INTRODUCERE ÎN PROGRAMAREA CU ELIXIR

Dănuț Chindriș

@danutchindris

AGENDA

- Ce este Elixir?
- De ce să învățăm Elixir?
- Erlang VM
- Instalare
- Shell-ul interactiv IEx
- Noțiuni de bază
- Elemente de programare funcțională în Elixir
- Build tool-ul Mix
- Framework-ul Phoenix

CE ESTE ELIXIR?

Un limbaj de programare

- dinamic
- funcțional

Un limbaj de programare bazat pe Erlang VM

- concurență *lightweight*, fin granulată
  - bazată pe *actor model* (Actor Model Explained)
- tolerant la erori (**"let it fail"**)
  - procese *lightweight*
  - supervizare
- disponibil tot timpul (*zero downtime*)
  - *upgrade* în timp ce sistemul merge
  - disponibilitate în 99.9999% din timp

CARE ESTE BAZA PENTRU ACESTE TRĂSĂTURI ALE SISTEMULUI?

PROGRAMAREA FUNCȚIONALĂ

- un mod simplu de a scrie programe
  - date
  - funcții
- nu avem "stare" ce trebuie actualizată
- avem *valori* care sunt calculate
  - **doar** valori
  - fără pointeri, referințe etc.
- fără efecte secundare
  - date imutabile
  - algoritmi mai clari

IMUTABILITATEA

- *safety* (foarte important și pentru limbajele imperative)
- ai nevoie de un nou "obiect"? nu modifici starea, creezi unul nou
- în Elixir (și Erlang) nu avem variabile; avem valori
- nu avem asignare (numele "variabilelor" sunt doar etichete legate de valori)

THREAD-SAFETY

- *safe caching*
- *safe sharing*
- consecvență

FP SUSȚINE TIPARE DE ABSTRACTIZARE DE NIVEL ÎNALT

- map
- filter
- reduce

ELIXIR ESTE PRAGMATIC

- permite unele "efecte" (ex: modul în care este gestionată comunicarea - un "efect" al evaluării)
- nu permite alte "efecte" (cum ar fi actualizările de stare, prin asignare)

ELIXIR ESTE UN LIMBAJ MODERN

FUNCȚIONAL

Părerea lui *Uncle Bob*:

> *...the future is looking very functional to me.*

DE CE SĂ ÎNVĂȚĂM ELIXIR?

PROBLEMA SISTEMELOR DISTRIBUITE

- Sisteme multi-core
- Sisteme multi-node

Joe Armstrong - How We Program Multicores

ELIXIR ESTE DISTRIBUIT

- Avem două noduri:

```
$ pi@raspberrypi-v3
$ danut@dc.local
```

- Definim funcționalitate în unul dintre ele
- Apelăm funcționalitatea din celălalt nod
- Nodurile știu să comunice *out of the box* 😀

COMPANII CARE FOLOSESC ELIXIR

- **Pinterest**: half the servers, 10 times less code
- **Moz**: 63 times less disk space, 20 times faster API
- **Lonely Planet**: better performance, scalable content
- **Financial Times**: easy to learn, quick to scale
- **Toyota Connected**: mobility at a global scale
- **Bleacher Report**: 8 times more traffic
- **Discord**: scaled to 5 million concurrent users

Surse: Monterail, Discord

SONDAJE DIN 2019

Programming Languages: Best Paid

Indeed: Most Requested	Stack Overflow: Best Paid	Hired.com: Most Interview Requests
Java	Scala	Go
SQL	Clojure	Scala
JavaScript	Go	Ruby
Python	Erlang	TypeScript
HTML	Objective-C	Kotlin
CSS	WebAssembly	JavaScript
C#	Kotlin	Objective-C
C++	Rust	PHP

Ruby	F#	Java
PHP	Elixir	HTML

Programatorii care folosesc Elixir sunt plătiți bine 😅

Surse: New Relic, Stackoverflow

SĂ INSTALĂM ELIXIR

https://elixir-lang.org/install.html

SHELL-UL INTERACTIV ELIXIR

În directorul `bin`, trei executabile importante:

- `elixir` - *runtime*-ul
- `elixirc` - compilatorul
- `iex` - *shell*-ul interactiv

`iex` (Linux, macOS) / `iex.bat` (Windows)

