DP23a - Funkcionális programozás, 2. gyakorlat

Konvenciók, jelölések

A gyakorlatok anyaga szakaszokra van felosztva, minden szakaszban a bevezetés után néhány feladatot definiálunk, néha megoldott feladatokat is bemutatunk.

Halványzöld peremű, fekete hátterű cellában (a továbbiakban: specifikációs cella) van a szükséges „keretezéssel", azaz a modul- és függvénydefinícióval együtt a megírandó függvény típusspecifikációja, valamint néhány teszthívás is. Ebbe a cellába nem lehet beleírni (csak szerkesztő módban), de a tartalmát ki lehet jelölni, lehet másolni.

Rózsaszín hátterű cellába írjuk az esetleges korlátozásokat: ne használja ezt, ne csinálja azt stb.

Halványzöld hátterű cellában jelennek meg a magyarázataink, illetve a javaslataink egyes feladatok megoldására. Az utóbbiak gyakran el vannak rejve: a Súgó feliratra kattintva jelennek meg.

Az egymást kölcsönösen kizáró minták használata...

Súgó

Ezt és ezt javasoljuk a függvény megírásához.

A feladatot megoldó függvényt, kifejezést egy Elixir-cellába írja be: a felugó menüben a + Elixir feliratra kattintva hozzon létre egy új cellát, másolja be a specifikációs cella tartalmát, majd írja meg és értékelje ki a specifikált függvényt vagy a kért kifejezést.

Feladatok listák rekurzív feldolgozására

Lista kettévágása

Két változatban is írja meg a split/2 függvényt: az első változatban ne használjon segédfüggvényt, és így akkumulátort sem, a másodikban pedig használjon segédfüggvényt akkumulátorral.

Ne használja az Enum.split és az Enum.reverse függvényeket a split/2 függvény megvalósítására! (De bármilyen könyvtári függvényt használhat az eredmény ellenőrzésére.)

```
defmodule Split do
 @spec split(xs :: [any()], n :: integer()) :: {ps :: [any()], ss :: [any()]}
 # Az xs lista n hosszú prefixuma (első n eleme) ps, length(xs)-n
 # hosszú szuffixuma (első n eleme utáni része) pedig ss
 def split(xs, n) do
 ...
 end
end
IO.puts(Split.split([10, 20, 30, 40, 50], 3) === {[10, 20, 30], [40, 50]})
```

```
IO.puts(IO.inspect(Split.split(~c"egyedem-begyedem", 8)) === Enum.split(~c"egyedem-begyedem", 8))
IO.puts(IO.inspect(Split.split(~c"papás-mamás", 6)) === Enum.split(~c"papás-mamás", 6))
IO.puts(Split.split(~c"nem_vágom", 0) === Enum.split(~c"nem_vágom", 0))
IO.puts(Split.split(~c"", 10) === Enum.split(~c"", 10))
IO.puts(Split.split(~c"", 0) === Enum.split(~c"", 0))

defmodule Split1 do
 @spec split(xs :: [any()], n :: integer()) :: {ps :: [any()], ss :: [any()]}
 # Az xs lista n hosszú prefixuma (első n eleme) ps, length(xs)-n
 # hosszú szuffixuma (első n eleme utáni része) pedig ss
 def split([], _n), do: {[], []}
 def split(xs, 0), do: {[], xs}

 def split([x | xs], n) do
  {ps, ss} = split(xs, n - 1)
  {[x | ps], ss}
 end
end

IO.puts(Split1.split([10, 20, 30, 40, 50], 3) === {[10, 20, 30], [40, 50]})
IO.puts(IO.inspect(Split1.split(~c"egyedem-begyedem", 8)) === Enum.split(~c"egyedem-begyedem", 8))
IO.puts(IO.inspect(Split1.split(~c"papás-mamás", 6)) === Enum.split(~c"papás-mamás", 6))
IO.puts(Split1.split(~c"nem_vágom", 0) === Enum.split(~c"nem_vágom", 0))
IO.puts(Split1.split(~c"", 10) === Enum.split(~c"", 10))
IO.puts(Split1.split(~c"", 0) === Enum.split(~c"", 0))
```

A második változathoz írjon segédfüggvényt egy lista megfordítására is, hiszen nem használhatja az Enum.reverse függvényeket.

```
defmodule Split2 do
 @spec split(xs :: [any()], n :: integer()) :: {ps :: [any()], ss :: [any()]}
 # Az xs lista n hosszú prefixuma (első n eleme) ps, length(xs)-n
 # hosszú szuffixuma (első n eleme utáni része) pedig ss
 def split(xs, n), do: split(xs, n, [])

 @spec split(xs :: [any()], n :: integer(), zs :: integer()) :: {ps :: [any()], ss :: [any()]}
 # Az xs lista n hosszú prefixuma ps, első n eleme utáni szuffixuma ss
 # zs a prefixumot gyűjtő akkumulátor
 def split([], _n, zs), do: {rev(zs), []}
 def split(xs, 0, zs), do: {rev(zs), xs}
 def split([x | xs], n, zs), do: split(xs, n - 1, [x | zs])

 @spec rev(xs :: [any()]) :: rs :: [any()]
 # xs megfordítottja rs
 def rev(xs), do: rev(xs, [])
 @spec rev(xs :: [any()], zs :: [any()]) :: rs :: [any()]
 def rev([x | xs], zs), do: rev(xs, [x | zs])
```

```
 def rev([], zs), do: zs
end

IO.puts(Split2.split([10, 20, 30, 40, 50], 3) === {[10, 20, 30], [40, 50]})
IO.puts(IO.inspect(Split2.split(~c"egyedem-begyedem", 8)) === Enum.split(~c"egyedem-begyedem", 8))
IO.puts(IO.inspect(Split2.split(~c"papás-mamás", 6)) === Enum.split(~c"papás-mamás", 6))
IO.puts(Split2.split(~c"nem_vágom", 0) === Enum.split(~c"nem_vágom", 0))
IO.puts(Split2.split(~c"", 10) === Enum.split(~c"", 10))
IO.puts(Split2.split(~c"", 0) === Enum.split(~c"", 0))
```

