function arglist) ; call function with arguments given in a list
- (apply function arg1 arg2 ... argn arglist) ; call function with arguments given by arg1, arg2, ..., argn, and the rest in the list arglist

Parameters

Parameter	Details
name	some (unevaluated) symbol which names a function
symbol	a symbol
function	a function which is to be called
args...	zero or more arguments (*not* a list of arguments)
arglist	a list containing arguments to be passed to a function
arg1, arg2, ..., argn	each is a single argument to be passed to a function

Remarks

When talking about Lisp-like languages there is a common distinction between what is known as a Lisp-1 and a Lisp-2. In a Lisp-1, symbols only have a value and if a symbol refers to a function then the value of that symbol will be that function. In a Lisp-2, symbols can have separate associated values and functions and so a special form is required to refer to the function stored in a symbol instead of the value.

Common Lisp is basically a Lisp-2 however there are in fact more than 2 namespaces (things that symbols can refer to) -- symbols can refer to values, functions, types and tags, for example.

Examples

Defining anonymous functions

Functions in Common Lisp are *first class values*. An anonymous function can be created by using

`lambda`. For example, here is a function of 3 arguments which we then call using `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
```

Anonymous functions can also be used directly. Common Lisp provides a syntax for it.

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

Anonymous functions can also be stored as global functions:

```
(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
```

Quoted lambda expressions are not functions

Note that quoted lambda expressions are not functions in Common Lisp. This does **not** work:

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

To convert a quoted lambda expression to a function use `coerce`, `eval` or `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>
```

Referring to Existing Functions

Any symbol in Common Lisp has a slot for a variable to be bound and a separate slot for a function to be bound.

Note that the naming in this example is only for illustration. Global variables should not be named `foo`, but `*foo*`. The latter notation is a convention to make it clear that the variable is a *special* variable using *dynamic binding*.

```
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
```

Higher order functions

Common Lisp contains many higher order functions which are passed functions for arguments and call them. Perhaps the most fundamental are `funcall` and `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)
```

There are many other higher order-function which, for example, apply a function many times to elements of a list.

```
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)
```

Summing a list

The **reduce** function can be used to sum the elements in a list.

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

By default, **reduce** performs a *left-associative* reduction, meaning that the sum 10 is computed as

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

The first two elements are summed first, and then that result (3) is added to the next element (3) to produce 6, which is in turn added to 4, to produce the final result.

This is safer than using **apply** (e.g., in **(apply '+ '(1 2 3 4))** because the length of the argument list that can be passed to **apply** is limited (see **call-arguments-limit**), and **reduce** will work with functions that only take two arguments.

By specifying the **from-end** keyword argument, **reduce** will process the list in the other direction, which means that the sum is computed in the reverse order. That is

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

is computing

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

Implementing reverse and revappend

Common Lisp already has a **reverse** function, but if it didn't, then it could be implemented easily using **reduce**. Given a list like

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

the reversed list is

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

That may not be an obvious use of **reduce**, but if we have a "reversed cons" function, say **xcons**, such that

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

Then

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

which is a reduction.

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

Common Lisp has another useful function, **revappend**, which is a combination of **reverse** and **append**. Conceptually, it reverses a list and appends it to some tail:

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

This can also be implemented with **reduce**. It fact, it's the same as the implementation of **reverse** above, except that the initial-value would need to be **(4 5 6)** instead of the empty list.

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

Closures

Functions remember the lexical scope they where defined in. Because of this, we can enclose a lambda in a let to define closures.

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

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

In the example above, the counter variable is only accessible to the anonymous function. This is more clearly seen in the following example

```
(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
```

Defining functions that take functions and return functions

A simple example:

```
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
```

The classical example of function composition: $(f \circ g \circ h)(x) = f\ (g\ (h\ (x))$:

```
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)))
```

Read Functions as first class values online: https://riptutorial.com/common-lisp/topic/1259/functions-as-first-class-values

Chapter 16: Grouping Forms

Examples

When is grouping needed?

In some places in Common Lisp, a series of forms are evaluated in order. For instance, in the body of a **defun** or **lambda**, or the body of a **dotimes**. In those cases, writing multiple forms in order works as expected. In a few places, however, such as the *then* and *else* parts of an **if** expressions, only a single form is allowed. Of course, one may want to actually evaluate multiple expressions in those places. For those situations, some kind of implicit of explicit grouping form is needed.

Progn

The general purpose special operator **progn** is used for evaluating zero or more forms. The value of the last form is returned. For instance, in the following, **(print 'hello)** is evaluated (and its result is ignored), and then **42** is evaluated and its result (**42**) is returned:

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

If there are no forms within the **progn**, then **nil** is returned:

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

In addition to grouping a series of forms, **progn** also has the important property that if the **progn** form is a *top-level form*, then all the forms within it are processed as top level forms. This can be important when writing macros that expand into multiple forms that should all be processed as top level forms.

Progn is also valuable in that it returns *all* the values of the last form. For instance,

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

In contrast, some grouping expressions only return the *primary value* of the result-producing form.

Implicit Progns

Some forms use *implicit progns* to describe their behavior. For instance, the **when** and **unless**

macros, which are essentially one-sided **if** forms, describe their behavior in terms of an *implicit progn*. This means that a form like

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

is evaluated and the condition **(foo-p foo)** is true, then the *form1* and *form2* are grouped as though they were contained within a **progn**. The expansion of the **when** macro is essentially:

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

Prog1 and Prog2

Often times, it is helpful to evaluate multiple expressions and to return the result from the first or second form rather than the last. This is easy to accomplish using **let** and, for instance:

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

Because this form is common in some applications, Common Lisp includes **prog1** and **prog2** that are like **progn**, but return the result of the first and second forms, respectively. For instance:

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

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

An important distinction between **prog1**/**prog2** and **progn**, however, is that **progn** returns *all* the values of the last form, whereas **prog1** and **prog2** only return the primary value of the first and second form. For instance:

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

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

For multiple values with **prog1** style evaluation, use **multiple-value-prog1** instead. There is no similar **multiple-value-prog2**, but it is not difficult to implement if you need it.

Block

The special operator **block** allows grouping of several Lisp forms (like an implicit `progn`) and it also takes a *name* to name the block. When the forms within the block are evaluated, the special operator **return-from** can be used to leave the block. For instance:

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

return-from can also be provided with a return value:

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

Named blocks are useful when a chunk of code has a meaningful name, or when blocks are nested. In some context, only the ability to return from a block early is important. In that case, you can use **nil** as the block name, and **return**. **Return** is just like **return-from**, except that the block name is always **nil**.

Note: enclosed forms are not top-level forms. That's different from `progn`, where the enclosed forms of a top-level `progn` form are still considered *top-level* forms.

Tagbody

For lots of control in a group forms, the **tagbody** special operator can be very helpful. The forms inside a **tagbody** form are either *go tags* (which are just symbols or integers) or forms to execute. Within a **tagbody**, the **go** special operator is used to transfer execution to a new location. This type of programming can be considered fairly low-level, as it allows arbitrary execution paths. The following is a verbose example of what a for-loop might look like when implemented as a **tagbody**:

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

