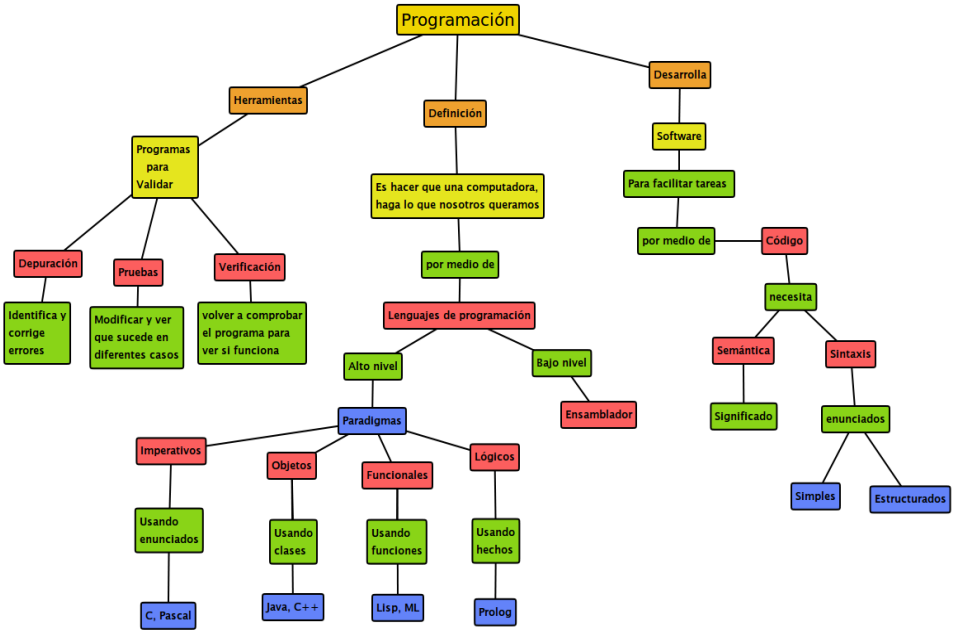
https://kimh.github.io/clojure-by-example/#about

U.T. 8 - PROGRAMACIÓ FUNCIONAL AMB Clojure



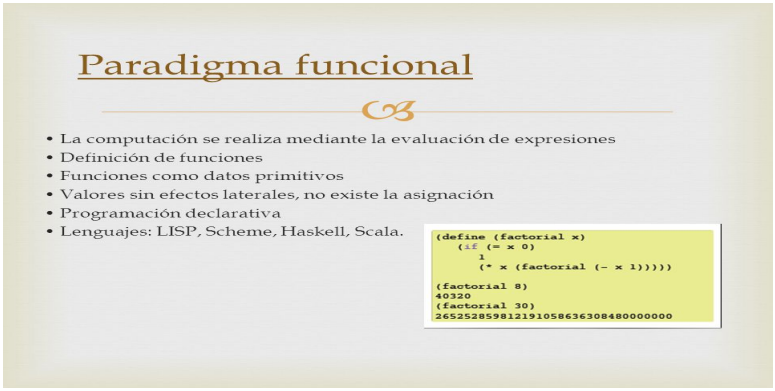
"Canviaríem el món però no tenim el codi font."

1 - INTRODUCCIÓ A LA PROGRAMACIÓ FUNCIONAL

Clojure és un llenguatge per a un paradigma diferent a la programació estructurada, com C, o a la programació orientada a objectes, com Java. C i Java, com ja sabem, són llenguatges imperatius, com la majoria de llenguatges d'ús més comú. En canvi, la programació funcional és un paradigma de programació declarativa basat en la **utilització de funcions** que eviten gestionar dades mutables o d'estat. Emfatitza en l'aplicació de funcions, en contrast amb l'estil de programació imperativa, que emfatitza els canvis d'estat de les dades. La seva sintaxi es podria descriure ...

```
(It 's (Flooded with (parenthesis)))
```

Rich Hickey, creador del llenguatge, descriu el desenvolupament de Clojure com la recerca d'un llenguatge que no va poder trobar: un Lisp funcional per defecte (Lisp se sol considerar multiparadigma), integrat sobre un entorn robust en lloc de ser la seva pròpia plataforma, i amb la programació concurrent en ment.



Com la resta de la família Lisp, la sintaxi de Clojure està construïda sobre expressions simbòliques que són convertides en estructures de dades per un lector abans de ser compilades. Les expressions es caracteritzen per estar delimitades per parèntesis, i per la seva **notació prefixa**, per la qual es crida al primer membre de cada llista com a funció, passant-li la resta de membres com a paràmetres.

```
(funcio parametre1 parametre2 ...)
```

Aquesta peculiaritat, estranya per als habituats als llenguatges més populars basats en la sintaxi del llenguatge de programació C, és la base de la seva flexibilitat. Estructures de dades com ara mapes, conjunts i vectors tenen una expressió literal; no requereixen cap transformació a l'hora d'incorporar-se a l'arbre sintàctic generat pel compilador. Clojure és basa en Lisp i no està particularment dissenyat per a ser compatible amb altres *lisps*.

Una altra particularitat a ressaltar és la seva relació amb el llenguatge Java. Clojure fa ús de la JVM (Java Virtual Machine) i les seves classes.