Pentru ieșire, apasă `Ctrl+C` de două ori

În IEx putem defini:

- expresii
- funcții
- module

SCRIPTURI

Putem regăsi și rula un script pentru a-l folosi în `iex`:

`iex -S scriptname`

Putem rula scripturi cu executabilul `elixir`

- creează un fișier `hello.exs`
- conținutul fișierului va fi:

```
IO.puts("Hello world!")
```

EXPRESII ȘI TIPURI DE DATE

ÎN ELIXIR

Totul este o expresie

EXPRESII ARITMETICE

(integer și float)

`+, -, *, /, div, rem`

integer - nu are limită de mărime (limitat doar de cantitatea de memorie disponibilă)

float - valori pe 64 biți în dublă precizie

EXPRESII ARITMETICE

(integer și float)

Notă: putem omite parantezele rotunde atunci când apelăm funcții cu nume

Exemplu: `div 10,2` vs. `div(10,2)`

Funcții utile: `is_integer/1`, `is_float/1`, `is_number/1`

CUM IDENTIFICĂM FUNCȚIILE?

Prin nume și numărul de parametri pe care îl primesc

Exemplu: `div/2`

Notă: numărul de parametri se numește *aritate*

Notă: funcțiile din modulul `Kernel` - precum `div/2`, `rem/2`, `round/1` - nu trebuie prefixate cu numele modulului. El este importat automat.

CUM ACCESĂM DOCUMENTAȚIA UNEI FUNCȚII?

IEx are la dispoziție funcția `h`

Exemplu: `h div/2` (sau `h(div/2)`)

Funcția `h` funcționează și cu operatori (ex: `h +/2`) sau alte construcții

Funcția `h` apelată fără parametri afișează documentația pentru modulul `IEx.Helpers` - locul în care `h` este definită

TIPUL BOOLEAN

Valori posibile: `true` și `false`

Putem verifica dacă o valoare este de tip boolean: `is_boolean/1`

Operatori care lucrează *doar* cu valori de tip boolean: `or`, `and` și `not`

`or` și `and` sunt operatori de tip *short-circuit*

ALȚI OPERATORI CARE LUCREAZĂ CU "VALORI DE ADEVĂR"

`||`, `&&`, `!`

- acceptă argumente de orice tip
- se consideră că toate valorile au valoare de adevăr "adevărat", cu excepția valorilor `false` și `nil`
- regulă: folosim `or`, `and` și `not` atunci când toate valorile sunt de tip boolean; folosim `||`, `&&` și `!` atunci când oricare dintre valori este non-boolean

OPERATORI DE COMPARARE

`==`, `!=`, `===`, `!==`, `<=`, `>=`, `<` şi `>`

- rezultatul evaluării este un boolean
- diferența dintre `==` și `===` este că al doilea este mai strict la compararea valorilor de tip `integer` și `float`

ALTE COMPARĂRI

În Elixir putem compara valori de tipuri diferite; de exemplu, expresia `1000 < "hello"` este evaluată la valoarea `true`

Limbajul este pragmatic; această abordare ne ajută, de exemplu, la sortare

Ordinea tipurilor: number < atom < reference < function < port < pid < tuple < map < list < bitstring

Documentație:
https://hexdocs.pm/elixir/operators.html

ATOMI

Atom - dată a cărei valoare este numele ei

Exemple: `:person`, `:car`, `:truck`, `:motorcycle`

- cunoscut în unele limbaje ca *simbol*
- util pentru a enumera un număr de valori distincte
- cel mai apropiat tip în Java - enum
- doi atomi sunt egali dacă numele lor sunt egale

ATOMI

- valorile de tip boolean sunt atomi:
  - `true == :true`
  - `false == :false`
- `nil` este atomul `:nil`
- adeseori, atomii sunt folosiți ca valori returnate de funcții, pentru a exprima starea unei operații (ex: `:ok`, `:error`)
- funcție utilă: `is_atom/1`

LISTE

Înşiruire de valori de orice tip

Un tip de date foarte des folosit

Listele sunt desemnate prin paranteze drepte: []

În Elixir avem liste înlănţuite

Exemple:

- `[1, 2, 3]`
- `[10, true, "România", 5]`
- `[:alune, :migdale]`
- `[]`

LISTE

Operațiile efectuate asupra unei liste nu modifică niciodată lista inițială (datele sunt *imutabile* în Elixir)