```
begin
  (print (list 'hello x))
epilogue
  (incf x)
  (go prologue)
end))
```

While **tagbody** and **go** are not commonly used, perhaps due to "GOTO considered harmful", but can be helpful when implementing complex control structures like state machines. Many iteration constructs also expand into an *implicit tagbody*. For instance, the body of a **dotimes** is specified as a series of tags and forms.

Which form to use?

When writing macros that expand into forms that might involve grouping, it is worthwhile spending some time considering what grouping construction to expand into.

For definition style forms, for instance, a **define-widget** macro that will usually appear as a top-level form, and that several **defun**s, **defstruct**s, etc., it usually makes sense to use a **progn**, so that child forms are processed as top-level forms. For iteration forms, an implicit **tagbody** is more common.

For instance, the body of **dotimes**, **dolist**, and **do** each expand into an implicit **tagbody**.

For forms that define a named "chunk" of code, an implicit **block** is often useful. For instance, while the body of a **defun** is inside an implicit **progn**, that implicit **progn** is within a block sharing the name of the function. That means that **return-from** can be used to exit from the function. Such a comp

Read Grouping Forms online: https://riptutorial.com/common-lisp/topic/4892/grouping-forms

Chapter 17: Hash tables

Examples

Creating a hash table

Hash tables are created by `make-hash-table`:

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

The function may take keyword parameters to further specify the behavior of the resulting hash table:

- `test`: Selects the function used to compare keys for equality. Maybe a designator for one of the functions `eq`, `eql`, `equal` or `equalp`. The default is `eq`.
- `size`: A hint to the implementation about the space that may initially be required.
- `rehash-size`: If an integer (>= 1), then when doing a rehash, the hash table will increase its capacity by the specified number. If otherwise an float (> 1.0), then the hash table will increase its capacity to the product of the `rehash-size` and the previous capacity.
- `rehash-threshold`: Specifies how full the hash table has to be in order to trigger a rehash.

Iterating over the entries of a hash table with maphash

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

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

Using `maphash` allows to iterate over the entries of a hash table. The order of iteration is unspecified. The first argument is a function accepting two parameters: the key and the value of the current entry.

`maphash` always returns `NIL`.

Iterating over the entries of a hash table with loop

The **loop** macro supports iteration over the keys, the values, or the keys and values of a hash table. The following examples show possibilities, but the full **loop** syntax allows more combinations and variants.

Over keys and values

```
(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))
```

Over keys

```
(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)
```

Over values

```
(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)
```

Iterating over the entries of a hash table with a hash table iterator

The keys and values of a hash table can be iterated over using the macro **with-hash-table-iterator**. This may be a bit more complex than **maphash** or **loop**, but it could be used to implement the iteration constructs used in those methods. **with-hash-table-iterator** takes a name and a hash table and binds the name within a body such that successive calls to the name produce multiple values: (i) a boolean indicating whether a value is present; (ii) the key of the entry; and (iii) the value of the entry.

```
(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)
```

Read Hash tables online: https://riptutorial.com/common-lisp/topic/4482/hash-tables

Chapter 18: Lexical vs special variables

Examples

Global special variables are special everywhere

Thus these variables will use dynamic binding.

```
(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
```

Give special variables distinct names to avoid this problem:

```
(defparameter *count* 0)

(defun handle-number (number)
  (incf *count*)
  (format t "~&~d~%" number))

(dotimes (count 4)
  (handle-number count))

(format t "~&Calls: ~d~%" *count*)
==>
0
1
2
3
Calls: 4
```

Note 1: it is not possible to make a global variable non-special in a certain scope. There is no declaration to make a variable *lexical*.

Note 2: it is possible to declare a variable *special* in a local context using the `special` declaration. If there is no global special declaration for that variable, the declaration is only locally and can be shadowed.

```
(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)
```

Read Lexical vs special variables online: https://riptutorial.com/common-lisp/topic/3362/lexical-vs-special-variables

Chapter 19: LOOP, a Common Lisp macro for iteration

Examples

Bounded Loops

We can repeat an action some number of times using `repeat`.

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

Looping over Sequences

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

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

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

Here is a summary of the keywords

Keyword	Sequence type	Variable type
in	list	element of list

Keyword	Sequence type	Variable type
on	list	some cdr of list
across	vector	element of vector

Looping over Hash Tables

```
(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))
```

Simple LOOP form

Simple LOOP form without special keywords:

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

To break out of the loop we can use `(return <return value>)` `

Some examples:

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

Looping over Packages

```
(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))
```

Arithmetic Loops

```
(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
```

Destructuring in FOR statements

We can destructure lists of compound objects

```
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)
```

We can also destructure a list itself

```
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)
```

This is useful when we want to iterate through only certain elements

```
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)
```

Using `NIL` to ignore a term:

```
(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 as an Expression

Unlike the loops in nearly every other programming language in use today, the `LOOP` in Common Lisp can be used as an expression:

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

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

`MAXIMIZE` causes the `LOOP` to return the largest value that was evaluated. `MINIMIZE` is the opposite of `MAXIMIZE`.