Lista adott feltételt kielégítő elemeiből álló prefixuma

Két változatban is írja meg a takewhile/2 függvényt: az első változatban ne használjon segédfüggvényt, és így akkumulátort sem, a másodikban pedig használjon segédfüggvényt akkumulátorral.

Ne használja az Enum.take_while/2 függvényt a takewhile/2 függvény megvalósítására! (De bármilyen könyvtári függvényt használhat az eredmény ellenőrzésére.)

```
defmodule Take do
 @spec takewhile(xs :: [any()], f :: (any() -> boolean())) :: rs :: [any()]
 def takewhile(xs, f) do
 ...
 end
end
IO.puts(Take.takewhile(~c"álom12" ++ [:a] ++ [~c"34brigád"], &is_integer/1) === ~c"álom12")
IO.puts(Take.takewhile(~c"abcdefghijkl", fn x -> x < ?f end) === ~c"abcde")

defmodule Take1 do
 @spec takewhile(xs :: [any()], f :: (any() -> boolean())) :: rs :: [any()]
 def takewhile([], _f), do: []

 def takewhile([x | xs], f) do
  cond do
   f.(x) -> [x | takewhile(xs, f)]
   true -> []
  end
 end
end

IO.puts(Take1.takewhile(~c"álom12" ++ [:a] ++ [~c"34brigád"], &is_integer/1) === ~c"álom12")
IO.puts(Take1.takewhile(~c"abcdefghijkl", fn x -> x < ?f end) === ~c"abcde")

defmodule Take2 do
 @spec takewhile(xs :: [any()], f :: (any() -> boolean())) :: rs :: [any()]
 def takewhile(xs, f), do: Split2.rev(takewhile(xs, f, []))
```

```
 @spec takewhile(xs :: [any()], f :: (any() -> boolean()), zs :: [any()]) :: rs :: [any()]
 def takewhile([], _f, zs), do: zs

 def takewhile([x | xs], f, zs) do
  cond do
   f.(x) -> takewhile(xs, f, [x | zs])
   true -> zs
  end
 end
end

IO.puts(Take2.takewhile(~c"álom12" ++ [:a] ++ [~c"34brigád"], &is_integer/1) === ~c"álom12")
IO.puts(Take2.takewhile(~c"abcdefghijkl", fn x -> x < ?f end) === ~c"abcde")
```

Lista minden n-edik elemének kihagyásával létrejövő lista

Írjon függvényt egy olyan lista létrehozására, amelyikből a paraméterként átadott lista minden n-edik eleme, az elsőtől kezdve, ki van hagyva. Két változatban is írja meg a dropevery/2 függvényt: az első változatban ne használjon segédfüggvényt, és így akkumulátort sem, a másodikban pedig használjon jobbrekurzív segédfüggvényt akkumulátorral.

Ne használja az Enum.drop_every/2 függvényt a dropevery/2 függvény megvalósítására! (De bármilyen könyvtári függvényt használhat az eredmény ellenőrzésére.)

```
defmodule Drop do
 @spec dropevery(xs :: [any()], n :: integer()) :: rs :: [any()]
 def dropevery(xs, n) do
  ...
 end
end
ls = ~c"álom" ++ [:a] ++ ~c"egybrigád"
IO.inspect(Drop.dropevery(ls, 4) === ~c"lomegyrigd")
ls = ~c"abcdefghijkl"
IO.inspect(Drop.dropevery(ls, 5) === ~c"bcdeghijl")
ls = ~c"1234567"
IO.inspect(Drop.dropevery(ls, 2) === ~c"246")
ls = []
IO.inspect(Drop.dropevery(ls, 3) === [])
ls = [:a, :b, :c, :d, :e, :f, :g, :h, :i, :j, :k, :l, :m]
IO.inspect(Drop.dropevery(ls, 3) === [:b, :c, :e, :f, :h, :i, :k, :l])
```

