DP23a - Funkcionális programozás, 1. 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.

Rekurzió

A deklaratív programozás alappillére a rekurzió. A rekurzió kétféle lehet: lineáris és elágazó (angolul linear recursion, tree recursion). Lineárisan rekurzív adatszerkezet a lista (láncolt lista, nem egydimenziós tömb!), elágazóan rekurzív a (bináris vagy több ágú) fa, ezek feldolgozására értelemszerűen rekurzív algoritmusokat írunk. De rekurzív algoritmusokat kell használnunk iterációk megvalósítására is ciklusok helyett, mivel az utóbbiak a deklaratív nyelvekben ismeretlen konstrukciók.

Lineáris rekurzió, jobb- és balrekurzió

Írjon lineárisan rekurzív függvényeket az alábbi feladatok megoldására. Írjon többféle függvényváltozatot, először direkt rekurzióval, majd esetleg könyvtári függvények használatával. Mindig törekedjen elegáns, tömör, érthető és hatékony függvények írására.

Kiírás a rekurzív hívás előtt

Írjon olyan rekurzív függvényt upto_by_3 néven, amelyik növekvő sorrendben kiírja az 1 és n közé eső, n-nél nem nagyobb, 3-mal osztható természetes számokat! Az n-et paraméterként adja át a függvénynek. A rekurzív hívás az adott klóz utolsó hívása, eredménye az adott klóz eredménye legyen, azaz a rekurzív hívás eredményével már ne végezzen semmilyen műveletet: a soron következő számot tehát a rekurzív hívás előtt írja ki. Segédfüggvényt definiálhat.

```
defmodule UptoBy3 do
 @spec upto_by_3(n :: integer()) :: :ok
 def upto_by_3(n) do
  IO.puts(i)
  ...
 end
end
UptoBy3.upto_by_3(20)
```

```
defmodule UptoBy31 do
 @spec upto_by_3(n :: integer()) :: :ok
 def upto_by_3(n), do: upto_by_3(n, 3)

 @spec upto_by_3(n :: integer(), i :: integer()) :: :ok
 def upto_by_3(n, i) when i <= n do
  IO.puts(i)
  upto_by_3(n, i + 3)
 end

 def upto_by_3(_, _), do: :ok
end

UptoBy31.upto_by_3(20)
```

Mint már bizonyára tudja, jobbrekurziónak (terminális, ritkábban farokrekurziónak - angolul: tail recursion) nevezzük a rekurzív hívást, ha egy klóz egyetlen és utolsó hívása, és az eredményével már nem végzünk semmilyen műveletet (a visszaadáson kívül). A jobbrekurzív kódot a modern értelmező- és fordítóprogramok nagyon hatékonyan, iteratív processzként valósítják meg.

Kiírás a rekurzív hívás után

Írja át előző megoldását úgy, hogy a rekurzív hívás az adott klóz első hívása legyen, azaz a rekurzív hívás előtt ne végezzen semmilyen műveletet: a soron következő számot tehát a rekurzív hívás után írja ki. Segédfüggvényt definiálhat.

```
defmodule UptoBy3 do
 @spec upto_by_3(n :: integer()) :: :ok
 def upto_by_3(n) do
```

```
  ...
  IO.puts(i)
 end
end
UptoBy3.upto_by_3(20)

defmodule UptoBy32 do
 @spec upto_by_3(n :: integer()) :: :ok
 def upto_by_3(n), do: upto_by_3(n, 3)

 @spec upto_by_3(n :: integer(), i :: integer()) :: :ok
 def upto_by_3(n, i) when i <= n do
  upto_by_3(n, i + 3)
  IO.puts(i)
 end

 def upto_by_3(_, _), do: :ok
end

UptoBy32.upto_by_3(20)
```

Vesse össze a két függvényalkalmazás által kiírt számsorozatot! Miben különbözik a kétféle megoldás veremhasználata?

A második változatban alkalmazott rekurzív hívást balrekurziónak (fejrekurziónak - angolul: head recursion) nevezzük: a balrekurzív hívás egy klóz első és egyetlen rekurzív hívása.