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

`COUNT` tells you how many times an expression evaluated to non-`NIL` during the loop:

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

`LOOP` also has equivalents of the `some`, `every`, and `notany` functions:

```
(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))
```

...except they're not limited to iterating over sequences:

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

`LOOP` value-generating verbs can also be written with an -ing suffix:

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

It is also possible to capture the value generated by these verbs into variables (which are created implicitly by the `LOOP` macro), so you can generate more than one value at a time:

```
 (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)))
```

You can have more than one `collect`, `count`, etc. clause that collects into the same output value. They will be executed in sequence.

The following converts an association list (which you can use with `assoc`) into a property list (which you can use with `getf`):

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

Although this is better style:

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

Conditionally executing LOOP clauses

`LOOP` has its own `IF` statement that can control how the clauses are executed:

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

Combining multiple clauses in an IF body requires special syntax:

```
 (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)))
```

Parallel Iteration

Multiple `FOR` clauses are allowed in a `LOOP`. The loop finishes when the first of these clauses finishes:

```
(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))
```

Other clauses that determine if the loop should continue can be combined:

```
(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 which list is longer, cutting off iteration as soon as the answer is known:

```
(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)))
```

Numbering the elements of a list:

```
(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))
```

Ensure that all the numbers in a list are even, but only for the first 100 items:

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

Nested Iteration

The special `LOOP NAMED foo` syntax allows you to create a loop that you can exit early from. The exit is performed using `return-from`, and can be used from within nested loops.

The following uses a nested loop to look for a complex number in a 2D array:

```
(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))))
```

RETURN clause versus RETURN form.

Within a `LOOP`, you can use the Common Lisp `(return)` form in any expression, which will cause the `LOOP` form to immediately evaluate to the value given to `return`.

`LOOP` also has a `return` clause which works almost identically, the only difference being that you don't surround it with parentheses. The clause is used within `LOOP`'s DSL, while the form is used within expressions.

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

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

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

The thing after `finally` must be an expression, so the `(return)` form must be used and not the `return` clause:

```
 (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.
```

Looping over a window of a list

Some examples for a window of size 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
```

Read LOOP, a Common Lisp macro for iteration online: https://riptutorial.com/common-lisp/topic/1369/loop--a-common-lisp-macro-for-iteration

Chapter 20: macros

Remarks

The Purpose of Macros

Macros are intended for generating code, transforming code and providing new notations. These new notations can be more suited to better express the program, for example by providing domain-level constructs or entire new embedded languages.

Macros can make source code more self-explanatory, but debugging can be made more difficult. As a rule of thumb, one should not use macros when a regular function will do. When you do use them, avoid the usual pitfalls, try to stick to the commonly used patterns and naming conventions.

Macroexpansion Order

Compared to functions, macros are expanded in a reverse order; outmost first, inmost last. This means that by default one cannot use an inner macro to generate syntax required for an outer macro.

Evaluation Order

Sometimes macros need to move user-supplied forms around. One must make sure not to change the order in which they are evaluated. The user may be relying on side effects happening in order.

Evaluate Once Only

The expansion of a macro often needs to use the value of the same user-supplied form more than once. It is possible that the form happens to have side-effects, or it might be calling an expensive function. Thus the macro must make sure to only evaluate such forms once. Usually this will be done by assigning the value to a local variable (whose name is `GENSYM`ed).

Functions used by Macros, using EVAL-WHEN

Complex macros often have parts of their logic implemented in separate functions. One must remember, however, that macros are expanded before the actual code is compiled. When compiling a file, then by default, functions and variables defined in the same file will not be

available during macro execution. All function and variable definitions, in the same file, used by a macro must be wrapped inside an `EVAL-WHEN`-form. The `EVAL-WHEN` should have all three times specified, when the enclosed code also should be evaluated during load and runtime.

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

This does not apply to functions called from the expansion of the macro, only the ones called by the macro itself.

Examples

Common Macro Patterns

TODO: Maybe move the explanations to remarks and add examples separately

FOOF

In Common Lisp, there is a concept of Generalized References. They allow a programmer to setf values to various "places" as if they were variables. Macros that make use of this ability often have a `F`-postfix in the name. The place is usually the first argument to the macro.

Examples from the standard: `INCF`, `DECF`, `ROTATEF`, `SHIFTF`, `REMF`.

A silly example, a macro that flips the sign of a number store in a place:

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

WITH-FOO

Macros that acquire and safely release a resource are usually named with a `WITH-`-prefix. The macro should usually use syntax like:

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

Examples from the standard: `WITH-OPEN-FILE`, `WITH-OPEN-STREAM`, `WITH-INPUT-FROM-STRING`, `WITH-OUTPUT-TO-STRING`.

One approach to implementing this type of macro that can avoid some of the pitfalls of name pollution and unintended multiple evaluation is by implementing a functional version first. For instance, the first step in implementing a `with-widget` macro that safely creates a widget and cleans up afterward might be a function:

```
(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
```

Because this is a function, there are no concerns about the scope of names within **function** or **supplier**, and it makes it easy to write a corresponding macro:

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

DO-FOO

Macros that iterate over something are often named with a `DO`-prefix. The macro syntax should usually be in form

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

Examples from the standard: `DOTIMES`, `DOLIST`, `DO-SYMBOLS`.

FOOCASE, EFOOCASE, CFOOCASE

Macros that match an input against certain cases are often named with a `CASE`-postfix. There is often a `E...CASE`-variant, which signals an error if the input doesn't match any of the cases, and `C...CASE`, which signals a continuable error. They should have syntax like

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

Examples from the standard: `CASE`, `TYPECASE`, `HANDLER-CASE`.