Súgó

A segédfüggvényt nem használó változatban, ha nincs más ötlete, használja a for jelölést, generátoraként a ../2, szűrőjeként a rem/2 műveletet, a listaelemek elérésére pedig a nemrég megírt AtEx.at/2 függvényt.

```
defmodule Drop1 do
 @spec dropevery(xs :: [any()], n :: integer()) :: rs :: [any()]
 def dropevery([], _n), do: []

 def dropevery(xs, n) do
  for i <- 1..length(xs), rem(i, n) != 1, do: AtEx.at(xs, i - 1)
 end
end

ls = ~c"álom" ++ [:a] ++ ~c"egybrigád"
IO.inspect(Drop1.dropevery(ls, 4) === ~c"lomegyrigd")
ls = ~c"abcdefghijkl"
IO.inspect(Drop1.dropevery(ls, 5) === ~c"bcdeghijl")
ls = ~c"1234567"
IO.inspect(Drop1.dropevery(ls, 2) === ~c"246")
ls = []
IO.inspect(Drop1.dropevery(ls, 3) === [])
ls = [:a, :b, :c, :d, :e, :f, :g, :h, :i, :j, :k, :l, :m]
IO.inspect(Drop1.dropevery(ls, 3) === [:b, :c, :e, :f, :h, :i, :k, :l])
```

A segédfüggvényt használó változatban mellőzze a for jelölést, a ../2 műveletet, valamint a listaelemeket indexelő függvényeket.

```
defmodule Drop2 do
 @spec dropevery(xs :: [any()], n :: integer()) :: rs :: [any()]
 def dropevery(xs, n), do: Split2.rev(dropevery(xs, n, 1, []))

 def dropevery([], _n, _i, zs), do: zs
 def dropevery([x | xs], n, i, zs) when rem(i, n) != 1, do: dropevery(xs, n, i + 1, [x | zs])
 def dropevery([_ | xs], n, i, zs), do: dropevery(xs, n, i + 1, zs)
end

ls = ~c"álom12" ++ [:a] ++ [~c"34brigád"]
IO.puts(Drop2.dropevery(ls, 5) == Enum.drop_every(ls, 5))
ls = ~c"abcdefghijkl"
IO.inspect(Drop2.dropevery(ls, 4))
ls = ~c"1234567"
IO.inspect(Drop2.dropevery(ls, 2))
ls = []
IO.inspect(Drop2.dropevery(ls, 3))
ls = [:a, :b, :c, :d, :e, :f, :g, :h, :i, :j, :k, :l, :m]
IO.inspect(Drop2.dropevery(ls, 3))
```

Lista egyre rövidülő szuffixumainak listája

```
defmodule Tails do
 @spec tails(xs :: [any]) :: zss :: [[any]]
```

```
 # Az xs lista egyre rövidülő szuffixumainak listája zss
 def tails(xs) do
 ...
 end
end
IO.puts(Tails.tails([1, 4, 2]) === [[1, 4, 2], [4, 2], [2], []])
IO.puts(Tails.tails([:a, :b, :c, :d]) === [[:a, :b, :c, :d], [:b, :c, :d], [:c, :d], [:d], []])
IO.puts(Tails.tails([:z]) === [[:z], []])
IO.puts(Tails.tails([]) === [[]])
```

Súgó

A tails függvénynek listák listája az eredménye, így ha üres listára alkalmazzuk, akkor olyan lista lesz a visszatérési értéke, melynek egyetlen eleme van, az üres lista.

```
defmodule Tails do
 @spec tails(xs :: [any]) :: zss :: [[any]]
 # Az xs lista egyre rövidülő szuffixumainak listája zss
 def tails([]), do: [[]]
 def tails([_x | xs] = xxs), do: [xxs | tails(xs)]
end

IO.puts(Tails.tails([1, 4, 2]) === [[1, 4, 2], [4, 2], [2], []])
IO.puts(Tails.tails([:a, :b, :c, :d]) === [[:a, :b, :c, :d], [:b, :c, :d], [:c, :d], [:d], []])
IO.puts(Tails.tails([:z]) === [[:z], []])
IO.puts(Tails.tails([]) === [[]])
```

Lista egymást követő két-két eleméből képzett párok listája

Írjon olyan rekurzív függvényt, amelyik egy lista 1. és 2., 3. és 4., 5. és 6. s.í.t. elemeiből képzett párok listáját adja eredményül. Ha a listának kettőnél kevesebb eleme van, az eredmény az üres lista legyen. Ha a listának páratlan számú eleme van, az utolsót dobja el.

```
defmodule Pairs do
 @spec pairs(xs::[any()]) :: zs :: [any()]
 def pairs(xs), do: ...
end
zs = [{1,2}, {3,4}, {5,6}, {7,8}, {9,10}, {11,12}, {13,14}, {15,16}, {17,18}, {19,20}]
(1..20 |> Range.to_list() |> Pairs.pairs() == zs) |> IO.puts
zs = [{1,2}, {3,4}, {5,6}, {7,8}, {9,10}]
(1..11 |> Range.to_list() |> Pairs.pairs() == zs) |> IO.puts
([1] |> Pairs.pairs() == []) |> IO.puts
([1] |> Pairs.pairs() == []) |> IO.puts
```

```
defmodule Pairs do
 @spec pairs(xs :: [any()]) :: zs :: [any()]
```

```
  def pairs([x1, x2 | xs]), do: [{x1, x2} | pairs(xs)]
  def pairs(_), do: []
end

zs = [{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}, {11, 12}, {13, 14}, {15, 16}, {17, 18}, {19, 20}]
(1..20 |> Range.to_list() |> Pairs.pairs() == zs) |> IO.puts()
zs = [{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}]
(1..11 |> Range.to_list() |> Pairs.pairs() == zs) |> IO.puts()
([1] |> Pairs.pairs() == []) |> IO.puts()
([1] |> Pairs.pairs() == []) |> IO.puts()
```

```
true
true
true
true
```

```
:ok
```