A második változata valószínűleg nem teljesíti a specifikációt, ti. hogy növekvő sorrendben kell kiírni a számokat. Ezen úgy segíthet, hogy nem 1-től felfelé halad a generáláskor, hanem n-től lefelé.

Ha őrt (when ...) használt volna, helyette inkább mintaillesztést használjon az alapeset felismerésére. Lehetőleg használjon segédfüggvényt. Írja meg ezt a harmadik változatot is!

```
defmodule UptoBy33 do
 @spec upto_by_3(n :: integer()) :: :ok
 def upto_by_3(n), do: downto_by_3(div(n, 3))

 @spec downto_by_3(i :: integer()) :: :ok
 def downto_by_3(0), do: :ok

 def downto_by_3(i) do
  downto_by_3(i - 1)
  IO.puts(i * 3)
 end
end
```

```
UptoBy33.upto_by_3(20)
```

Legnagyobb közös osztó (greatest common divisor) euklideszi algoritmussal

```
defmodule Gcd do
 @spec gcd(a :: integer(), b :: integer()) :: d :: integer()
 # a és b legnagyobb közös osztója d
 def gcd(a, b) do
 ...
 end
end
IO.puts(Gcd.gcd(96, 42) === 6)
IO.puts(r2 = Gcd.gcd(90, 45))
```

Súgó

Ha $a = b \cdot q + r$, akkor $gcd(a, b) = gcd(b, r)$, ahol a, b, q és r egész számok.

Írjon több változatot, pl. kivonással vagy maradékos osztással (rem/2). Gondoljon arra, hogy a függvény első paramétere nagyobb is, kisebb is lehet a másodiknál.

```
defmodule Gcd do
 @spec gcd(a :: integer(), b :: integer()) :: d :: integer()
 # a és b legnagyobb közös osztója d
 def gcd(a, 0), do: a
 def gcd(a, b) when a < b, do: gcd(b, a)
 def gcd(a, b), do: gcd(b, a - b)
end

IO.puts(Gcd.gcd(96, 42) === 6)
IO.puts(r2 = Gcd.gcd(90, 45))
```

3-mal osztható egész számok listája

Írja meg (esetleg otthoni gyakorló feladatként) az upto_by_3/1 függvény olyan jobb- és balrekurzív változatait is upto_by_3_to_list/1 néven, amelyek listát adnak eredményül.

Ne használja a ../2 és ../3függvényeket az upto_by_3_to_list/1 függvény megvalósítására! (De használhatja az eredmény ellenőrzésére.)

```
defmodule UptoBy3ToList do
 @spec upto_by_3_to_list(n :: integer()) :: ns :: [integer()]
 # Az 1 és n közé eső, 3-mal osztható egész számok listája ns
 def upto_by_3_to_list(n) do
  ...
```

```
 end
end
(UptoBy3ToList.upto_by_3_to_list(20) === [3, 6, 9, 12, 15, 18]) |> IO.puts

defmodule UptoBy3ToList1 do
 @spec upto_by_3_to_list(n :: integer()) :: ns :: [integer()]
 # Az 1 és n közé eső, 3-mal osztható egész számok listája ns
 def upto_by_3_to_list(n), do: upto_by_3_to_list(n, 3)

 @spec upto_by_3_to_list(n :: integer(), i :: integer()) :: ns :: [integer()]
 def upto_by_3_to_list(n, i) when i <= n, do: [i | upto_by_3_to_list(n, i + 3)]
 def upto_by_3_to_list(_n, _i), do: []
end

(UptoBy3ToList1.upto_by_3_to_list(20) === [3, 6, 9, 12, 15, 18]) |> IO.puts()

defmodule UptoBy3ToList2 do
 @spec upto_by_3_to_list(n :: integer()) :: ns :: [integer()]
 # Az 1 és n közé eső, 3-mal osztható egész számok listája ns
 def upto_by_3_to_list(n), do: upto_by_3_to_list(n, 3, [])

 @spec upto_by_3_to_list(n :: integer(), i :: integer(), zs :: [integer()]) :: ns :: [integer()]
 def upto_by_3_to_list(n, i, zs) when i <= n, do: upto_by_3_to_list(n, i + 3, [i | zs])
 def upto_by_3_to_list(_n, _i, zs), do: zs
end

UptoBy3ToList2.upto_by_3_to_list(20)

defmodule UptoBy3ToList3 do
 @spec upto_by_3_to_list(n :: integer()) :: ns :: [integer()]
 # Az 1 és n közé eső, 3-mal osztható egész számok listája ns
 def upto_by_3_to_list(n), do: upto_by_3_to_list(div(n, 3) * 3, [])

 @spec upto_by_3_to_list(n :: integer(), zs :: [integer()]) :: ns :: [integer()]
 def upto_by_3_to_list(i, zs) when i >= 3, do: upto_by_3_to_list(i - 3, [i | zs])
 def upto_by_3_to_list(_i, zs), do: zs
end

(UptoBy3ToList3.upto_by_3_to_list(20) === [3, 6, 9, 12, 15, 18]) |> IO.puts()
```