Operatori pentru liste: `++/2`, `--/2`

Funcții utile: `length/1`, `hd/1` (head), `tl/1` (tail)

Funcție utilă pentru inspectarea tipului unei valori: `i/1`

LISTE

Caz special: `[65,66,67]`

Când detectează valori ASCII tipăribile, Elixir afișează structura de date sub forma unei liste de caractere (*charlist*)

Structurile *charlist* nu sunt același lucru cu *string*-urile, dar putem converti dintr-un tip în celălalt

De obicei, folosim *charlist* atunci când vrem să apelăm cod Erlang

LISTE

O listă e o structură de date recursivă

Construim o listă, reprezentând-o ca pe o pereche

`[head|tail]`

Operatorul | se numește *cons* (de la "construct")

`head` este o valoare de orice tip, iar `tail` este o listă

LISTE

Exemplu: `[3,1,0,2]`, unde 3 este *head*, iar `[1,0,2]` este *tail*

Exemplu: `[4,2,5]` este echivalent cu `[4|[2,5]]`

Exercițiu: cum construim recursiv lista `[1,2,3,4]`?

Exercițiu: cum putem afișa recursiv elementele unei liste?

STRING-URI

Înșiruiri de caractere, delimitate prin ghilimele; Elixir folosește *encoding* UTF-8

Nu există un tip dedicat string-urilor; sunt reprezentate prin intermediul tipului *binary*. Un *binary* este o secvență continuă de baiți

Exemple:

```
"Elixir" # acesta este un string
is_binary("hello") # returnează true
```

STRING-URI

Aflăm numărul de baiți dintr-un string cu `byte_size/1`, de exemplu: `byte_size("Brașov")`

Aflăm lungimea unui string cu `length/1`, de exemplu: `length("Brașov")`

Putem concatena string-uri cu operatorul <>, de exemplu: `"hello " <> "world"`

STRING-URI

Putem interpola (îngloba valori în) string-uri:

`"Numărul PI: #{3 + 0.14}"`

Avem posibilitatea să definim string-uri multilinie:

```
"Acesta
este un
text"
```

Într-un script sau program, afișăm pe ecran un string cu `IO.puts/1`

Afișarea pe ecran este un **efect**, dar funcția returnează atomul `:ok`

BINARY

Un *binary* e o înșiruire de baiți, delimitată prin << și >>

Exemplu: `<<1,2,3,4>>`

Concatenarea a două structuri de tip *binary* se face cu operatorul... <>

Ce se întâmplă dacă introducem în IEx următoarea expresie? `<<97,98,99>>`

Dar pentru `<<97,98,99>> <> <<100>>`?

TUPLURI

Enumerare de valori, finită și ordonată; exemple: pereche, triplet

Reprezentată printr-o înșiruire eterogenă de valori, delimitate prin acolade; exemple: `{:ok, 3.14}`, `{:error, "Database error"}`, `{"primii", "pași", "cu", "Elixir"}`

Spre deosebire de liste, accesăm elementele unui tuplu prin index (stochează elementele într-o zonă contiguă de memorie) cu `elem/2`

PATTERN MATCHING

OPERATORUL "MATCH"

Operatorul = se numește "match"

Nu este operator de asignare; în Elixir nu avem asignare, ci "binding"

Variabilele sunt doar etichete pentru valori imutabile

OPERATORUL =

Exemplu: `x=10`
Variabilei care nu etichetează o valoare i se "bind"-uiește valoarea din dreapta
Dacă aplicăm din nou = (de exemplu, `x=15`) spunem că facem "re-binding"; valoarea anterioară nu se distruge

OPERATORUL =

`15=x` realizează un "match"; expresia returnează valoarea din dreapta

`20=x` ar "ridica" o eroare: `** (MatchError) no match of right hand side value: 15`

DESTRUCTURARE CU =

Folosim = și pentru a descompune structuri de date:

Tupluri:

- `{x,y}={-7,3}`
- `{a,a}={3,8} # wait, what?!`
- `{:ok,result}={:ok,100}`

Liste:

- `[a,b,c]=[5,3,9]`
- `[h|t]=["first","Elixir","session"]`

+ alte tipuri, cum ar fi *binary* sau *map*

OPERATORUL "PIN"

Operatorul ^ se numește "pin"