Listában párosával előforduló elemek listája

Írjon olyan rekurzív függvényt, amelyik egy lista elemei közül az összes olyat visszaadja az eredménylistában, amelyet vele azonos értékű elem követ, azaz például két egymást követő, azonos értékű elemből egyet, három egymást követőből kettőt stb. Írjon segédfüggvényt és akkumulátort nem használó, valamint akkumulátoros segédfüggvényt használó változatokat is. Próbáljon meg egyéb változatokat is írni, pl. for jelöléssel, az Enum.zip/1 függvény alkalmazásával.

```
defmodule Parosan do
  @spec parosan(xs :: [any()]) :: rs :: [any()]
  # Az xs lista összes olyan elemének listája rs, amely
  # után vele azonos értékű elem áll
  def parosan xs do
  ...
  end
end
IO.puts(Parosan.parosan([:a, :a, :a, 2, 3, 3, :a, 2, :b, :b, 4, 4]) === [:a, :a, 3, :b, 4])
IO.puts(Parosan.parosan([:a, 2, 3, :a, 2, :b, 4]) === [])
IO.puts(Parosan.parosan([:a]) === [])
IO.puts(Parosan.parosan([]) === [])

defmodule Parosan1 do
  @spec parosan(xs :: [any()]) :: rs :: [any()]
  # Az xs lista összes olyan elemének listája rs, amely
  # után vele azonos értékű elem áll
  def parosan([x | [x | _xs] = xxs]), do: [x | parosan(xxs)]
  def parosan([_ | [_x | _xs] = xxs]), do: parosan(xxs)
  def parosan(_), do: []
```

```
end

IO.puts(Parosan1.parosan([:a, :a, :a, 2, 3, 3, :a, 2, :b, :b, 4, 4]) === [:a, :a, 3, :b, 4])
IO.puts(Parosan1.parosan([:a, 2, 3, :a, 2, :b, 4]) === [])
IO.puts(Parosan1.parosan([:a]) === [])
IO.puts(Parosan1.parosan([]) === [])

defmodule Parosan2 do
 @spec parosan(xs :: [any()]) :: rs :: [any()]
 # Az xs lista összes olyan elemének listája rs, amely
 # után vele azonos értékű elem áll
 def parosan(xs), do: parosan(xs, []) |> Split2.rev()

 @spec parosan(xs :: [any()], zs :: [any()]) :: rs :: [any()]
 def parosan([x | [x | _xs] = xxs], zs), do: parosan(xxs, [x | zs])
 def parosan([_ | [_x | _xs] = xxs], zs), do: parosan(xxs, zs)
 def parosan(_, zs), do: zs
end

IO.puts(Parosan2.parosan([:a, :a, :a, 2, 3, 3, :a, 2, :b, :b, 4, 4]) === [:a, :a, 3, :b, 4])
IO.puts(Parosan2.parosan([:a, 2, 3, :a, 2, :b, 4]) === [])
IO.puts(Parosan2.parosan([:a]) === [])
IO.puts(Parosan2.parosan([]) === [])

defmodule Parosan3 do
 @spec parosan(xs :: [any()]) :: rs :: [any()]
 # Az xs lista összes olyan elemének listája rs, amely
 # után vele azonos értékű elem áll
 def parosan([]), do: []

 def parosan(xs) do
  for {x, y} <- Enum.zip(xs, tl(xs)), x === y, do: x
 end
end

IO.puts(Parosan3.parosan([:a, :a, :a, 2, 3, 3, :a, 2, :b, :b, 4, 4]) === [:a, :a, 3, :b, 4])
IO.puts(Parosan3.parosan([:a, 2, 3, :a, 2, :b, 4]) === [])
IO.puts(Parosan3.parosan([:a]) === [])
IO.puts(Parosan3.parosan([]) === [])
```

Listában kulcs-érték párokban előforduló értékek listája

Egy listában többféle típusú és szerkezetű elem fordul elő, köztük {:v, v} párok is, ahol a párok első tagja a :v atom, második tagja az itt v-vel jelölt, tetszőleges érték.