For example, a macro that matches a string against regular expressions and binds the register groups to variables. Uses CL-PPCRE for regular expressions.

```
(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

Macros that define things are usually named either with `DEFINE-` or `DEF` -prefix.

Examples from the standard: `DEFUN`, `DEFMACRO`, `DEFINE-CONDITION`.

Anaphoric Macros

An Anaphoric Macro is a macro that introduces a variable (often `IT`) that captures the result of a user-supplied form. A common example is the Anaphoric If, which is like a regular `IF`, but also defines the variable `IT` to refer to the result of the test-form.

```
(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

Macro expansion is the process of turning macros into actual code. This usually happens as part of the compilation process. The compiler will expand all macro forms before actually compiling code. Macro expansion also happens during *interpretation* of Lisp code.

One can call `MACROEXPAND` manually to see what a macro form expands to.

```
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` is the same, but only expands once. This s useful when trying to make sense of a macro form that expands to another macro form.

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

Note that neither `MACROEXPAND` nor `MACROEXPAND-1` expand the Lisp code on all levels. They only expand the top-level macro form. To macroexpand a form fully on all levels, one needs a *code walker* to do so. This facility is not provided in the Common Lisp standard.

Backquote - writing code templates for macros

Macros return code. Since code in Lisp consists of lists, one can use the regular list manipulation functions to generate it.

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

This is often very hard to read, especially in longer macros. The Backquote reader macro allows one to write quoted templates that are filled in by selectively evaluating elements.

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

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

This version looks almost like regular code. The commas are used to evaluate `FORM`; everything else is returned as is. Notice that in `',form` the single quote is outside the comma, so it will be returned.

One can also use `,@` to splice a list in the position.

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

(macroexpand '(echo (+ 3 4)
```

```
                    (print "foo")
                    (random 10)))
;=> (PROGN
;     (FORMAT T "Form: ~a~%" (+ 3 4))
;     (FORMAT T "Form: ~a~%" (PRINT "foo"))
;     (FORMAT T "Form: ~a~%" (RANDOM 10))
;     (+ 3 4)
;     (PRINT "foo")
;     (RANDOM 10))
```

Backquote can be used outside macros too.

Unique symbols to prevent name clashes in macros

The expansion of a macro often needs to use symbols that weren't passed as arguments by the user (as names for local variables, for example). One must make sure that such symbols cannot conflict with a symbol that the user is using in the surrounding code.

This is usually achieved by using `GENSYM`, a function that returns a fresh uninterned symbol.

Bad

Consider the macro below. It makes a `DOTIMES`-loop that also collects the result of the body into a list, which is returned at the end.

```
(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)
```

This seems to work in this case, but if the user happened to have a variable name `RESULT`, which they use in the body, the results would probably not be what the user expects. Consider this attempt to write a function that collects a list of sums of all integers up to `N`:

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

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

Good

To fix the problem, we need to use `GENSYM` to generate a unique name for the `RESULT`-variable in the

macro expansion.

```
(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: How to make symbols from strings

TODO: Avoiding problems with symbols in different packages

if-let, when-let, -let macros

These macros merge control flow and binding. They are an improvement over anaphoric anaphoric macros because they let the developer communicate meaning through naming. As such their use is recommended over their anaphoric counterparts.

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

`FOO-LET` macros bind one or more variables, and then use those variables as the test form for the corresponding conditional (`IF`, `WHEN`). Multiple variables are combined with `AND`. The chosen branch is executed with the bindings in effect. A simple one variable implementation of `IF-LET` might look something like:

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

A version that supports multiple variables is available in the Alexandria library.

Using Macros to define data structures

A common use of macros is to create templates for data structures which obey common rules but may contain different fields. By writing a macro, you can allow the detailed configuration of the data structure to be specified without needing to repeat boilerplate code, nor to use a less efficient structure (such as a hash) in memory purely to simplify programming.

For example, suppose that we wish to define a number of classes which have a range of different

properties, each with a getter and setter. In addition, for some (but not all) of these properties, we wish to have the setter call a method on the object notifying it that the property has been changed. Although Common LISP already has a shorthand for writing getters and setters, writing a standard custom setter in this way would normally require duplicating the code that calls the notification method in every setter, which could be a pain if there are a large number of properties involved. However, by defining a macro it becomes much easier:

```
(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)))
```

We can now write `(defclass-notifier-slots foo (bar baz qux) (waldo))` and immediately define a class `foo` with a regular slot `waldo` (created by the second part of the macro with the specification `(waldo :accessor waldo)`), and slots `bar`, `baz`, and `qux` with setters that call the `changed` method (where the getter is defined by the first part of the macro, `(bar :reader bar)`, and the setter by the invoked `notifier` macro).

In addition to allowing us to quickly define multiple classes that behave this way, with large numbers of properties, without repetition, we have the usual benefit of code reuse: if we later decide to change how the notifier methods work, we can simply change the macro, and the structure of every class using it will change.

Read macros online: https://riptutorial.com/common-lisp/topic/1257/macros

Chapter 21: Mapping functions over lists

Examples

Overview

A set of high-level mapping functions is available in Common Lisp, to apply a function to the elements of one or more lists. They differ in the way in which the function is applied to the lists and how the final result is obtained. The following table summarize the differences and shows for each of them the equivalent LOOP form. *f* is the function to be applied, that must have a number of arguments equal to the number of lists; “applied to car” means that it is applied in turn to the elements of the lists, “applied to cdr” means that it is applied in turn to the lists, their cdr, their cddr, etc.; the “returns” column shows if the global result is the obtained by listing the results, concatenating them (so they must be lists!), or simply used for side-effects (and in this case the first list is returned).

Function	Applied to	Returns	Equivalent LOOP
(mapcar f l_1… l_n)	car	list of results	(loop for x_1 in l_1… for x_n in l_n collect (f x_1 … x_n))
(maplist f l_1… l_n)	cdr	list of results	(loop for x_1 on l_1… for x_n on l_n collect (f x_1 … x_n))
(mapcan f l_1 … l_n)	car	concatenation of results	(loop for x_1 in l_1… for x_n in l_n nconc (f x_1 … x_n))
(mapcon f l_1 … l_n)	cdr	concatenation of results	(loop for x_1 on l_1… for x_n on l_n nconc (f x_1 … x_n))
(mapc f l_1… l_n)	car	l_1	(loop for x_1 in l_1… for x_n in l_n do (f x_1… x_n) finally (return l_1))
(mapl f l_1… l_n)	cdr	l_1	(loop for x_1 on l_1… for x_n on l_n do (f x_1… x_n) finally (return l_1))

Note that, in all the cases, the lists can be of different lengths, and the application terminates when the shortest list is terminated.

Another couple of map functions are available: `map`, that can be applied to sequences (strings, vectors, lists), analogous to `mapcar`, and that can return any type of sequence, specified as first argument, and `map-into`, analogous to `map`, but that destructively modifies its first sequence argument to keep the results of the application of the function.

Examples of MAPCAR

MAPCAR is the most used function of the family:

```
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)
```

An idiomatic use of `mapcar` is to transpose a matrix represented as a list of lists:

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

For an explanation, see this answer.

Examples of 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))
```

Examples of MAPCAN and 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)
```

One of the uses of MAPCAN is to create a result list without NIL values:

```
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)
```

Examples of MAPC and 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)
```

Read Mapping functions over lists online: https://riptutorial.com/common-lisp/topic/6064/mapping-functions-over-lists

Chapter 22: Pattern matching

Examples

Overview

The two main libraries providing pattern matching in Common Lisp are Optima and Trivia. Both provide a similar matching API and syntax. However trivia provides a unified interface to extend matching, `defpattern`.

Dispatching Clack requests

Because a clack request is represented as a plist, we can use pattern matching as the entry point to the clack app as a way to route request to their appropriate controllers

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

Note: To start `*app*` we pass it to clackup. ej `(clack:clackup *app*)`

defun-match

Using pattern matching one can intertwine function definition and pattern matching, similar to 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
```

Constructor patterns

Cons-cells, structures, vectors, lists and such can be matched with constructor patterns.

```
(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
```

Guard-pattern

Guard patterns can be used to check that a value satisfies an arbitrary test-form.

```
(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!
```

Read Pattern matching online: https://riptutorial.com/common-lisp/topic/2933/pattern-matching

Chapter 23: Quote

Syntax

- (quote object) -> object

Remarks

There are some objects (for example keyword symbols) that don't need to be quoted since they evaluate to themselves.

Examples

Simple quote example

Quote is a **special operator** that prevents evaluation of its argument. It returns its argument, unevaluated.

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

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

' is an alias for the special operator QUOTE

The notation `'thing` is equal to `(quote thing)`.

The *reader* will do the expansion:

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

Quoting is used to prevent further evaluation. The quoted object evaluates to itself.

```
> 'a
A

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

If quoted objects are destructively modified, the consequences are undefined!

Avoid destructive operations on quoted objects. Quoted objects are literal objects. They are possibly embedded in the code in some way. How this works and the effects of modifications are

unspecified in the Common Lisp standard, but it can have unwanted consequences like modifying shared data, trying to modify write protected data or creating unintended side-effects.

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

Quote and self-evaluating objects

Note that many datatypes don't need to be quoted, since they evaluate to themselves. `QUOTE` is especially useful for symbols and lists, to prevent evaluation as Lisp forms.

Example for other datatypes not needed to be quoted to prevent evaluation: strings, numbers, characters, CLOS objects, ...

Here an example for strings. The evaluation results are strings, whether they are quoted in the source or not.

```
> (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")
```

Quoting for the objects thus is optional.

Read Quote online: https://riptutorial.com/common-lisp/topic/1315/quote

Chapter 24: Recursion

Remarks

Lisp is often used in educational contexts, where students learn to understand and implement recursive algorithms.

Production code written in Common Lisp or portable code has several issues with recursion: They do not make use of implementation-specific features like *tail call optimization*, often making it necessary to avoid recursion altogether. In these cases, implementations:

- Usually have a *recursion depth limit* due to limits in stack sizes. Thus recursive algorithms will only work for data of limited size.
- Do not always provide optimization of tail calls, especially in combination with dynamically scoped operations.
- Only provide optimization of tail calls at certain optimization levels.
- Do not usually provide *tail call optimization*.
- Usually do not provide *tail call optimization* on certain platforms. For example, implementations on JVM may not do so, since the JVM itself does not support *tail call optimization*.

Replacing tail calls with jumps usually makes debugging more difficult; Adding jumps will cause stack frames to become unavailable in a debugger. As alternatives Common Lisp provides:

- Iteration constructs, like `DO`, `DOTIMES`, `LOOP`, and others
- Higher-order functions, like `MAP`, `REDUCE`, and others
- Various control structures, including low-level `go to`

Examples

Recursion template 2 multi-condition

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


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

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

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

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

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

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

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

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

Recursion template 1 single condition single tail recursion

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

Compute nth Fibonacci number

```
;;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.
            )
        )
    )
)
```

Recursively print the elements of a list

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

To test this, run:

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

The result will be:

```
1
2
3
4
```

Compute the factorial of a whole number

One easy algorithm to implement as a recursive function is 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)))
        )
    )
)
```

Read Recursion online: https://riptutorial.com/common-lisp/topic/3190/recursion

Chapter 25: Regular Expressions

Examples

Using with pattern matching to bind captured groups

The pattern matching library trivia provides a system `trivia.ppcre` that allows captured groups to be bound through pattern matching

```
(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")
```

- Note: the library Optima provides a similar facility in the system `optima.ppcre`

Binding register groups with CL-PPCRE

`CL-PPCRE:REGISTER-GROUPS-BIND` will match a string against a regular expression, and if it matches, bind register groups in the regex to variables. If the string does not match, `NIL` is returned.