Però la característica fonamental a destacar d'aquest paradigma, i d'aquest llenguatge, és que les funcions a utilitzar han de ser **pures**. Què és una funció pura? És tota funció que complisca les dues condicions següents:

- ha de retornar el mateix resultat si se li passen els mateixos paràmetres (això és anomenat transparència referencial).
- no ha de produir efectes col·laterals (fonamentalment, no ha de modificar variables, fins i tot, en alguns casos, no ha d'escriure en fluxos d'eixida com fitxers o pantalla).

El compliment d'aquestes condicions, especialment el referit a la transparència referencial, fa, a les funcions que la compleixen, paralelitzables. Si més d'un fil (*thread*) crida a la mateixa funció amb els mateixos paràmetres, només un d'ells farà els càlculs (la resta ja coneixeran el resultat que produirà). Això és ideal per a la programació concurrent.

https://clojure.org

https://clojure.github.io/

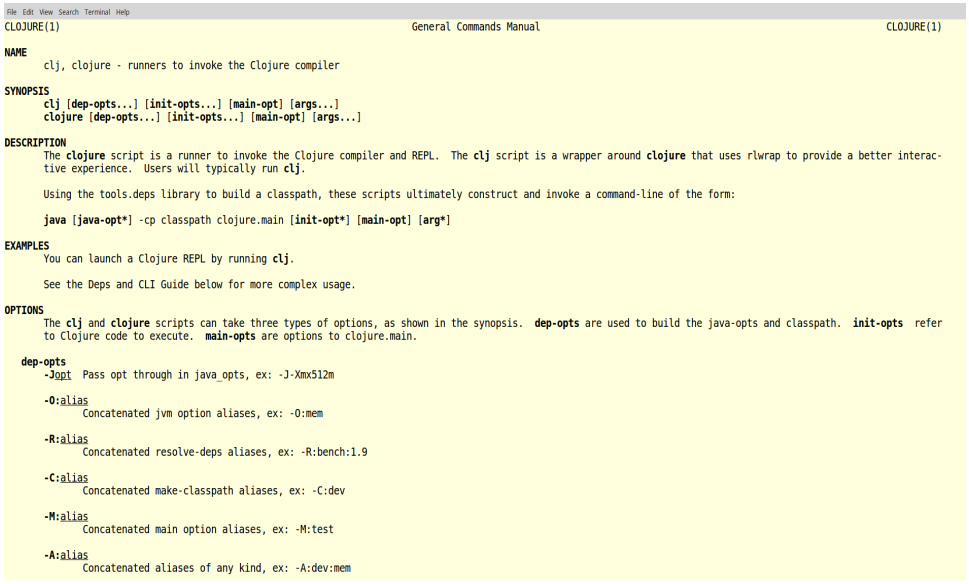
https://clojuredocs.org

2 - ENTORN DE PROGRAMACIÓ I EXECUCIÓ

El compilador i l'entorn d'execució del llenguatge poden ser descarregats i instalats amb les següents ordres:

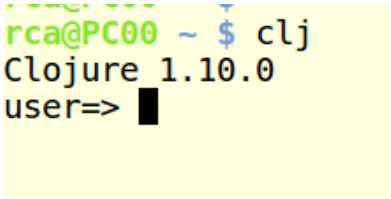
```
curl -O https://download.clojure.org/install/linux-install-1.10.0.442.sh
chmod +x linux-install-1.10.0.442.sh
sudo ./linux-install-1.10.0.442.sh
```

Els programes es poden executar des de fitxers amb extensió *.clj* amb el comandament *clojure*.



```
File  Edit  View  Search  Terminal  Help
CLOJURE(1)                                General Commands Manual                                CLOJURE(1)

NAME
       clj, clojure - runners to invoke the Clojure compiler

SYNOPSIS
       clj [dep-opts...] [init-opts...] [main-opt] [args...]
       clojure [dep-opts...] [init-opts...] [main-opt] [args...]

DESCRIPTION
       The clojure script is a runner to invoke the Clojure compiler and REPL.  The clj script is a wrapper around clojure that uses rlwrap to provide a better interac-
       tive experience.  Users will typically run clj.

       Using the tools.deps library to build a classpath, these scripts ultimately construct and invoke a command-line of the form:

       java [java-opt*] -cp classpath clojure.main [init-opt*] [main-opt] [arg*]

EXAMPLES
       You can launch a Clojure REPL by running clj.

       See the Deps and CLI Guide below for more complex usage.

OPTIONS
       The clj and clojure scripts can take three types of options, as shown in the synopsis.  dep-opts are used to build the java-opts and classpath.  init-opts  refer
       to Clojure code to execute.  main-opts are options to clojure.main.

   dep-opts
       -Jopt  Pass opt through in java_opts, ex: -J-Xmx512m

       -O:alias
              Concatenated jvm option aliases, ex: -O:mem

       -R:alias
              Concatenated resolve-deps aliases, ex: -R:bench:1.9

       -C:alias
              Concatenated make-classpath aliases, ex: -C:dev

       -M:alias
              Concatenated main option aliases, ex: -M:test

       -A:alias
              Concatenated aliases of any kind, ex: -A:dev:mem
```

L'anterior imatge està extreta de la pàgina del manual d'aquest comandament, el compilador de Clojure. En ella, podem veure que ens parla d'alguna cosa coneguda com REPL. Podem provar que, quan invoquem el compilador passant-li com a paràmetre un fitxer amb codi Clojure, l'executa i mostra l'eixida produïda. En canvi, sense paràmetres ens mostra un terminal diferenciat al de la línia de comandaments de sistema operatiu. Aquest terminal és REPL.