Írjon olyan rekurzív függvényt, amelyik egy lista elemei közül az összes {:v,

v} párban található v értéket visszaadja az eredménylistában. Írjon segédfüggvényt és akkumulátort nem használó, valamint akkumulátoros segédfüggvényt használó változatokat is. Feltétlenül írjon egyéb változatokat is , pl. Enum.map/2 és Enum.filter/2 használatával, továbbá for jelöléssel (meg fog lepődni!).

```
defmodule Ertekek do
 @spec ertekek(xs :: [any()]) :: vs :: [any()]
 # Az xs lista elemei közül a {:v::atom(), v::any()} mintára illeszkedő
 # párok 2. tagjából képzett lista vs
 def ertekek(xs), do: for({:v, v} <- xs, do: v)
end
Ertekek.ertekek([:alma, {:s, 3}, {:v, 1}, 3, {:v, 2}]) === [1, 2]
```

Súgó

Ha a for komprehenzió generátorában egy mintaillesztés sikertelen, akkor az adott érték nem kerül be az eredménylistába (nem teljesült a kiválasztási feltétel), és a kiértékelés a következő listaelemmel folytatódik. Ezért a for-ral külön szűrőfeltétel használata nélkül, nagyon egyszerűen megvalósítható az elvárt működés.

```
defmodule Ertekek1 do
 @spec ertekek(xs :: [any()]) :: vs :: [any()]
 # Az xs lista elemei közül a {:v::atom(), v::any()} mintára illeszkedő
 # párok 2. tagjából képzett lista vs
 def ertekek([]), do: []
 def ertekek([{:v, v} | xs]), do: [v | ertekek(xs)]
 def ertekek([_ | xs]), do: ertekek(xs)
end

Ertekek1.ertekek([:alma, {:s, 3}, {:v, 1}, 3, {:v, 2}]) === [1, 2]

defmodule Ertekek2 do
 @spec ertekek(xs :: [any()]) :: vs :: [any()]
 # Az xs lista elemei közül a {:v::atom(), v::any()} mintára illeszkedő
 # párok 2. tagjából képzett lista vs
 def ertekek(xs), do: ertekek(xs, []) |> Split2.rev()

 @spec ertekek(xs :: [any()], zs :: [any()]) :: vs :: [any()]
 def ertekek([], zs), do: zs
 def ertekek([{:v, v} | xs], zs), do: ertekek(xs, [v | zs])
 def ertekek([_ | xs], zs), do: ertekek(xs, zs)
end

Ertekek2.ertekek([:alma, {:s, 3}, {:v, 1}, 3, {:v, 2}]) === [1, 2]

defmodule Ertekek3 do
 @spec ertekek(xs :: [any()]) :: vs :: [any()]
```

```
 # Az xs lista elemei közül a {:v::atom(), v::any()} mintára illeszkedő
 # párok 2. tagjából képzett lista vs
 def ertekek(xs) do
  Enum.filter(xs, fn
   {:v, _} -> true
   _ -> false
  end)
  |> Enum.map(fn {:v, v} -> v end)
 end
end

Ertekek3.ertekek([:alma, {:s, 3}, {:v, 1}, 3, {:v, 2}]) === [1, 2]

defmodule Ertekek4 do
 @spec ertekek(xs :: [any()]) :: vs :: [any()]
 # Az xs lista elemei közül a {:v::atom(), v::any()} mintára illeszkedő
 # párok 2. tagjából képzett lista vs
 def ertekek(xs), do: for({:v, v} <- xs, do: v)
end

Ertekek4.ertekek([:alma, {:s, 3}, {:v, 1}, 3, {:v, 2}]) === [1, 2]
```

Nehezebb feladatok listák rekurzív feldolgozására

Lista elején azonos értékű elemekből álló részlisták listája

Írjon függvényt olyan nemüres, folytonos részlisták előállítására, amelyek egy lista elejétől indulnak, és velük azonos értékű és elemszámú részlisták követik őket. Lehetőleg írjon többféle változatot, pl. akkumulátort használó és nem használó változatot, könyvtári függvényeket alkalmazó és nem alkalmazó változatot.

```
defmodule Repeated do
 @spec repeated(xs :: [any()]) :: rs :: [any()]
 def repeated(xs) do
 ...
 end
end
IO.inspect(Repeated.repeated([:a, :a, :a, 2, 3, 3, :a, :b, :b, :b, :b]) === [[:a]])
IO.inspect(Repeated.repeated([:a, :b, :b, :b, :b]) === [])
IO.inspect(Repeated.repeated([:b, :b, :b, :b]) === [[:b], [:b, :b]])
IO.inspect(Repeated.repeated([]) === [])
```

Súgó

Ha nincs jobb ötlete, használja az Enum.take/2 és Enum.drop/2 függvényeket egy segédfüggvényben.

```
defmodule Repeated1 do
```

```
  @spec repeated(xs :: [any()]) :: rs :: [any()]
  def repeated([]), do: []
  def repeated(xs), do: repeated(xs, 1)

  def repeated(xs, i) do
    cond do
      Enum.take(xs, i) == Enum.drop(xs, i) |> Enum.take(i) ->
        [Enum.take(xs, i) | repeated(xs, i + 1)]

      true ->
        []
    end
  end
end

IO.inspect(Repeated1.repeated([:a, :a, :a, 2, 3, 3, :a, :b, :b, :b, :b]) === [[:a]])
IO.inspect(Repeated1.repeated([:a, :b, :b, :b, :b]) === [])
IO.inspect(Repeated1.repeated([:b, :b, :b, :b]) === [[:b], [:b, :b]])
IO.inspect(Repeated1.repeated([]) === [])

defmodule Repeated2 do
  @spec repeated(xs :: [any()]) :: rs :: [any()]
  def repeated([]), do: []
  def repeated(xs), do: repeated(xs, 1, [])

  def repeated(xs, i, zs) do
    cond do
      Enum.take(xs, i) == Enum.drop(xs, i) |> Enum.take(i) ->
        repeated(xs, i + 1, [Enum.take(xs, i) | zs])

      true ->
        Enum.reverse(zs)
    end
  end
end

IO.inspect(Repeated2.repeated([:a, :a, :a, 2, 3, 3, :a, :b, :b, :b, :b]) === [[:a]])
IO.inspect(Repeated2.repeated([:a, :b, :b, :b, :b]) === [])
IO.inspect(Repeated2.repeated([:b, :b, :b, :b]) === [[:b], [:b, :b]])
IO.inspect(Repeated2.repeated([]) === [])
```