Se pune în fața numelui unei variabile și ne ajută să facem *pattern matching* cu valoarea "bind"-uită

```
# binding
x = 10
# re-binding
x = 20
# match cu valoarea existentă asociată lui x
^x = 30
```

Putem folosi ^ și cu structuri de date

WILDCARD

Simbolul _ este un *wildcard*

Util când facem *pattern matching* dar nu ne interesează o valoare

Exemplu: `{a,b,_}={1,2,3}`

Poate fi pus și în fața unui nume de variabilă

Exemplu: `{a,b,_c}={1,2,3}` (variabila "c" nu este"bind"-uită)

STRUCTURI DE CONTROL: CASE, COND ȘI IF

CASE

`case` ne permite să comparăm o valoare, structură de date etc. cu mai multe *pattern*-uri

Compararea se oprește la primul *match*

```
x = {:ok, "Welcome to Elixir!"}
case x do
  {:error, reason} ->
    "This clause doesn't match right now"
  {:ok, res} ->
    "This clause matches and binds the string to res: #{res}"
  _ ->
    "This clause matches any value"
end
```

GUARDS

Un *guard* este o expresie booleană introdusă de cuvântul *when*

Folosim *guards* pentru a îmbunătăți *pattern matching*-ul cu condiții în plus

Un *guard* acceptă puține construcții; el trebuie să termine execuția întotdeauna (și rapid)

Documentație: https://hexdocs.pm/elixir/guards.html

CASE CU GUARDS

```
case {2, 4, 6} do
  {a, _b, _c} when rem(a, 2) == 0 ->
    a
  {a, b, c} ->
    a + b + c
  _ ->
    0
end
```

CLAUZĂ CASE CU ^

Atunci când nu vrem să facem *re-binding* unei variabile, ci *match* pe valoarea ei

```
x = :apple
...
case :orange do
  ^x       -> "Doesn't match right now"
  :papaya  -> "Neither does this one"
  _        -> "Some other fruit"
end
```

ÎNCĂ CEVA DESPRE GUARDS

Expresiile *guard* nu ridică erori, ci pur și simplu nu fac *match*

```
case 10 do
  x when length(x) > 0 -> "Looks good"
  x -> "I received #{x}"
end
```

COND

Util când vrem să verificăm diferite condiții

Se oprește când găsește prima condiție care nu se evaluează la `false` sau `nil`

```
v = 121
...
cond do
  1 + 1 == 3 -> "Go back to school"
  rem(v, 3) -> "Good arithmetic exercise"
  true -> "So you ended up here..."
end
```

COND

Echivalent cu `else if` din limbajele imperative; folosit mai rar

În Elixir totul este o expresie (ce trebuie să se evalueze la o valoare); se ridică `CondClauseError` dacă nicio condiție nu se verifică

E o idee bună să adăugăm o clauză *true* la sfârșit

Orice valoare în afară de `false` și `nil` este considerată adevarată

IF ȘI UNLESS

`if` și `unless` ne ajută atunci când trebuie să verificăm o singură condiție

```
if x = hd([1, 2]) do
  IO.puts(x + 10)
end
```

`if` funcționează similar cu instrucțiunea clasică din limbajele imperative; dacă nu este verificată condiția, evaluează la `nil`

IF ȘI UNLESS

`unless` este opusul lui `if`

Ambele construcții acceptă clauze `else`

```
if elem({30, -3, 33}, 1) > 0 do
  "This is not true"
else
  "But this is"
end
```

BLOCURILE DO/END

Delimitează multe structuri în Elixir, precum `case`, `cond`, `if`

(le putem scrie și sub o altă formă, despre care învățăm după ce discutăm despre *keywords*)

STRUCTURI DE DATE ASOCIATIVE (CHEIE, VALOARE)

LISTE DE CUVINTE CHEIE (*KEYWORD LISTS*)

Liste cu elemente de tip tuplu (cu două elemente), primul element fiind un atom

```
[{:ferrari, 324_000}, {:"alfa romeo", 87_000}]
```

echivalent cu

```
[ferrari: 324_000, "alfa romeo": 87_000]
```
(sintaxă specială)

KEYWORD LISTS

Tot ce am spus până acum despre liste e valabil și pentru *keyword lists*

Sunt folosite frecvent în construcțiile limbajului, în DSL-uri și pentru mecanismul de parametri "opționali" pentru funcții