```
(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
```

Read Regular Expressions online: https://riptutorial.com/common-lisp/topic/2897/regular-expressions

Chapter 26: sequence - how to split a sequence

Syntax

1. split regex target-string &key start end limit with-registers-p omit-unmatched-p sharedp => list
2. lispworks:split-sequence separator-bag sequence &key start end test key coalesce-separators => sequences
3. split-sequence delimiter sequence &key start end from-end count remove-empty-subseqs test test-not key => list of subsequences

Examples

Split strings using regular expressions

The library CL-PPCRE provides the function `split` which allows us to split strings in substrings that match a regular expression, discarding the parts of the string that do not.

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

SPLIT-SEQUENCE in LIspWorks

Simple split of an IP number string.

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

Simple split of an URL:

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

Using the split-sequence library

The split-sequence library provides a function `split-sequence`, which allows to split on elements of a sequence

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

Read sequence - how to split a sequence online: https://riptutorial.com/common-

lisp/topic/1454/sequence---how-to-split-a-sequence

Chapter 27: Streams

Syntax

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

Parameters

Parameter	Detail
`stream`	The stream to read from or write to.
`eof-error-p`	Should an error be signalled if end of file is encountered.
`eof-value`	What value should be returned if eof is encountered, and `eof-error-p` is false.
`recursive-p`	Is the read-operation called recursively from `READ`. Usually this should be left as `NIL`.
`character`	The character to write, or the character that was read.
`line`	The line to write, or the line that was read.

Examples

Creating input streams from strings

The macro `WITH-INPUT-FROM-STRING` can be used to make a stream from a string.

```
(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
```

The same can be done manually using `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)))
```

Writing output to a string

The macro `WITH-OUTPUT-TO-STRING` can be used to create a string output stream, and return the resulting string at the end.

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

The same can be done manually using `MAKE-STRING-OUTPUT-STREAM` and `GET-OUTPUT-STREAM-STRING`.

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

Gray streams

Gray streams are a non-standard extension that allows user defined streams. It provides classes and methods that the user can extend. You should check your implementations manual to see if it provides Gray streams.

For a simple example, a character input stream that returns random characters could be implemented like this:

```
(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
```

Reading file

A file can be opened for reading as a stream using `WITH-OPEN-FILE` macro.

```
(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
```

The same can be done manually using `OPEN` and `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)))
```

Note that `READ-LINE` creates a new string for each line. This can be slow. Some implementations provide a variant, which can read a line into a string buffer. Example: `READ-LINE-INTO` for Allegro CL.

Writing to a file

A file can be opened for writing as a stream using `WITH-OPEN-FILE` macro.

```
(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)))
```

The same can be done manually with `OPEN` and `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))
```

Copying a file

Copy byte-per-byte of a file

The following function copies a file into another by performing an exact byte-per-byte copy, ignoring the kind of content (which can be either lines of characters in some encoding or binary data):

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

The type `(unsigned-byte 8)` is the type of 8-bit bytes. The functions `read-byte` and `write-byte` work on bytes, instead of `read-char` and `write-char` that work on characters. `read-byte` returns a byte read from the stream, or `NIL` at the end of the file if the second optional parameter is `NIL` (otherwise it signals an error).

Bulk copy

An exact copy, more efficient the the previous one. can be done by reading and writing the files with large chunks of data each time, instead of single bytes:

```
(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` and `write-sequence` are used here with a buffer which is a vector of bytes (they can operate on sequences of bytes or characters). `read-sequence` fills the array with the bytes read each time, and returns the numbers of bytes read (that can be less than the size of the array when the end of file is reached). Note that the array is destructively modified at each iteration.

Exact copy line-per-line of a file

The final example is a copy performed by reading each line of characters of the input file, and writing it to the output file. Note that, since we want an exact copy, we must check if the last line of the input file is terminated or not by an end of line character(s). For this reason, we use the two values returned by `read-line`: a new string containing the characters of the next line, and a boolean value that is *true* if the line is the last of the file and does not contain the final newline character(s). In this case `write-string` is used instead of `write-line`, since the former does not add

a newline at the end of the line.

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

Note that this program is platform independent, since the newline character(s) (varying in different operating systems) is automatically managed by the `read-line` and `write-line` functions.

Reading and writing entire files to and from strings

The following function reads an entire file into a new string and returns it:

```
(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))))
```

The result is `NIL` if the file does not exists.

The following function writes a string to a file. A keyword parameter is used to specify what to do if the file already exists (by default it causes an error, the values admissible are those of the `with-open-file` macro).