Listában párosával előforduló részlisták listája

Írjon függvényt egy lista összes olyan nemüres, folytonos részlistájának az előállítására, amelyet vele azonos értékű részlista követ. Lehetőleg írjon többféle változatot.

```
defmodule Stammering do
 @spec stammering(xs :: [any()]) :: zss :: [[any()]]
 # zss az xs lista összes olyan nemüres, folytonos részlistájából
 # álló lista, amelyet vele azonos értékű részlista követ
 def stammering(xs) do
   ...
 end
end
(Stammering1.stammering([:a, :a, :a, 2, 3, 3, :a, :b, :b, :b, :b]) ===
  [[:a], [:a], [3], [:b], [:b, :b], [:b], [:b]]) |> IO.puts()
IO.puts(Stammering1.stammering([]) === [])
IO.puts(Stammering1.stammering([:a]) === [])
IO.puts(Stammering1.stammering([:a, :a]) === [[:a]])
IO.puts(Stammering1.stammering([:a, :b]) === [])
```

Súgó

Javasoljuk a Tails.tails/1 használatát. Felhasználhatja a Repeated1.repeated/1 vagy a Repeated2.repeated/1függvényt is.

```
defmodule Stammering1 do
 @spec stammering(xs :: [any()]) :: zss :: [[any()]]
 # zss az xs lista összes olyan nemüres, folytonos részlistájából
 # álló lista, amelyet vele azonos értékű részlista követ
 def stammering(xs) do
   for zs <- Tails.tails(xs) do
     # IO.inspect(zs)
     Repeated1.repeated(zs)
   end
   |> Enum.concat()
 end
end

(Stammering1.stammering([:a, :a, :a, 2, 3, 3, :a, :b, :b, :b, :b]) ===
   [[:a], [:a], [3], [:b], [:b, :b], [:b], [:b]])
|> IO.puts()

IO.puts(Stammering1.stammering([]) === [])
IO.puts(Stammering1.stammering([:a]) === [])
IO.puts(Stammering1.stammering([:a, :a]) === [[:a]])
IO.puts(Stammering1.stammering([:a, :b]) === [])
```

További műveletek listákkal

For-jelölés, névtelen függvény, kifejezések

for-jelölés (for comprehension), kifejezések, függvények

Az eddigi feladatok között is voltak olyanok, amelyek megoldásához könyvtári függvényeket, for-jelölést vagy magasabb rendű függvényeket lehetett használni. Most kifejezetten kérjük a for-jelölés, a magasabb rendű függvények és más könyvtári függvények alkalmazását.

Korábban nevesített függvények megírását kértük (def és defp deklarációval), ehhez modulokat is kellett definiálni. A iex Elixir-interpretert futtató Livebookban azonban modul definiálása nélkül is lehet kifejezéseket kiértékeltetni, többek között névtelen függényeket változóhoz kötni. Lássunk egy kis példát!

Kis példák névtelen függvények definiálására

Először egy kis névtelen segédfüggvényt írunk annak eldöntésére, hogy a paramétere magánhangzó-e a magyar ábécé szerint (nagy- és kisbetű között nem teszünk különbséget). A paramétert egykarakteres sztringként adjuk át.

Sajnos, specifikációt csak függvénydefinícióhoz lehet írni, ezért itt kommentbe tesszük, hogy segítse a megértést.

```
# @spec is_vowel(string()) :: boolean()
is_vowel = fn c -> String.contains?("aáeéiíoóöőuúüű", String.downcase(c)) end
IO.puts(is_vowel.("É"))
IO.puts(is_vowel.("ő"))
IO.puts(is_vowel.("V"))
```

Emlékezzünk rá, hogy egy névtelen függvény vagy egy névhez kötött (de nem def vagy defp deklarációval létrehozott) függvény alkalmazásakor a függvényérték és a paraméter közé egy .-ot kell tenni.

Mivel most néhány példamegoldást mutatunk be, két cellában is ugyanaz a (vagy nagyon hasonló) programkód van. A felső, halványzöld keretes cella szövege nem írható át (csak szerkesztő módban), nem futtatható, de másolható. Az alsó cellában a kód átírható, futtatható. Javasoljuk, hogy ne csak futtassa, hanem módosítsa is a kódot, írjon más, esetleg jobb megoldásokat.

```
is_vowel = fn c -> String.contains?("aáeéiíoóöőuúüű", String.downcase(c)) end
IO.puts(is_vowel.("É"))
IO.puts(is_vowel.("ő"))
IO.puts(is_vowel.("V"))
```

```
true
true
false

:ok
```

Most írjunk olyan névtelen függvényt, amelyik az egykarakteres sztringként megkapott paraméterét, ha az magánhangzó, nagybetűssé, ha mássalhangzó, kisbetűssé konvertálja.