KEYWORD LISTS

Macro-ul `if/2`, suportă și sintaxa:

```
if div(x, 10), do: x + 1, else: x - 1
# echivalent cu
if div(x, 10) do
  x + 1
else
  x - 1
end
# echivalent cu
if (div(x, 10), [do: x + 1, else: x - 1])
# echivalent cu
if (div(x, 10), [{:do, x + 1}, {:else, x - 1}])
```

KEYWORD LISTS

Blocul `do`/`end` este *syntactic sugar* construit peste *keyword lists*

Această sintaxă e întâlnită și în alte locuri, cum ar fi definiții de funcții

În general, dacă ultimul parametru al unei funcții este un *keyword list*, putem omite parantezele drepte

MAP-URI

Structură de tip cheie/valoare foarte des folosită

Un *map* se creează cu sintaxa `%{}`

```
%{"Maria" => 20, "Gigel" => 14, "Ionel" => 25}
```

Cheile pot fi de orice tip și nu sunt ordonate (spre deosebire de *keyword lists*)

MAP-URI

Foarte utile în *pattern matching*

```
%{:a => a} = %{:a => 1, 2 => :b}
```

Map-ul gol "match"-uiește toate *map*-urile

```
%{} = %{:a => 1, 2 => :b}
```

Modulul Map ne oferă funcții utile de lucru cu *map*-uri

MAP-URI

Când toate cheile sunt atomi, putem folosi sintaxa de *keyword list*: `my_map = %{name: "George", age: 25}`

O altă sintaxă de accesare a cheilor de tip atom:

`my_map.name`

Programatorii Elixir preferă sintaxa `map.field` și *pattern matching*-ul, în detrimentul funcțiilor din modulul *Map* => cod asertiv

STRUCTURI DE DATE IMBRICATE

```
videos = [
  mycat: %{title: "Ninja Cat", duration: 208, rating: 4.7,
    tags: ["cat", "home", "pets"]},
  diy: %{title: "How to build a house", duration: 1200,
    rating: 3.8, tags: ["house", "building"]}
]
```

MODULE ȘI FUNCȚII

MODULE

În Elixir, organizăm codul în module

```
defmodule MyModule do
...
end
```

Numele unui modul începe cu majusculă; respectă convenția *CamelCase*

Numele modulelor și ale funcțiilor sunt atomi

FUNCȚII

```elixir
defmodule Temperatures do
  def celsius_to_kelvin(c) do
    c + 273.15
  end
end
```

Definim funcțiile în interiorul modulelor

Valoarea ultimei expresii din funcție este returnată; nu există cuvântul `return`

Numele funcțiilor respectă convenția *snake_case*

FUNCȚII

Tipuri:

- funcții cu nume
- funcții anonime (expresii lambda)

FUNCȚII CU NUME

Introduse prin:

- `def` - funcție publică (exportată, poate fi apelată și în alte module)
- `defp` - funcție privată (disponibilă local)

FUNCȚII PRIVATE

```elixir
defmodule Messages do
  def sum_as_message(a, b) do
    msg(a + b)
  end

  def multiplication_as_message(a, b) do
    msg(a * b)
  end

  defp msg(x) do
    "Result: #{x}"
  end
end
```

Funcția `msg/1` nu poate fi apelată din afara modulului `Messages`

FUNCȚII CU *GUARDS*

```
defmodule Dates do
  def leap_year?(y) when is_integer(y) do
    cond do
      rem(y, 4) != 0 -> false
      rem(y, 100) != 0 -> true
      rem(y, 400) != 0 -> false
      true -> true
    end
  end
end
```

Funcția se execută doar dacă se verifică *guard*-ul

Convenție: dacă numele funcției se termină cu ?
returnează un boolean

CLAUZE DE FUNCȚII (*FUNCTION CLAUSES*)

Atunci când definim o funcție, folosim *pattern matching* pentru a separa diverse ramuri logice ale acesteia

O astfel de ramură se numește *function clause*

O funcție - identificată prin nume și aritate - poate avea una sau mai multe clauze

Se execută prima clauză întâlnită care face *match*

FUNCTION CLAUSES

```elixir
defmodule Lists do
  def head([]) do
    {:error, "empty list"}
  end

  def head([h | _t]) do
    {:ok, h}
  end
end
```