Sokféle egyéb lehetőség is van a megoldásra könyvtári függvények, köztük magasabb rendű függvények vagy a for-jelölés alkalmazásával. Keressen minél tömörebb, jobb, ugyanakkor könnyen érthető, magyarázható megoldásokat.

Súgó

Néhány megoldási lehetőség:

```
1..div(20,3) |> Enum.map(&(Kernel.*(&1,3)))
```

```
for i <- 1..div(20,3), do: 3*i
for i <- 1..20, rem(i,3) === 0, do: i
3..20//3 |> Enum.to_list()
for i <- 3..20//3, do: i
```

Egyszerű feladatok listák rekurzív feldolgozása

Lista hossza

Írjon rekurzív függvényt egy lista hosszának meghatározására! Ne használjon segédfüggvényt! (A feladat szerepelt az előadáson, de gyakorlásképpen most írja meg önállóan, ne nézze meg a megoldást.)

Rekurzív adatszerkezetek feldolgozásának természetes módja a rekurzív algoritmus. Lineáris adatszerkezetek pl. listák feldolgozására (imperatív nyelveken) még csak lehet ciklust írni, de elágazóan rekurzív adatszerkezeteket, pl. fákat ciklusokkal bejárni már nagy kihívás.

Amikor algoritmust írunk egy rekurzív adatszerkezet feldolgozására, akkor legalább két, esetleg több klózt írunk. Közöttük vanank olyanok, amelyekre az adatszerkezet jellegzetessége miatt csak egyszer vagy csak nagyon ritkán, másokra gyakrabban kerül sor. Ilyen például az üres és a nemüres lista esete: az üres listát feldolgozó klóz kiértékelésére csak egyszer kerül sor, a nemüres listát feldolgozó klózt többször, a lista hosszától függően esetleg nagyon sokszor meghívjuk.

Az algoritmus hatékonyságát javítja, ha

egy függvény klózai kölcsönösen kizárják egymást, és

közülük a gyakrabban hívott(ak) megelőzi(k) a ritkábban hívott(ak)at.

A length/1 függvény esetében ez azt jelenti, hogy az első a legalább egy elemű listákra, a második pedig az üres listákra illeszkedő klóz legyen.

```
def fun([x|xs])...
def fun([])...
```

Ne használja a Kernel.length/1 függvényt!

```
defmodule Length do
 @spec len(xs: [any()]) :: n :: integer()
 # Az xs lista hossza n
 def len(n) do
 ...
 end
end
IO.puts(Length.len([]) == 0)
IO.puts(Length.len(Range.to_list(1..5)) == 5)
IO.puts(Length.len(~c"kőszerű") == 7)
```

```elixir
defmodule Length do
 @spec len(xs: [any()]) :: n :: integer()
 # Az xs lista hossza n
 def len([_ | xs]), do: len(xs) + 1
 def len([]), do: 0
end

IO.puts(Length.len([]) == 0)
IO.puts(Length.len(Range.to_list(1..5)) == 5)
IO.puts(Length.len(~c"kőszerű") == 7)
```