A már bevezetett változókat használhatjuk a további cellákban.

```
# @spec vow_up_cons_down(string()) :: string()
vow_up_cons_down = fn c -> if is_vowel.(c), do: String.upcase(c), else: String.downcase(c) end
IO.puts(vow_up_cons_down.("É"))
IO.puts(vow_up_cons_down.("ő"))
IO.puts(vow_up_cons_down.("V"))
```

```
vow_up_cons_down = fn c -> if is_vowel.(c), do: String.upcase(c), else: String.downcase(c) end
IO.puts(vow_up_cons_down.("É"))
IO.puts(vow_up_cons_down.("ő"))
IO.puts(vow_up_cons_down.("V"))
```

```
É
Ő
v
```

```
:ok
```

Most írjunk olyan névtelen függvényt for jelöléssel, amelyik a sztringként megkapott paraméterében az összes magánhangzót nagybetűssé, az összes mássalhangzót kisbetűssé konvertálja.

```
# @spec my_conv(string()) :: string()
my_conv = fn s ->
  for(c <- to_charlist(s), do: vow_up_cons_down.(to_string([c]))) |> Enum.join()
end
IO.puts(my_conv.("hátasló"))
IO.puts(my_conv.("hatásvizsgálat"))
IO.puts(my_conv.("Üllői úti fák"))
IO.puts(my_conv.("Táncórák Idősebbeknek Május 35-étől"))
```

```
my_conv = fn s ->
  for(c <- to_charlist(s), do: vow_up_cons_down.(to_string([c]))) |> Enum.join()
end


IO.puts(my_conv.("hátasló"))
IO.puts(my_conv.("hatásvizsgálat"))
IO.puts(my_conv.("Üllői úti fák"))
IO.puts(my_conv.("Táncórák Idősebbeknek Május 35-étől"))
```

```
hÁtAslÓ
hAtÁsvIzsgÁlAt
ÜllŐI ÚtI fÁk
tÁncÓrÁk IdŐsEbbEknEk mÁjUs 35-ÉtŐl
```

```
:ok
```

A névtelen függvény definiálásának van tömörebb változata is:

```
# @spec my_conv(string()) :: string()
my_conv = &(for(c <- to_charlist(&1), do: vow_up_cons_down.(to_string([c]))) |> Enum.join())
IO.puts(my_conv.("hátasló"))
IO.puts(my_conv.("hatásvizsgálat"))
IO.puts(my_conv.("Üllői úti fák"))
IO.puts(my_conv.("Táncórák Idősebbeknek Május 35-étől"))
```

```
my_conv = &(for(c <- to_charlist(&1), do: vow_up_cons_down.(to_string([c]))) |> Enum.join())
IO.puts(my_conv.("hátasló"))
IO.puts(my_conv.("hatásvizsgálat"))
IO.puts(my_conv.("Üllői úti fák"))
IO.puts(my_conv.("Táncórák Idősebbeknek Május 35-étől"))
```

```
hÁtAslÓ
hAtÁsvIzsgÁlAt
ÜllŐI ÚtI fÁk
tÁncÓrÁk IdŐsEbbEknEk mÁjUs 35-ÉtŐl

:ok
```

Természetes szám valódi osztói

Egy természetes szám valódi osztóinak nevezzük az 1-en és önmagán kívüli pozitív osztóit.

Írjon olyan kifejezést/függvényt a for jelölés felhasználásával, amelyik egy listában visszaadja a paraméterként átadott természetes szám valódi osztóit.

```
# @spec proper_divisors(i :: integer()) :: ds :: [integer()]
# Az i természetes szám valódi osztóinak listája ds
proper_divisors = ...
(proper_divisors.(10) === [2, 5]) |> IO.inspect(charlists: :as_list)
(proper_divisors.(23) === []) |> IO.inspect(charlists: :as_list)
(proper_divisors.(48) === [2, 3, 4, 6, 8, 12, 16, 24]) \
|> IO.inspect(charlists: :as_list)
(proper_divisors.(128) === [2, 4, 8, 16, 32, 64]) \
|> IO.inspect(charlists: :as_list)
```

Súgó

Egy k természetes szám valódi osztóit a legegyszerűbben úgy találhatja meg, hogy a k-t rendre elosztja a 2 és $k/2$ közötti egészekkel, és ha az egészosztásnak nincs maradéka, akkor az adott szám osztója k-nak.

```
proper_divisors = &for j <- 2..div(&1, 2), rem(&1, j) === 0, do: j
(proper_divisors.(10) === [2, 5]) |> IO.inspect(charlists: :as_list)
(proper_divisors.(23) === []) |> IO.inspect(charlists: :as_list)
```

```
(proper_divisors.(48) === [2, 3, 4, 6, 8, 12, 16, 24]) |> IO.inspect(charlists: :as_list)
(proper_divisors.(128) === [2, 4, 8, 16, 32, 64]) |> IO.inspect(charlists: :as_list)
```

```
true
true
true
true
```

```
true
```

Összetett számok

Összetett számnak nevezzük azt a természetes számot, amelynek van valódi osztója. A legkisebb összetett szám a 4.