Exercițiu (cu clauze 😁): scrie o funcție care primește un *map*. Dacă există cheia "city", returnează un mesaj conținând valoarea asociată; altfel, caută cheia "country"; altfel, afișează un mesaj relevant

NOTAȚIA `, do:`

Așa cum am văzut în cazul construcției `if`, putem folosi pentru funcții notația ce provine de la *keyword lists*

```
1 defmodule Greetings do
2   def hello(n) when is_binary(n), do: IO.puts("Hello #{n}")
3   def hello(_), do: IO.puts("I don't understand your name")
4 end
```

Regulă: folosim `, do:` pentru funcții simple, cu implementarea pe o linie; în rest, folosim blocul

CAPTURAREA FUNCȚIILOR CU OPERATORUL & (*CAPTURE*)

Uneori avem nevoie să *bind*-uim o funcție cu nume unei variabile sau să o transmitem ca parametru altei funcții

Ne putem folosi de numele și aritatea acesteia, împreună cu operatorul *capture*:

```
h = &Lists.head/1
is_function(h) # returnează true
```

CAPTURE &

Acum putem transmite variabila ca parametru altei funcții sau chiar să apelăm funcția asociată

O putem apela astfel: `h.([1,4,0])`

DEFINIREA DE FUNCȚII CU &

Capture ne oferă un mod rapid de a defini funcții:

```
plus = &(&1 + &2)
```

Apoi putem apela funcția cu ajutorul variabilei și a notației cu punct:

```
plus.(4, 1)
```

Notație foarte utilă în cazul funcțiilor care primesc alte funcții ca parametri (*higher-order functions*)

ARGUMENTE IMPLICITE (*DEFAULT*)

Specificăm argumente *default* astfel

```
def hello(name \\ "User"), do: "Hello, #{name}"
...
hello "Gigel"
...
hello
```

Argumentul implicit nu e evaluat la compilare, ci doar atunci când e nevoie de el

FUNCȚII ANONIME

Pe lângă funcțiile cu nume, avem și funcții anonime sau expresii lambda

Asemenea funcțiilor construite cu *capture*, expresiile lambda pot fi *bind*-uite de variabile sau transmise ca parametri

Funcțiile sunt date

```
mult = fn x, y -> x * y end
```

FUNCȚII ANONIME

```
pow = fn x -> x * x end
is_function(pow) # returnează true
pow.(7) # apelăm funcția cu ajutorul notației cu punct
# punctul este necesar pentru a face diferența între apelul
# unei funcții cu numele pow și apelul unei funcții anonime
# bind-uită de variabila pow
```

La fel ca în cazul funcțiilor construite cu operatorul *capture*, expresiile lambda sunt foarte folositoare când lucrăm cu *HOFs*

```
fn x -> x * x end
# echivalent cu
&(&1 * &1)
```

FUNCȚII ANONIME

Funcțiile anonime sunt *closures*

Pot accesa variabilele care sunt în *scope* în momentul definirii funcției

```
mult = fn a, b -> a * b end
sqr = fn a -> mult(a, a) end
```

Expresiile lambda nu influențează mediul exterior

```
year = 2020
other_year = fn -> year = 2014 end
IO.puts(other_year.())
IO.puts(year)
```

FUNCȚII ANONIME

Expresiile lambda pot avea mai multe clauze

De asemenea, pot avea *guards*

```
division = fn
        x, y when y != 0 -> {:ok, x / y}
        x, y -> {:error, "Can't divide #{x} and #{y}"}
    end
```

Numărul argumentelor trebuie să fie același în fiecare clauză a funcției

RECURSIVITATE

O tehnică de programare des întâlnită în programarea funcțională

Elixir nu are instrucțiuni de ciclare (*for*, *while*, *do/while*) - datorită imutabilității datelor

Motivul: nu putem modifica date *in place* cum facem în limbajele imperative

RECURSIVITATEA PENTRU CICLARE

În Elixir, ciclăm pe o listă cu ajutorul recursivității

Cum dublăm (în mod tradițional) elementele unei liste?

```java
// în Java
List<Integer> list = Arrays.asList(1, 2, 3);
for (int i = 0; i < list.size(); i++) {
    list.set(i, list.get(i) * 2);
}
```

```elixir
# în Elixir
def double_elem([]), do: []
def double_elem([h | t]), do: [h * 2 | double_elem(t)]
```