```
(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)))
```

In this case `write-sequence` can be substituted with `write-string`.

Read Streams online: https://riptutorial.com/common-lisp/topic/3028/streams

Chapter 28: Types of Lists

Examples

Plain Lists

Plain lists are the simplest type of list in Common Lisp. They are an ordered sequence of elements. They support basic operations like getting the first element of a list and the rest of a list in constant time, support random access in linear time.

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

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

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

There are many functions that operate on "plain" lists, insofar as they only care about the elements of the list. These include **find**, **mapcar**, and many others. (Many of those functions will also work on 17.1 Sequence Concepts for some of these functions.

Association Lists

Plain lists are useful for representing a sequence of elements, but sometimes it is more helpful to represent a kind of key to value mapping. Common Lisp provides several ways to do this, including genuine hash tables (see 18.1 Hash Table Concepts). There are two primary ways or representing key to value mappings in Common Lisp: property lists and association lists. This example describes association lists.

An association list, or *alist* is a "plain" list whose elements are dotted pairs in which the *car* of each pair is the key and the *cdr* of each pair is the associated value. For instance,

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

can be considered as an association list that maps symbols indicating a personal name with an integer indicating age. It is possible to implement some retrieval functions using plain list functions, like **member**. For instance, to retrieve the age of **john**, one could write

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

The **member** function returns the tail of the list beginning with with a cons cell whose *car* is **mary**, that is, **((mary . 23) (tim . 72))**, **first** returns the first element of that list, which is **(mary . 23)**, and **cdr** returns the right side of that pair, which is **23**. While this is one way to access values in an association list, the purpose of a convention like association lists is to abstract away from the

underlying representation (a list) and to provide higher-level functions for working with the data structure.

For association lists, the retrieval function is **assoc**, which takes a key, an association list and optional testing keywords (key, test, test-not), and returns the pair for the corresponding key:

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

Since the result will always be a cons cell if an item is present, if **assoc** returns **nil**, then the item was not in the list:

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

For updating values in an association list, **setf** may be used along with **cdr**. For instance, when **john**'s birthday arrives and his age increases, either of the following could be performed:

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

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

incf works in this case because it is based on **setf**.

Association lists can also be used as a type of bidirectional map, since key to value mappings be retrieved based on the value by using the reversed assoc function, **rassoc**.

In this example, the association list was created by using **list** and **cons** explicitly, but association lists can also be created by using **pairlis**, which takes a list of keys and data and creates an association list based on them:

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

A single key and value pair can be added to an association list using **acons**:

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

The **assoc** function searches through the list from left to right, which means that is is possible to "mask" values in an association list without removing them from a list or updating any of the structure of the list, just by adding new elements to the beginning of the list. The **acons** function is provided for this:

```
(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))
```

And now, a lookup for **mary** will return the first entry:

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

Property Lists

Plain lists are useful for representing a sequence of elements, but sometimes it is more helpful to represent a kind of key to value mapping. Common Lisp provides several ways to do this, including genuine hash tables (see 18.1 Hash Table Concepts). There are two primary ways or representing key to value mappings in Common Lisp: property lists and association lists. This example describes property lists.

A property list, or *plist*, is a "plain" list in which alternating values are interpreted as keys and their associated values. For instance:

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

can be considered as a property list that maps symbols indicating a personal name with an integer indicating age. It is possible to implement some retrieval functions using plain list functions, like **member**. For instance, to retrieve the age of **john**, one could write

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

The **member** function returns the tail of the list beginning with **mary**, that is, **(mary 23 tim 72)**, and **second** returns the second element of that list, that is **23**. While this is one way to access values in a property list, the purpose of a convention like property lists is to abstract away from the underlying representation (a list) and to provide higher-level functions for working with the data structure.

For property lists, the retrieval function is **getf**, which takes the property list, a key (more commonly called an *indicator*), and an optional default value to return in case the property list does not contain a value for the key.

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

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

For updating values in a property list, **setf** may be used. For instance, when **john**'s birthday arrives and his age increases, either of the following could be performed:

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

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

incf works in this case because it is based on **setf**.

To look up multiple properties in a property list as once, use **get-properties**.

The **getf** function searches through the list from left to right, which means that is is possible to "mask" values in a property list without removing them from a list or updating any of the structure of the list. For instance, using **list***:

```
(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)
```

And now, a lookup for **mary** will return the first entry:

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

Read Types of Lists online: https://riptutorial.com/common-lisp/topic/3744/types-of-lists

Chapter 29: Unit testing

Examples

Using FiveAM

Loading the library

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

Define a test case

```
(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))))
```

Run tests

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

which reports

```
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
```

Notes

- Tests are grouped by test-suites
- By defaults tests are added to the global test-suite

Introduction

There are a few libraries for unit testing in Common Lisp

- FiveAM
- Prove, with a few unique features like extensive test reporters, colored output, report of test duration and asdf integration.
- Lisp-Unit2, similar to JUnit
- Fiasco, focusing on providing a good testing experience from the REPL. Successor to hu.dwim.stefil

Read Unit testing online: https://riptutorial.com/common-lisp/topic/2349/unit-testing

Chapter 30: Working with databases

Examples

Simple use of PostgreSQL with Postmodern

Postmodern is a library to interface the relational database PostgreSQL. It offers several levels of access to PostgreSQL, from the execution of SQL queries represented as strings, or as lists, to an object-relational mapping.

The database used in the following examples can be created with these SQL statements:

```
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}');
```

The first example shows the result of a simple query returning a relation:

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

Note that the result can be returned as list of alists or plists adding the optional parameters `:alists` or `:plists` to the query function.

An alternative to `query` is `doquery`, to iterate over the results of a query. Its parameters are `query (&rest names) &body body`, where names are bound to the values in the row at each iteration:

```
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
```

When the query requires parameters, one can use prepared statements:

```
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
```

The function `prepare` receives a query with placeholders `$1`, `$2`, etc. and returns a new function that requires one parameter for each placeholder and executes the query when called with the right number of arguments.

In case of updates, the function `exec` returns the number of tuples modified (the two DDL statements are enclosed in a transaction):

```
CL-USER> (let ((parameters '("database" "dbuser" "dbpass" "localhost")))
           (pomo:with-connection parameters
             (pomo:ensure-transaction
               (values
                 (pomo:execute "alter table employees add column salary integer")
                 (pomo:execute "update employees set salary =
                                       case when skills @> '{Common Lisp}'
                                       then 100000 else 50000 end")))))
0
4
```

In addition to writing SQL queries as strings, one can use of lists of keywords, symbols and constants, with a syntax reminiscent of 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
```

Read Working with databases online: https://riptutorial.com/common-lisp/topic/4558/working-with-databases

Chapter 31: Working with SLIME

Examples

Installation

It is best to use latest SLIME from Emacs MELPA repository: the packages may be a bit unstable, but you get the latest features.