Írjon olyan kifejezést/függvényt a for jelölés felhasználásával, amelyik egy listában visszaadja a függvénynek paraméterként átadott természetes számnál nem nagyobb összes összetett számot, 4-től kezdve. A függvényben felhasználhatja a szám valódi osztóit előállító függvényt, amit az előbb írt meg.

```
# @spec composite_numbers(i :: integer()) :: ns :: [integer()]
# Az i-nél nem nagyobb összetett számok listája ns
composite_numbers =
(composite_numbers.(11) === [4, 6, 8, 9, 10]) |> IO.inspect(charlists: :as_list)
(composite_numbers.(17) === [4, 6, 8, 9, 10, 12, 14, 15, 16]) \
|> IO.inspect(charlists: :as_list)
```

```
composite_numbers = &for j <- 4..&1, length(proper_divisors.(j)) > 0, do: j
(composite_numbers.(11) === [4, 6, 8, 9, 10]) |> IO.inspect(charlists: :as_list)
(composite_numbers.(17) === [4, 6, 8, 9, 10, 12, 14, 15, 16]) |> IO.inspect(charlists: :as_list)
```

```
true
true
```

```
true
```

Hatékonyabb megoldás Írjon olyan megoldást az előző feladatra, amelyik nem állítja elő a valódi osztók listáját.

Az n természetes szám összetett, ha osztható

1. a 2 és $\sqrt{n}$ közötti prímszámok bármelyikével;
2. 2-vel vagy a 3 és $\sqrt{n}$ közötti páratlan egészek bármelyikével.

Az első módszer gyorsabb, de ha a prímszámok előállítása nem triviális, elég hatékony a második módszer is. További hatékonyságnövelő lehetőségekről olvashat pl. itt: pl. https://en.wikipedia.org/wiki/Primality_test.

```
composite_numbers_faster = &for i <- 4..&1, composite?.(i), do: i
(composite_numbers_faster.(11) === [4, 6, 8, 9, 10] ) \
|> IO.inspect(charlists: :as_list)
```

```
(composite_numbers_faster.(21) === [4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21])\
|> IO.inspect(charlists: :as_list)
```

Súgó

Javasoljuk, hogy a megoldást bontsa lépésekre: állítsa elő a szám összetett voltának vizsgálatához használandó osztókat; állapítsa meg, hogy a szám összetett szám-e; állítsa elő az összetett számok listáját a kért tartományban. Fontolja meg az alább felsorolt függvények használatát.

A Kernel modulban definiált ../// 3 operátor operandusainak egész számoknak kell lenniük, ezért ha a felső határ beállítására a :math.sqrt/1 függvényt használja, egész számmá kell konvertálni (vö. round/1, floor/1, ceil/1 függvények).

Az Enum.to_list/1 függvény tetszőleges felsorolható sorozatot alakít át listává, így tartomány-típusú értéksorozatot is.

Az Enum.reduce/3 függvény egy lista összes elemére alkalmaz egy kétoperandusú függvényt: ha pl. egy szám összetett voltát akarjuk vizsgálni vele, akkor olyan függvényt kell átadnunk paraméterként, amelyik az adott számnak a sorozat egy elemével való oszthatósága esetén igaz, egyébként hamis értékkel tér vissza.

```
# @spec probes(i :: integer()) :: ps :: [integer()]
# A 2-t, továbbá a 3 és a :math.sqrt(i) közé eső egészeket tartalmazó lista ps
probes = ...
# @spec composite?(i :: integer()) :: b :: boolean()
# b igaz, ha i összetett szám
composite? = ...
# @spec composite_numbers_faster(i :: integer()) :: ns :: [integer()]
# Az i-nél nem nagyobb összetett számok listája ns
composite_numbers_faster = ...

probes = &[2 | Enum.to_list(3..floor(:math.sqrt(&1))//2)]
composite? = &Enum.reduce(probes.(&1), false, fn x, y -> rem(&1, x) == 0 or y end)
composite_numbers_faster = &for i <- 4..&1, composite?.(i), do: i
(composite_numbers_faster.(11) === [4, 6, 8, 9, 10]) |> IO.inspect(charlists: :as_list)

(composite_numbers_faster.(21) === [4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21])
|> IO.inspect(charlists: :as_list)
```

```
true
true

true
```

A megoldást, ha a javaslatunkat megfogadta, három kis – esetleg több – lépésre bontotta, mindegyikre egy-egy névtelen (de változóhoz kötött) függvényt írt, így ezeket könnyebb volt megérteni, és a helyességüket könnyebb volt belátni. Mivel a funkcionális nyelvekben a függvényhívás nagyon hatékonyan van megoldva, ez

a jó és követendő gyakorlat: minden kicsit is összetett függvényt több egyszerűbb függvényből rakjunk össze, és ahol csak lehet, használjuk a hatékonyan megvalósított és sokat tesztelt könyvtári függvényeket.