REPL no és més que una consola d'avaluació (en anglès, REPL: *read eval print loop*). Una altra eina fonamental per al desenvolupament de projectes amb aquest llenguatge és *Leiningen*. *Leiningen* és un gestor de projectes semblant a Maven o Ant; ens permet crear els nostres projectes Clojure, gestionar les dependències, executar una aplicació, etc ... Com ho instalem a Linux? Fàcil, obrim una consola i escrivim:

```
wget https://raw.github.com/technomancy/leiningen/stable/bin/lein
```

Li donem permís:

```
chmod +x lein
```

i després executem:

```
./lein
```

i, si volem, podem crear un projecte així:

```
lein new app my-stuff
```

3 - PRIMERS PASSOS AMB CLOJURE

Anirem veient algun dels conceptes anteriorment explicats amb alguns exemples i, al mateix temps, les estructures o funcions més utilitzades. Imaginem el següent codi per a imprimir els enters entre 1 i 5:

```
(def n 1)
(while (< n 5)
    (print n)
    (def n (inc n))
)
```

Si l'executem comprovarem que funciona correctament, però aquesta forma de programar **no és correcta** en Clojure. Sí que ho seria en el paradigma de programació imperativa; per exemple, en C o en Java. El problema és que estem modificant dades, en aquest cas la variable comptadora. En el seu lloc podríem fer alguna cosa com:

```
(loop [n 1]
    (print n)
    (recur (inc n))
)
```

Bé ... si ho provem ... no exactament. L'anterior codi produeix un bucle infinit. Anem a modificar-ho:

```
(loop [n 1]
    (print n)
    (if (< n 5)
        (recur (inc n)))
)
```

Aquesta solució sí que compleix amb el propòsit indicat: que les funcions han de ser pures, especialment en el fet que evita la modificació de variables. Aquest programa, d'aquesta manera, no modifica la variable comptadora, sinó que, cada vegada que ha de incrementar-la, genera una nova instància de la variable amb el valor anterior incrementat en una unitat.

4 - ESPAIS DE NOMS

En programació, un espai de noms (tècnica i correctament definit com *namespace*), en la seva accepció més simple, és un conjunt de noms en el qual tots els identificadors són únics. Un identificador definit en un espai de noms està associat amb aquest espai de noms. El mateix identificador pot independentment ser definit en múltiples espais de noms.

Un espai de noms també pot contenir assignacions de símbols referits a altres espais de noms. Els espais de noms també són dinàmics; es poden crear, eliminar i modificar en temps d'execució, en la REPL, etc.

La manera de definir l'espai de noms a què pertany el codi d'un fitxer font de Clojure és usar la macro *ns*. Per defecte, això crearà un nou espai de noms que contindrà assignacions per als noms de classe a *java.lang* més *clojure.lang.Compiler* i les funcions a *clojure.core*.

A la REPL, és millor utilitzar *in-ns*, en aquest cas el nou espai de noms contindrà referències només per als noms de classe a *java.lang*. Per accedir als noms de l'espai de noms clojure.core, hem d'incloure *(clojure.core/refer 'clojure.core)*. L'espai de noms d'usuari a la REPL ja ha fet això.

https://clojuredocs.org/clojure.core/in-ns

https://clojure.org/guides/repl/navigating_namespaces

Un exemple: hi han diverses funcions útils a l'hora de treballar amb cadenes de text. De fet, al estar integrat a la màquina virtual de Java, podem utilitzar des de Clojure qualsevol funció de la classe *java.lang.String*. Aquí veiem una crida a la funció *String.startsWith* per comprovar si una cadena comença per una determinada subcadena o lletra.

```
(.startsWith "Hola" "H") ; retorna true
```

Cal notar com anteposem el punt al nom de la funció. Aquesta sintaxi seria equivalent a fer *"Hola".startsWith("H")* en Java. En aquest cas, Clojure incorpora funcions pròpies per a gestió de cadenes de text (algunes d'elles solapades amb les de Java). Moltes d'elles estan incorporades a l'espai de noms *clojure.string* vist abans, encara que altres formen part del nucli central *clojure.core*. El següent exemple utilitza la classe *Math* de *java.lang*:

```
(Math/abs -4); retorna 4
```

Per a accedir, per exemple, a la funció *upper-case* de l'espai de noms *clojure.string* fariem:

```
(require '[clojure.string :as str])
(str/upper-case "hello world")
```

5 - CARACTERÍSTIQUES DEL LLENGUATGE

https://clojure.github.io/clojure/

La primera característica que anem a ressaltar del llenguatge és com afegir **comentaris** al codi. En Clojure hi han tres tipus de comentaris:

- comentaris d'una sola línia: comencen amb el punt i coma (;), i conclouen a la fi de la línia que els conté:

```
(print "Hola món"); Mostra "Hola món" per pantalla
```

És habitual trobar diversos punts i comes seguits com a inici de comentari. El nombre d'ells sol indicar el nivell del comentari (quants més tinga, més general és). Per exemple, comentaris a nivell de fitxer, a nivell de funció, i a nivell de línia o fragment de codi.

```
;;; Programa que mostra un missatge bàsic per pantalla
(ns hola-món-clojure.core)
;; Funció per a mostrar el missatge "Hola món"
(defn salutació
"Mostra hola món per pantalla"
[]
; Imprimim "Hola món"
(println "Hola món"))
```