Portale and multiplatform Emacs, Slime, Quicklisp, SBCL and Git

You can download a portable and multiplatform version of Emacs25 already configured with Slime, SBCL, Quicklisp and Git: Portacle. It's a quick and easy way to get going. If you want to learn how to install everything yourself, read on.

Manual install

In GNU Emacs (>= 24.5) initialization file (`~/.emacs` or `~/.emacs.d/init.el`) add the following:

```
;; 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 on its own is OK, but it works better with Quicklisp package manager. To install Quicklisp, follow the instruction on the website (if you use roswell, follow roswell instructions). Once installed, in your lisp invoke:

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

and add the following lines to Emacs init file:

```
;; 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)))
```

After the restart, GNU Emacs will install and set up all the necessary packages.

Starting and finishing SLIME, special (comma) REPL commands

In Emacs `M-x slime` will start slime with the default (first) Common Lisp implementation. If there are multiple implementations provided (via variable `slime-lisp-implementations`), other implementations can be accessed via `M-- M-x slime`, which will offer the choice of available implementations in mini-buffer.

`M-x slime` will open REPL buffer which will look as follows:

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

SLIME REPL buffer accepts a few special commands. All of them start with `,`. Once `,` is typed, the list of options will be shown in mini-buffer. They include:

- `,quit`
- `,restart-inferior-lisp`
- `,pwd` - prints current directory from where Lisp is running
- `,cd` - will change current directory

Using 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)
```

The result that is printed after evaluation is not only a string: there is full-on Lisp object behind it which can be inspected by right-clicking on it and choosing Inspect.

Multi-line input is also possible: use `C-j` to put new line. `Enter`-key will cause the entered form to be evaluated and if the form is not finished, will likely cause an error:

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

Error handling

If evaluation causes an error:

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

This will pop up a debugger buffer with the following content (in 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 ---
```

Moving cursor down passed `--- more ---` will cause the backtrace to expand further.

At each line of the backtrace pressing `Enter` will show more information about a particular call (if available).

Pressing `Enter` on the line of restarts will cause a particular restart to be invoked. Alternatively, the restart can be chosen by number `0`, `1` or `2` (press corresponding key anywhere in the buffer). The default restart is marked by a star and can be invoked by pressing key `q` (for "quit"). Pressing `q` will close the debugger and show the following in REPL

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

Finally, quite rarely, but Lisp might encounter an error that cannot be handled by Lisp debugger, in which case it will drop into low-level debugger or finish abnormally. To see the cause of this kind of

error, switch to `*inferior-lisp*` buffer.

Setting up a SWANK server over a SSH tunnel.

1. Install a Common Lisp implementation on the server. (E.g. `sbcl`, `clisp`, etc...)
2. Install `quicklisp` on the server.
3. Load SWANK with `(ql:quickload :swank)`
4. Start the server with `(swank:create-server)`. The default port is `4005`.
5. [On your local machine] Create a SSH tunnel with `ssh -L4005:127.0.0.1:4005 [remote machine]`
6. Connect to the running remote swank server with `M-x slime-connect`. The host should be `127.0.0.1` and the port `4005`.

Read Working with SLIME online: https://riptutorial.com/common-lisp/topic/4097/working-with-slime

Credits

S. No	Chapters	Contributors
1	Getting started with common-lisp	blambert, Community, CPHPython, Dan Robertson, Ehvince, Gustav Bertram, Inaimathi, JAL, Rainer Joswig, Renzo, Robert Columbia, WarFox
2	ANSI Common Lisp, the language standard and its documentation	Rainer Joswig, sds
3	ASDF - Another System Definition Facility	Inaimathi, jkiiski, Joao Tavora, PuercoPop, Rainer Joswig, Sim, Svante
4	Basic loops	Joshua Taylor, MatthewRock, Rainer Joswig, sadfaf, Svante
5	Booleans and Generalized Booleans	Rainer Joswig, Renzo, Terje D.
6	CLOS - the Common Lisp Object System	Joshua Taylor, PuercoPop, Rainer Joswig, Sim
7	CLOS Meta-Object Protocol	PuercoPop, Rainer Joswig
8	Cons cells and lists	eyqs, Joshua Taylor, Rainer Joswig, Renzo
9	Control Structures	eyqs, Rainer Joswig, Robert Columbia, Soupy, Svante, Throwaway Account 3 Million
10	Creating Binaries	Inaimathi
11	Customization	Daniel Kochmański, Rainer Joswig
12	Equality and other comparison predicates	Renzo
13	format	Dan Robertson, Inaimathi, jkiiski, otyn, Renzo
14	Functions	jkiiski, Rainer Joswig, Svante
15	Functions as first	Dan Robertson, Joshua Taylor, PuercoPop, Rainer Joswig,

	class values	Renzo
16	Grouping Forms	Joshua Taylor, Rainer Joswig
17	Hash tables	Daniel Jour, Joshua Taylor
18	Lexical vs special variables	Rainer Joswig, Terje D.
19	LOOP, a Common Lisp macro for iteration	Dan Robertson, Elias Mårtenson, Inaimathi, PuercoPop, Rainer Joswig, RamenChef, Renzo, Throwaway Account 3 Million
20	macros	JAL, jkiiski, Joshua Taylor, Mark Green, PuercoPop, Rainer Joswig
21	Mapping functions over lists	Aaron, Rainer Joswig, Renzo
22	Pattern matching	jkiiski, PuercoPop
23	Quote	MatthewRock, Rainer Joswig, Svante
24	Recursion	4444, Rainer Joswig, Robert Columbia, sadfaf
25	Regular Expressions	jkiiski, PuercoPop
26	sequence - how to split a sequence	PuercoPop, Rainer Joswig, sadfaf
27	Streams	jkiiski, Rainer Joswig, Renzo, Svante
28	Types of Lists	jkiiski, Joshua Taylor
29	Unit testing	Ehvince, PuercoPop, Rainer Joswig, sadfaf
30	Working with databases	Renzo
31	Working with SLIME	Ehvince, mobiuseng, tsikov