TAIL CALL OPTIMIZATION

Pentru liste foarte mari, e posibil ca implementarea anterioară să umple *stack*-ul cu *frame*-urile apelurilor funcției recursive

Putem elimina adăugarea pe *stack* a noului apel prin mecanismul *tail call optimization*

Această optimizare e un *feature* al compilatorului

TAIL CALL OPTIMIZATION

Tail call optimization: dacă ultimul lucru pe care-l face o funcție e să se apeleze recursiv, nu se adaugă un nou *frame* pe *stack* - o astfel de funcție se numește *tail recursive*

Procesul e foarte eficient - sare la începutul funcției cu noile valori ale parametrilor - și imită un *loop* dintr-un limbaj imperativ

Exercițiu: să implementăm exemplul anterior în manieră *tail recursive* (hint: folosim un acumulator)

RECURSIVITATE CU *GUARDS*

Exercițiu: scrie o funcție recursivă care afișează numerele de la 0 la n, unul sub celălalt. Definește clauzele funcției folosind *guards*

PATTERN-URI ÎN PROGRAMAREA FUNCȚIONALĂ

- map
- filter
- reduce

Operații pe care le aplicăm asupra colecțiilor de date

Pattern-uri preluate și de alte limbaje, nu neapărat funcționale (de exemplu, Java 8 Streams)

Framework-urile *big data* folosesc aceste *pattern*-uri (de exemplu, Hadoop)

MAPPING

O operație de transformare a elementelor; colecția rezultată va avea același număr de elemente

Exemplul din secțiunea despre recursivitate (`double_elem/1`) este o mapare

Exercițiu: Să implementăm o funcție generală `mapping/2`, care primește o listă de numere și aplică fiecărui element o transformare (prin intermediul unei funcții)

FILTERING

O operație de filtrare a elementelor; reținem doar elementele ce respectă niște condiții (un *predicat*)

Exercițiu: Să implementăm o funcție generală `filtering/2`, care primește o listă de numere și reține doar elementele ce verifică un predicat

REDUCING

O operație de combinare a elementelor, pentru a obține o valoare

Exercițiu: Să implementăm o funcție generală `reducing/3`, care primește o listă de numere și reduce elementele la o valoare

MAP, FILTER, REDUCE

`mapping/2`, `filtering/2`, `reducing/3` sunt *higher-order functions*

Nici una dintre funcții nu modifică lista inițială (imutabilitate), ci construiesc date noi

ENUM ȘI STREAM

Pattern-urile *map*, *filter* și *reduce* sunt deja implementate în biblioteca standard de funcții

Avem la dispoziție modulele `Enum` și `Stream`

Enum implementează operații *eager*, iar Stream implementează operații *lazy*

Documentație: https://hexdocs.pm/elixir/Enum.html, https://hexdocs.pm/elixir/Stream.html

OPERATORUL *PIPE* |>

Preia rezultatul expresiei din stânga lui și îl transmite ca prim parametru funcției din dreapta lui

```
[1, :a, 2, 3, :b, :c]
|> Enum.filter(&is_number/1)
|> Enum.map(&(&1 * 2))
```

Folosit foarte des pentru înlănțuirea mai multor apeluri de funcții care transformă date, evidențiind datele care se transformă

CU ȘI FĂRĂ |>

Exercițiu: să găsim numerele dintr-o listă oarecare și să le înmulțim

```
Enum.reduce(
    Enum.filter(
        [10, :a, 2, 3, "numbers", "100", {3, 4}, 2],
        &is_number/1
    ),
    1, fn x, acc -> x * acc end
) # fără operatorul "pipe"
```

```
[10, :a, 2, 3, "numbers", "100", {3, 4}, 2]
|> Enum.filter(&is_number/1)
|> Enum.reduce(1, fn x, acc -> x * acc end)
```

TIPUL *STRUCT*

Tipul *map* este foarte util, însă uneori avem nevoie de verificări la compilare

Struct - structură de date construită peste *map*-uri

- face verificări la compilare
- oferă valori implicite

```elixir
defmodule Car do
defstruct brand: "Dacia", model: "Sandero", price: 11_000
end
```