- comentaris de diverses línies: comencen amb la macro *comment* i finalitzen amb el parèntesi de tancament corresponent. El següent comentari abasta tota la funció salutació:

```
(comment
    (defn salutacio
    "Mostra hola món per pantalla"
    []
    (println "Hola món"))
)
```

- comentaris de forma: comencen amb un coixinet i caràcter de subratllat #_ i s'empren per a comentar una forma sencera. Una **forma** és la crida a una funció amb la sintaxi *(nomFuncio param1 ...)*. Per exemple, ens poden servir per comentar un bloc sencer entre parèntesis. El següent comentari abasta tota la instrucció println "Hola món", però deixa actiu l'últim parèntesi, que és el que tanca la funció salutació:

```
(defn salutacio
"Mostra hola món per pantalla"
[]
#_ (println "Hola món"))
```

def: Aquesta forma permet definir una dada amb el seu valor associat. Recorda que en Clojure no existeix instrucció d'assignació.

Una altra alternativa per a l'associació de valors és utilitzar la funció **let**. Amb aquesta funció podem associar múltiples valors a variables (per parelles, tots inclosos entre claudàtors), i definir el codi que s'executarà amb elles. Per exemple, el següent codi inicialitza una sèrie de variables simples (*x* i *y*), i amb elles calcula una variable composta per les anteriors (z), mostrant el seu valor:

```
(let [x 1
    y 2
    z (+ x y)]
    (println z)); mostraria 3
```

https://clojuredocs.org/clojure.core/let

A diferència de *def*, amb *let* les variables deixen d'existir fora del context del propi *let*. És a dir, si intentem fer alguna cosa com això, obtindrem un error de compilació en la segona línia, al no reconèixer ja la variable x:

```
(let [x 1] (println x)); OK
(println "x és" x)
; ERROR! La variable x ja no existeix
```

Al igual que altres llenguatges com Java o C#, Clojure és sensible a majúscules i minúscules, de manera que haurem de respectar aquest aspecte tot i que, com hem vist amb anterioritat, no és habitual trobar majúscules en els noms dels elements de Clojure, sinó guions o subratllats per separar paraules.

defn: permet definir una funció. Per exemple:

```
(defn saluda
"Mostre una salutació a qui em digues"
[x]
(println "Hola" x))
```

El que fa aquesta funció és mostrar per pantalla la salutació incloent el paràmetre (x). El text que ve després del nom de la funció és simplement un text de documentació que explica què fa la funció.

Una altra característica, ja destacada, és la notació prefixa per l'ús dels seus operadors. Per exemple:

```
(+ 2 3); mostra 5
(* 3 (inc 5)); mostra 18
(< 3 4); mostra true
```

Això, que pot semblar una complexitat innecessària en un principi, té un sentit molt clar. Com la resta de la família Lisp, la sintaxi de Clojure està construïda sobre expressions simbòliques que són convertides en estructures de dades per un lector abans de ser compilades. Les expressions, com estem veient, es caracteritzen per estar delimitades per parèntesis, i per aquesta notació prefixa, per la qual es crida al primer membre de cada llista com a nom de funció, passant-li la resta de membres com a arguments.

Aquesta peculiaritat, estranya per als habituats als llenguatges més populars, és la base de la seva flexibilitat.

A continuació, tot i no ser un llenguatge imperatiu, anirem fent alguns exercicis d'algorismia que ens permeten exercitar la sintaxi del llenguatge. En ells, haurem de fer lectura de valors per teclat. Per a això utilitzarem les funcions **read-line** o **read**.

https://clojuredocs.org/clojure.core/read-line

6 - CONTROL DE FLUX

Les instruccions de control de flux són, bàsicament, les alternatives i els bucles. Pel que portem dit, és previsible que aquestes estructures diferisquen respecte de les ja conegudes dels llenguatges imperatius. Vegem algunes d'elles.

do. L'expressió *do* permet agrupar un conjunt d'instruccions perquè s'executen en bloc. S'utilitza dins d'altres elements que només permeten executar una instrucció, com ara *if*, per poder realitzar més d'una tasca. Per exemple, aquest bloc engloba dues instruccions println per imprimir dos missatges:

```
(do
    (println "Hola")
    (println "Bon dia"))
```

if: L'expressió if permet executar una instrucció (o conjunt d'instruccions agrupades amb do) si es compleix una determinada condició, i una altra instrucció (o conjunt) si no es compleix. La seva sintaxi és:

```
(if (condicio)
    (InstruccionsSiTrue)
    (InstruccionsSiFalse)); les instruccionesFalse són opcionals
```

exemple:

```
(if (< hora 12)
    (do
        (println "Hola")
        (println "Bon dia"))
    (do
        (println "Què tal?")
        (println "Bona vesprada")))
```