A lista hosszát most jobbrekurzív függvénnyel, akkumulátort és segédfüggvényt használva írja meg! (Ez is szerepelt az előadáson, de gyakorlásképpen most írja meg önállóan, ne nézze meg a megoldást.)

```elixir
defmodule Length2 do
 @spec len(xs :: [any()]) :: n :: integer()
 # Az xs lista hossza n
 def len(xs), do: len(xs, 0)

 @spec len(xs :: [any()], count :: integer()) :: n :: integer()
 # Az xs lista hossza és count összege n
 def len([_ | xs], cnt), do: len(xs, cnt + 1)
 def len([], cnt), do: cnt
end

IO.puts(Length2.len([]) == 0)
IO.puts(Length2.len(Range.to_list(1..5)) == 5)
IO.puts(Length2.len(~c"kőszerű") == 7)
```

Lista utolsó eleme Erlang, illetve Elixir-stílusú eredmény- és hibajelzéssel

Amikor egy listán műveleteket végzünk, gyakran előfordul, hogy bizonyos listákon bizonyos műveleteket nem lehet elvégezni. Egy üres listának értelemszerűen egyetlen eleme sincs, így pl. az utolsó elemét sem lehet visszaadni. Ilyenkor dönthetünk úgy, hogy az adott műveletet üres listára nem értelmezzük, és rábízzuk a rendszerre a hiba jelzését. Ha úgy döntünk, hogy jelezzük a helyes eredmény mellett a hibát is, akkor ezt hagyományosan kétféle stílusban tehetjük meg:

Erlang-stílusban a visszatérési érték típusa {:ok, any()} | :error, azaz siker esetén a visszatérési érték egy {:ok, value} pár, ahol egy any() típusú value a visszaadott érték, meghiúsulás esetén pedig az :error atom;

Elixir-stílusban a visszatérési érték típusa any() | nil, azaz siker esetén az any() típusú value visszaadott érték, meghiúsulás esetén pedig a nil atom.

Ne használja a List.last függvényeket!

Esetek megkülönböztetése kölcsönös kizárással A last/1 függvénynek, amennyiben az üres listát nem kell kezelnie, két esetet kell megkülönböztetnie: azt, amikor a listának pontosan egy eleme van, és azt, amikor a listának legalább egy eleme van. A korábban látott sémát csak kicsit kell módosítanunk: a második klóznak nem az üres listára, hanem az egyelemű listára kell illeszkednie.

```
def fun([x|xs])...
def fun([x])...
```

Csakhogy ezzel van egy kis gond: az első klóz minden olyan listára illeszkedik, amelyiknek van feje, a farka pedig tetszőleges elemszámú, azaz üres is lehet. Emiatt az első klóz az egyelemű listára is illeszkedik, azaz a második klózra soha nem kerül sor – a minták nem egymást kölcsönösen kizáróak. (Erre figyelmeztet is az Elixir fordító). A két klóz sorrendjének megfordításával a mintaillesztés a két esetet meg tudja különböztetni, ám ennek hatékonyságromlás az ára.

```
def fun([x])...
def fun([x|xs])...
```

Lehet azonban olyan mintát is írni, amelyik legalább két elemű listákra illeszkedik, és ezáltal kölcsönös kizárásban van a pontosan egy egyelemű listára illeszkedő mintával:

```
def fun([x1,x2|xs])...
def fun([x])...
```

Az első klóz törzsében a lista fejére az x1 változóval, a lista farkára az [x2|xs] kifejezéssel hivatkozhatunk. Az utóbbi hivatkozást egyszerűbbé (és olcsóbbá!) tehetjük az ún. réteges mintával (layered pattern):

```
def fun([x1 | xxs = [_x2|_xs]])...
def fun([x])...
```

A lista farkára az xxs változóval lehet hivatkozni. Az aláhúzásjellel kezdődő nevek helyett elég aláhúzásjelet írni, a beszédes nevek azonban segítik a megértést, a változó szerepére utalnak. Ha a lista második elemére vagy a harmadik elemtől kezdődő farkára akarunk hivatkozni a klóz törzsében, akkor ne aláhúzásjellel kezdődő változóneveket használjunk.