Observație: lista de câmpuri este un *keyword list*

STRUCT

Garanții la compilare: doar câmpurile (toate câmpurile) declarate în `defstruct` sunt permise într-un struct

Câmpurile acceptă valori *default* (dar nu e obligatoriu); câmpurile fără valoare implicită primes `nil` și trebuie definite primele

```elixir
defmodule User do
  defstruct [:name, :age, country: "România"]
end
```

CREAREA, ACCESARE ȘI ACTUALIZAREA UNUI STRUCT

```
1 # creare
2 ferrari = %Car{brand: "Ferrari", model: "F-50", price: 780_0
3 # accesare
4 ferrari.brand
5 # actualizare
6 enzo = %{ferrari | model: "Enzo"}
7 # pattern matching
8 %Car{price: p} = enzo
9 p
```

TESTARE UNITARĂ CU *EXUNIT*

ExUnit este *framework*-ul standard de testare unitară cu care vine echipat Elixir

Testele sunt rulate cu comanda `mix test`

```elixir
defmodule DatesTest do
  use ExUnit.Case

  test "2020 is a leap year" do
    assert Dates.leap_year?(2020) == true
  end

  @tag :pending
  test "2019 is not a leap year" do
    assert Dates.leap_year?(2019) == false
  end
end
```

MIX

Elixir are *tooling* integrat; *Mix* este *build tool*-ul care ne susține pe toată durata de viață a unui proiect

- crearea unui proiect Mix: `mix new my_project --module MyModule`
- compilarea proiectului: `mix compile`
- sesiune IEx în interiorul proiectului: `iex -S mix`
  după modificări, recompilăm cu `recompile()`
- alte comenzi utile: `mix format`, `mix help`

FRAMEWORK-UL PHOENIX

```
connection
|> endpoint() # funcții ce se cheamă pentru fiecare request
|> router()
|> pipeline() # conexiunea trece printr-un pipeline numit,
               # ce are funcții comune pentru fiecare tip
               # principal de request
|> controller() # invocă modelul și randează un template,
                 # printr-un view
```

Crearea unui proiect nou

`mix phx.new air`

DENUMIRI, STIL ȘI FORMATARE

Convenții:

- Denumiri de module: *CamelCase*
- Denumiri de funcții și variabile: *snake_case*
- Indentare: două spații
- Putem formata automat codul dintr-un proiect cu: `mix format`

RESURSE

- Elixir: The Documentary
- Erlang: The Movie
- Getting Started
- Elixir Forum
- Elixir Digest
- /r/elixir

CĂRȚI RECOMANDATE

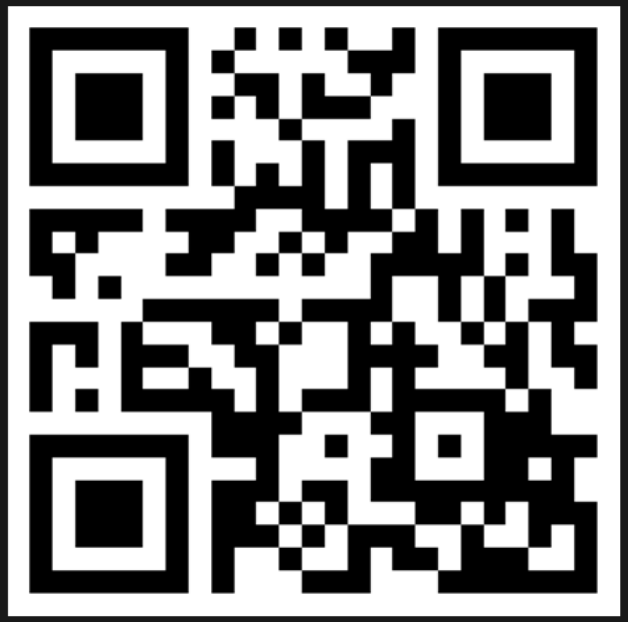
- Saša Jurić - Elixir in Action, Second Edition (Manning, 2019)
- Chris McCord, Bruce Tate and José Valim - Programming Phoenix 1.4 (The Pragmatic Bookshelf, 2019)

Acesta a fost... primul pas spre cariera de alchimiști.

E timpul să preparăm niște Elixir! 😉

MULȚUMESC!

Aşteptăm feedback-ul tău



http://bit.ly/agilehub-feedback

http://bit.ly/agilehub-feedback