Cal tindre en compte que el valor *nil* (retornat per exemple per print o println) és avaluat sempre a false i que, en canvi, qualsevol altre valor és avaluat a true:

```
if (nil ....) ; sempre serà avaluat com a false
if ("blablabla" ...) ; qualsevol cosa diferent a nil serà avaluat a true
```

when: Una alternativa al bloc *if* és utilitzar *when*. El funcionament és el mateix, però només permet especificar què fer si es compleix la condició (no hi ha un cas "else"). A més, en el cas de l'estructura *when* no cal utilitzar *do* per agrupar les instruccions a executar, ja que la incorpora de forma implícita: tot el que vaja dins del bloc *when* s'executa si es compleix la condició. Un exemple:

```
(when (< hora 12)
    (println "Hola")
    (println "Bon dia"))
```

case: és l'equivalent a la tradicional *switch..case* d'altres llenguatges. Li indiquem a continuació el valor a avaluar, i després cadascun dels casos. Cada cas està format per un valor i una forma a avaluar (o diverses, agrupades per *do*). L'últim cas pot ser un cas defecte si no es compleix cap dels anteriors, i en aquest cas només s'indica la forma (o formes) a avaluar. Per exemple:

```
(case dada
    0 (println "Zero")
    1 (println "Un")
    (println "No és un bit"))
```

Cal tenir en compte que els casos a avaluar (0 i 1, en l'exemple anterior), només poden ser dades constants. No es poden avaluar expressions (comparacions amb variables, per exemple).

cond: és semblant a *case* però permet avaluar expressions:

```
(cond
    (> nota 10) (println "Nota massa alta")
    (< nota 0) (println "Nota massa baixa")
    :else (println "Nota correcta"))
```

Anem ara amb els bucles.

while: tenim un exemple en l'apartat 3.

doseq: aquesta estructura equival al *foreach* d'altres llenguatges, i s'utilitza per iterar sobre una seqüència de dades. Per exemple, el següent bucle imprimeix el valor de la dada n, que prendrà els valors de la seqüència 0, 1 i 2:

```
(doseq [n [0 1 2]]
(println n))
```

dotimes: permet repetir un conjunt d'instruccions un nombre determinat de vegades, de manera que equival al tradicional for. El següent bucle imprimeix "Hola" un total de 5 vegades:

```
(dotimes [n 5]
    (println "Hola"))
```

Si volguerem imprimir el valor de n, mostraria els valors del 0 al 4.

loop-recur: finalment, aquesta estructura es fa servir per assignar un valor inicial a una dada (o dades) i repetir un conjunt d'instruccions, fins a aconseguir un cas base. El següent codi mostra els números parells del 2 al 10 (inclusivament):

```
(loop [n 2]
    (when (<= n 10)
        (println n)
        (recur (+ n 2))))
```

El que es fa és associar inicialment a la variable n el valor 2, i després amb la instrucció *when* avaluar la condició a complir (que n seguisca sent menor o igual que 10), i cridar de nou al bucle (amb *recur*), incrementant el valor en 2 unitats.

Per al control de flux és important tenir en compte també com fer les operacions lògiques *and*, *or* i *not*. Per a això tenim les següents funcions que retornen el corresponent booleà:

```
(and (< 3 4) (> 4 8)); retorna false
(or (< 3 4) (> 4 8)); retorna true
(not (< 3 4)); retorna false
(= 3 4) ; retorna false
(not= 3 4) ; retorna true
```

7 - FUNCIONS

Anem a citar algunes funcions útils, predefinides en Clojure o importades de Java.

Per a maneig de cadenes de text: moltes d'elles estan incorporades a l'espai de noms *clojure.string*, encara que altres formen part del nucli central *clojure.core*. Per exemple: *str*, *compare*, *count*, etc.

A l'espai *clojure.string* tenim entre altres: *lower-case* i *upper-case*, *join*, *split*, *reverse*, *replace*, *trim* (i *triml* i *trimr*, que només netegen l'extrem dret i esquerre de la cadena) ...

Altres funcions útils: algunes d'elles (per exemple, les funcions matemàtiques) s'utilitzen directament des del repositori de funcions de Java (classe *java.lang.Math*). Només un exemple: *pow* és importada des de Java, ens permet calcular potències de nombres. Per exemple:

```
(Math/pow 5 3) ; retorna 125.0
```

Pròpies de Clojure tenim també: *rem* (residu de la divisió entera) i *mod* (molt pareguda, però no igual. Consulta en Internet les diferències entre una i altra); *max* i *min*, *rand* (genera un nombre real entre 0 i 1, o entre 0 i el número que li passem com a paràmetre), o *rand-int* (genera un nombre enter entre 0 i el nombre rebut com a paràmetre). Un altre grup de funcions útils són les que ens permeten convertir entre diferents tipus de dades simples (per exemple, de sencer a real o viceversa). Per a això tenim *int*, *float*, *double*, ... Convé tenir en compte que, en aquest conjunt de funcions, queden exclosos tipus més complexos, com les cadenes de text (no podem usar, per exemple, la funció *int* per convertir "123" a nombre sencer). Per a això, podem utilitzar, com hem vist abans, les corresponents funcions de conversió de Java (*Integer.parseInt*, en aquest cas).

Una altra funció a destacar és **doc**. Per exemple:

```
(doc doc); mostra la forma d'ús de la macro doc
(doc range); mostra la forma d'ús de la funció range
```

Totes les funcions en Clojure retornen un valor. No cal (és més, no és possible) indicar explícitament quin valor es retorna, com passa amb la instrucció return en molts llenguatges. El valor retornat en el cas de Clojure vindrà donat per **l'última instrucció** que s'execute en el codi de la funció. Si és una operació de suma, per exemple, tornarem el resultat de la suma. Si és una comprovació, retornarem un valor booleà. Si és una instrucció d'impressió (*println*, per exemple), es retornarà *nil* com a resultat.