Először is írjon egy olyan függvényt, amelyik visszadja egy lista utolsó elemét, de a hibajelzést rábízza a rendszerre.

```
defmodule Last do
 @spec last(xs :: [any()]) :: x :: any()
 # ha xs nem üres, az utolsó eleme x
 def last(xs) do
 ...
 end
```

```
end
IO.puts(Last.last(~c"Élvezed?") == ??)
IO.puts(Last.last([]))

defmodule Last do
 @spec last(xs :: [any()]) :: x :: any()
 # ha xs nem üres, az utolsó eleme x
 def last([_ | [_ | _] = xs]), do: last(xs)
 def last([x]), do: x
end

IO.puts(Last.last(~c"Élvezed?") == ??)
IO.puts(Last.last([]))
```

Most írja meg a függvényt Erlang-stílusú hibakezeléssel!

```
defmodule LastEr do
 @spec last(xs :: [any()]) :: r :: {:ok, x :: any()} | :error
 # Ha xs nem üres, r == {:ok, x}, ahol az xs utolsó eleme x, egyébként r == :error
 def last(xs) do
 ...
 end
end
IO.puts(LastEr.last(~c"Élvezed?") == {:ok, ??})
IO.puts(LastEr.last([]) == :error)

defmodule LastEr do
 @spec last(xs :: [any()]) :: r :: {:ok, x :: any()} | :error
 # Ha xs nem üres, r == {:ok, x}, ahol az xs utolsó eleme x, egyébként r == :error
 def last([_ | xs = [_ | _]]), do: last(xs)
 def last([x]), do: {:ok, x}
 def last([]), do: :error
end

IO.puts(LastEr.last(~c"Élvezed?") == {:ok, ??})
IO.puts(LastEr.last([]) == :error)
```

Most pedig írja meg Elixir-stílusú hibakezeléssel!

```
defmodule LastEx do
 @spec last(xs :: [any()]) :: r :: (x :: any()) | nil
 # Ha xs nem üres, r == x, ahol az xs utolsó eleme x, egyébként r == nil
 def last(xs) do
 ...
 end
end
IO.puts(LastEx.last(~c"Élvezed?") == ??)
IO.puts(LastEx.last([]) == nil)
```

```
defmodule LastEx do
 @spec last(xs :: [any()]) :: r :: (x :: any()) | nil
 # Ha xs nem üres, r == x, ahol az xs utolsó eleme x, egyébként r == nil
 def last([_ | [_ | _] = xs]), do: last(xs)
 def last([x]), do: x
 def last([]), do: nil
end

IO.puts(LastEx.last(~c"Élvezed?") == ??)
IO.puts(LastEx.last([]) == nil)
```

Mátrix első oszlopában lévő elemek összegyűjtése

Egy $n*m$ $(n, m >= 0)$ méretű mátrixot listák listájaként, sorfolytonosan ábrázolunk. Írjon olyan függvényt, amelyik egy listába gyűjti a mátrix első oszlopában lévő elemeket, az eredeti sorrendet megőrizve. Írjon több változatot, pl. (1) saját rekurzív függvényt, (2) az Enum.map/2 magasabb rendű függvénnyel, (3) for-komprehenzióval. Használjon mintaillesztést, ahol csak lehet!

```
defmodule FirstCol do
 @spec first_col(xss :: [[any()]]) :: zs :: [any()]
 def first_col(xss) do
 ...
 end
end
xss = [[A,B,C],[:a,:b,:c],[1,2,3],[3.2,2.1,1.0]]
(FirstCol.first_col(xss) === [A, :a, 1, 3.2]) |> inspect() |> IO.puts()
(FirstCol.first_col([]) === []) |> inspect() |> IO.puts()
```

```
defmodule FirstCol1 do
 @spec first_col(xss :: [[any()]]) :: zs :: [any()]
 def first_col([]), do: []
 def first_col([[x | _xs] | xss]), do: [x | first_col(xss)]
end

xss = [[A, B, C], [:a, :b, :c], [1, 2, 3], [3.2, 2.1, 1.0]]
(FirstCol1.first_col(xss) === [A, :a, 1, 3.2]) |> inspect() |> IO.puts()
(FirstCol1.first_col([]) === []) |> inspect() |> IO.puts()
```

```
defmodule FirstCol2 do
 @spec first_col(xss :: [[any()]]) :: zs :: [any()]
 def first_col(xss) do
   Enum.map(xss, &hd(&1))
 end
end

xss = [[A, B, C], [:a, :b, :c], [1, 2, 3], [3.2, 2.1, 1.0]]
```

```
(FirstCol2.first_col(xss) === [A, :a, 1, 3.2]) |> inspect() |> IO.puts()
(FirstCol2.first_col([]) === []) |> inspect() |> IO.puts()

defmodule FirstCol3 do
 @spec first_col(xss :: [[any()]]) :: zs :: [any()]
 def first_col(xss) do
  for [x | _xs] <- xss, do: x
 end
end

xss = [[A, B, C], [:a, :b, :c], [1, 2, 3], [3.2, 2.1, 1.0]]
(FirstCol3.first_col(xss) === [A, :a, 1, 3.2]) |> inspect() |> IO.puts()
(FirstCol3.first_col([]) === []) |> inspect() |> IO.puts()
```