Les **funcions variàdiques** són aquelles que poden cridar-se amb qualsevol nombre d'arguments. A més, poden contemplar en el seu codi diferents casos segons els arguments que reben. D'aquesta manera, es permet la sobrecàrrega d'una mateixa funció amb una sola definició del seu codi. Imaginem el següent cas: hem definit tres funcions diferents, amb noms diferents, per poder saludar 0, 1 i 2 persones. Podríem fer una única funció *salutació* que contemplara aquestes tres (o més) possibilitats. Una manera de resumir les tres funcions anteriors en una sola és aquesta:

```
(defn salutacio
"Mostra una salutació als noms que rep com a paràmetres"
([] (println "Hola món"))
([nom] (println "Hola " nom))
([nom nom2] (println "Hola " nom "i" nom2)))
```

En aquest anterior cas, seguim tenint la limitació de no poder saludar més de dues persones. La millor opció és afegir un cas per admetre qualsevol nombre d'arguments. Això es fa anteposant el símbol & a l'argument que pot ser variable (podent ser el primer, el segon ... etc.). Així, per exemple, definiríem un cas per saludar 0 noms, a un nom o qualsevol nombre més gran que 1):

```
(defn salutacio
"Mostra una salutació als noms que rep com a paràmetres"
```

```
([] (println "Hola món"))
([nom] (println "Hola " nom))
([nom & resta] (println "Hola " nom "i"
    (clojure.string/join "i" resta)))
```

Amb això, podem cridar a la mateixa funció amb tants arguments com vulguem:

```
(salutacio)
(salutacio "Nacho")
(salutacio "Nacho" "Ana" "Joan" "Enrique")
```

La recursivitat és una eina potent en la programació funcional, ja que permet expressar càlculs relativament complexos d'una forma curta i (de vegades) senzilla. Hi han diverses formes d'expressar la recursivitat en Clojure, començant per la tradicional de cridar a la pròpia funció. El cas típic del factorial, per exemple, podríem expressar-ho així:

```
(defn factorial [n]
    (if (> n 1)
        (* n (factorial (- n 1)))
        1))
```

No obstant això, si consultem documentació en línia, es recomana utilitzar la instrucció *loop..recur* per provocar recursivitat en Clojure. El mateix exemple anterior quedaria d'aquesta forma:

```
(defn factorial [n]
"en aquesta versió la funció només admiteix un paràmetre"
    (loop [numeroActual n total 1]
        (if (= numeroActual 1)
            total
            (recur (dec numeroActual) (* total numeroActual)))))
```

També es pot utilitzar *recur* sense necessitat de *loop*, en el cas que la funció ja tinga tots els paràmetres que necessita. En l'exemple anterior no era així, perquè necessitàvem un acumulador per als productes parcials (total). Però podríem redefinir la funció així:

```
(defn altre-factorial [n total]
"ara la funció admet 2 paràmetres"
    (if (=  n 1)
        total
        (recur (dec n) (* total n))))
```

i, en aquest cas, la crida a un recurs implicaria cridar a la funció original directament. A l'hora de cridar a aquesta funció des de fora, ho hauríem de fer amb dos paràmetres en lloc d'un:

```
(factorial 5)
(altre-factorial 5 1)
```

Aquesta manera d'aplicar recursivitat amb la funció *recur* s'anomena recursivitat per cua (*tail recursion*), i és un concepte molt emprat en la programació funcional. Es denomina així a causa que aquest tipus de recursivitat realitza la crida recursiva a la fi de la funció (és a dir, la crida recursiva és l'últim que es fa en el codi), i en certs llenguatges com Clojure això suposa certs avantatges pel que fa a consum de recursos. Per exemple, quan les crides recursives comencen a ser grans, o el conjunt de dades a manipular ho és, la recursivitat per cua sol admetre majors quantitats de crides o dades que la normal.

8 - COL·LECCIONS

Clojure incorpora diversos tipus de col·leccions (conjunts de dades), útils a l'hora de treballar amb grups de dades de diferents característiques. Totes elles són **immutables**, és a dir, no podem canviar el valor dels seus elements. Sí podrem crear còpies de la col·lecció original, amb altres elements modificats a partir dels originals.

Tots els tipus de col·leccions en Clojure poden emmagatzemar qualsevol tipus de dada, i dades de diferents tipus. Així, per exemple, podem tenir un vector que emmagatzeme alhora sencers i cadenes, o una llista amb nombres reals i vectors d'enters. També veurem que hi han múltiples funcions que treballen amb qualsevol tipus de col·lecció, sense importar quin; és el que s'anomena abstracció de seqüències.

Mapes (també anomenats *diccionaris* o *taules hash*). Es fan servir per associar una sèrie de claus amb els seus determinats valors:

```
{ :un "one" :dos "two" :vermell "red" :blau "blue"} ; mapa o "taula hash"

{ "un" "one" "dos" "two" "vermell" "red" "blau" "blue"}; equivalent a l'anterior

(def contactes
{
"611223344" { :nom "Juan Pérez" :edat 36 :adreça "C / L'Horta 23"}
"655667788" { :nom "Luisa López" :edat 29 :adreça "C / Novelda 34"}
})
(println (get (get contactes "611223344") :nom )) ; mostra "Juan Pérez"

(merge { :un "one" :dos "two"}
    { :dos "2" :3 "3"}) ; retorna { :un "one" :dos "2" :3 "3"}
```

Els **vectors** són col·leccions de dades indexades (començant en la posició 0). Aquestes dades es representen entre claudàtors, i poden ser de diferents tipus. Es poden crear directament posant les dades entre claudàtors, o amb la funció vector passant-li les dades del vector com a paràmetres:

```
[ 1 2 3]
(vector 1 2 3) ; equivalent a l'anterior
( [1 2 3] 1) ; tornaria 2
( get [1 2 3] 0); retornaria 1, la funció get serveix per vectors, però no per llistes.
```

Les **llistes** són semblants als vectors, quant a que són col·leccions indexades de dades, que poden ser heterogènies. Es representen entre parèntesis. Per crear-les, podem definir-les amb els mateixos parèntesis precedits d'una cometa simple (per diferenciar-les d'una forma, i així impedir que s'avaluen directament). També podem crear-les amb la funció *list*, passant-li com a paràmetres les dades de la llista. Aquestes dues instruccions són equivalents:

```
'(1 2 3 4)
(list 1 2 3 4)
```

Qualsevol llista que Clojure avalue s'interpreta com una crida a una funció, on el primer element de la llista és la funció a cridar, i la resta són els arguments de la funció. Per tant, si escrivim això en un terminal REPL:

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

Clojure l'avalua com una llista amb els elements "funció suma", 1, 2 i 3, però no fa res, només retorna la llista sense més. Però si passem aquesta mateixa llista com a argument a la funció **eval** (que s'encarrega de forçar l'avaluació del que li passa), llavors es fa la suma i es retorna 6 en aquest cas.

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

A l'hora d'afegir elements a la llista o al vector comptem amb certes funcions (com la funció *conj* que veurem després), però es fa de forma diferent: els elements s'afegeixen al principi, en el cas de les llistes, i al final, en el cas dels vectors. Altres diferències estan en el rendiment segons el tipus d'operació a realitzar:

- Els vectors són apropiats per a l'accés a posicions concretes, però és més costós modificar-los.
- Les llistes s'implementen com llistes enllaçades, de manera que el temps requerit per a accedir a un element concret no és constant (cal recórrer la llista), però modificar-les és menys costós.

Entre les moltes funcions per a llistes i vectors anem a destacar algunes amb els següents exemples:

```
(range 5) ; Retorna (0 1 2 3 4)
(range 0 10) ; Retorna (0 1 2 3 4 5 6 7 8 9)
(range 0 10 2) ; Retorna (0 2 4 6 8)
(seq [1 2 3]) ; Retorna (1 2 3)
(vec '(1 2 3)) ; Retorna [1 2 3]
(iterate inc 1) ; Tornaria (1 2 3 4 5 ...)
```

La funció *iterate* genera una seqüència infinita aplicant una funció a un determinat valor. És una funció que no té molta utilitat per si mateixa, però sí combinada amb altres com la funció *take*. Com a alternativa, tenim també la funció *repeatedly*, que rep com a argument una funció a la qual cridar repetidament. Per exemple, així generaríem una seqüència infinita de nombres enters aleatoris del 0 al 9:

```
(repeatedly (fn [] (rand-int 10)))
```

Observa a l'anterior exemple com estem utilitzant una funció anònima, a la qual no donem nom ja que és d'un únic ús.

Els **conjunts** són col·leccions de dades no ordenades, però on es garanteix que no hi han elements repetits. Es representen entre claus, precedides per un coixinet. Podem comprovar com l'ordre no és emmagatzemat, simplement definint un conjunt:

```
#{1 2 3 4}; Retorna #{1 4 3 2}
```

Si tractem d'afegir un element repetit a un conjunt, l'operació no tindrà cap efecte.

Algunes funcions per a la creació de conjunts són:

- La funció *set* permet definir un conjunt a partir d'una altra col·lecció:

```
(set [1 2 3 4 1]); Retorna #{1 4 3 2}
```

- La funció *hash-set* permet definir un conjunt indicant un a un els seus elements:

```
(hash-set 1 2 3 4 1); Retorna #{1 4 3 2}
```

En els apartats anteriors hem vist algunes funcions específiques per treballar amb cada tipus concret de col·lecció. Així, per exemple, la funció *range* s'aplica a vectors i llistes, per obtenir o generar de forma ràpida una seqüència entre uns límits indicats. No obstant això, hi ha algunes funcions de maneig de col·leccions que es poden aplicar a tots els tipus vistos abans (o gairebé tots, segons el cas).

- Les funcions **cons** i **conj** permeten afegir un element a una col·lecció. Així afegiríem el número 5 a la llista formada pels números de l'1 al 4:

```
(conj '(1 2 3 4) 5); Retorna (5 1 2 3 4)
(cons 5 '(1 2 3 4)); Retorna (5 1 2 3 4)
```

La funció *cons* (s d'*start*, inici) sempre afegeix l'element al principi de la col·lecció, i retorna una llista com a resultat. Per contra, usant *conj* respectem el tipus de col·lecció d'entrada, i depenent d'aquest tipus, el nou element s'afegeix en un lloc o un altre. Per exemple, en el cas de les llistes, els elements s'afegeixen al principi, però en el cas de vectors, s'afegeixen al final:

```
(conj [1 2 3 4] 5); Retorna un vector amb [1 2 3 4 5]
(cons 5 [1 2 3 4); Retorna una llista amb (5 1 2 3 4)
```

La funció *concat* s'empra per unir dues col·leccions en una, afegint a la fi de la primera el contingut de la segona:

```
(concat [1 2] [2 3 4]); Retorna (1 2 2 3 4)
```

La funció *count* permet obtenir quants elements té una col·lecció:

```
(count [1 2 3 4]); retorna 4
```

Pots provar tu mateix la utilitat de les funcions *first*, *second*, *last*, *nth*, *reverse*, *rest*, *next*.

La funció *contains?* permet comprovar si una col·lecció conté un determinat element. Per exemple, així comprovaríem si el nombre 3 està contingut dins d'aquest vector:

```
(contains? [1 2 3 4] 3); retorna true
```

La funció *take* pren el nombre d'elements indicat de la col·lecció indicada, començant pel principi. Si el nombre d'elements de la col·lecció és insuficient, obté tots els elements possibles. Sol aplicar juntament amb la funció *iterate* per obtenir els n primers nombres d'una seqüència:

```
(take 2 [1 2 3 4]); Retorna (1 2)
(take 4 [2 1]); Retorna (1 2)
(take 5 (iterate inc 1)); Retorna (1 2 3 4 5)
```

La funció *drop* fa el contrari a *take*: descarta els n primers elements de la col·lecció, i es queda amb la resta:

```
(drop 2 [1 2 3 4]); Retorna (3 4)
```

Ordenant col·leccions: **sort** i **sort-by**. La funció *sort* s'empra per ordenar una col·lecció en ordre ascendent. Depenenent del tipus de colecció (numèrica, alfabètica...) utilitzarem un criteri o altre. Per exemple, per a ordenar una llista d'enters usarem ordre numèric, però per a ordenar un vector de noms, utilitzariem ordre alfabètic:

```
(sort [4 1 2 3]) ; Retorna (1 2 3 4)
(sort ["Nacho" "Ana" "Joan"]) ; Retorna ("Ana" "Joan" "Nacho")
```

Si volem definir el criteri d'ordenació de la colección (quan no siga algo tan trivial com ordre numèric o alfabètic) podem utilitzar la funció *sort-by*, que afegeix un nou argument que és la funció a emprar per a discernir l'ordre dels elements. Per exemple, si tenim un vector de dades personals como el següent i volem ordenar-lo per edat de forma ascendent, podem utilitzar *sort-by*:

```
(def dades-personals [{:nom "Nacho" :edat 39}{:nom "Ana" :edat 35}
{:nom "Joan" :edat 70}])
(sort-by :edat dades-personals)
```

9 - MACROS

Les macros són similars a les funcions, però permeten modificar el comportament del propi llenguatge, Clojure, de manera que no és habitual fer en altres llenguatges. Podem dir que les macros ens permeten estendre el LLENGUATGE a voluntat.

Vegem un exemple. Com ja hem vist, *when* té la següent sintaxi:

```
(when boolean-expression
  expressio-1
  expressio-2
  expressio-3
...
  expressio-x)
```

Podríem pensar que *when* és una forma característica del LLENGUATGE com ho és *if*. Però no. Es tracta d'una macro. Anem a expandir aquesta macro per veure com està realment implementada:

```
(macroexpand '(when boolean-expression
                    expression-1
                    expression-2
                    expression-3))
; mostraria =>

 (if boolean-expression
           (do expression-1
               expression-2
               expression-3))
```

Això ens mostra que les macros són part fonamental del desenvolupament de Clojure, de manera que són utilitzades fins i tot per operacions fonamentals. Nosaltres podem crear les nostres pròpies macros per adoptar el llenguatge a la nostra sintaxi preferida.

La definició d'una macro és molt similar a la de les funcions. En la seva definició s'inclou el nom, un text descriptiu, una llista de paràmetres admesos, i un cos. El cos gairebé sempre retornarà una llista. Això té sentit perquè les macros són una forma de transformar una estructura de dades en una forma que Clojure puga avaluar, i Clojure utilitza les llistes per representar les cridades a les funcions, a les formes especials (com *if*) i a les macros. Podem utilitzar qualsevol funció, macro o forma especial dins el cos de la macro, i utilitzem o fem una crida a la macro com ho faríem amb una funció o una forma especial.

Com a exemple vegem la macro infixa:

```
(defmacro infix
  "Utilitza aquesta macro per a tornar a la notación infixa"
  [infixed]
  (list (second infixed) (first infixed) (last infixed))
)
```

Como pots veure, aquesta macro reorganitza la llista en l'ordre correcte per a la notació infixa. La seua forma d'ús seria:

```
(infix (1 + 1)) ; => 2
```

Una diferència clau entre funcions i macros és que els arguments de les funcions són totalment avaluats abans del seu pas a la funció, mentre que les macros els reben com a dades inevaluades. Pots observar això en l'exemple anterior. Si intentàrem avaluar (1 + 1) tal qual, obtindríem una excepció. No obstant això, al tractar-se d'una crida a macro, la llista (1 + 1) se li passa a la macro sense avaluar. Així la macro pot utilitzar els mètodes *first*, *second*, i *last* per reorganitzar la llista de manera que Clojure la puga avaluar:

```
(macroexpand '(infix (1 + 1)))  ; => (+ 1 1)
```

Un altre exemple:

```
(defmacro repeteix
 [x]
         (list 'do
          x
          x
         )
)
```

Aquesta macro repeteix 2 vegades el codi proporcionat. Observa la cometa simple abans del *do*, és necessària per a evitar que s'avalue. Un exemple d'ús seria:

```
(repeteix (println "a"))
```

```
(macroexpand '(repeteix (println "a")))
```