Listák listája első elemeinek összegyűjtése

Egy listában különböző hosszúságú listák vannak, ezek üresek is lehetnek. Írjon olyan függvényt, amelyik egy listába gyűjti a nemüres listák első elemeit, az eredeti sorrendet megőrizve. Írjon több változatot, pl. (1) saját rekurzív függvényt, (2) az Enum.map/2, Enum.filter/2 vagy más magasabb rendű függvényekkel, (3) for-komprehenzióval. Használjon mintaillesztést, ahol csak lehet!

```
defmodule FirstElem do
 @spec first_elem(xss :: [[any()]]) :: zs :: [any()]
 def first_elem(xss) do
 ...
 end
end
xss = [[A,B,C],[:a,:b,:c],[1,2,3],[3.2,2.1,1.0]]
(FirstElem.first_elem(xss) === [A, :a, 1, 3.2]) |> inspect() |> IO.puts()
xss = [[A,B],[:a,:b,:c,:d],[1],[],[3.2,2.1,1.0]]
(FirstElem.first_elem(xss) === [A, :a, 1, 3.2]) |> inspect() |> IO.puts()
(FirstElem.first_elem([]) === []) |> inspect() |> IO.puts()

defmodule FirstElem1 do
 @spec first_elem(xss :: [[any()]]) :: zs :: [any()]
 def first_elem([]), do: []
 def first_elem([[] | xss]), do: first_elem(xss)
 def first_elem([[x | _xs] | xss]), do: [x | first_elem(xss)]
end

xss = [[A, B, C], [:a, :b, :c], [1, 2, 3], [3.2, 2.1, 1.0]]
(FirstElem1.first_elem(xss) === [A, :a, 1, 3.2]) |> inspect() |> IO.puts()
xss = [[A, B], [:a, :b, :c, :d], [1], [], [3.2, 2.1, 1.0]]
(FirstElem1.first_elem(xss) === [A, :a, 1, 3.2]) |> inspect() |> IO.puts()
(FirstElem1.first_elem([]) === []) |> inspect() |> IO.puts()

defmodule FirstElem2 do
```

```
 @spec first_elem(xss :: [[any()]]) :: zs :: [any()]
 def first_elem(xss) do
   xss
   |> Enum.filter(&(&1 != []))
   |> Enum.map(&hd(&1))
 end
end

xss = [[A, B, C], [:a, :b, :c], [1, 2, 3], [3.2, 2.1, 1.0]]
(FirstElem2.first_elem(xss) === [A, :a, 1, 3.2]) |> inspect() |> IO.puts()
xss = [[A, B], [:a, :b, :c, :d], [1], [], [3.2, 2.1, 1.0]]
(FirstElem2.first_elem(xss) === [A, :a, 1, 3.2]) |> inspect() |> IO.puts()
(FirstElem2.first_elem([]) === []) |> inspect() |> IO.puts()

defmodule FirstElem3 do
 @spec first_elem(xss :: [[any()]]) :: zs :: [any()]
 def first_elem(xss) do
   for [x | _xs] <- xss, do: x
 end
end

xss = [[A, B, C], [:a, :b, :c], [1, 2, 3], [3.2, 2.1, 1.0]]
(FirstElem3.first_elem(xss) === [A, :a, 1, 3.2]) |> inspect() |> IO.puts()
xss = [[A, B], [:a, :b, :c, :d], [1], [], [3.2, 2.1, 1.0]]
(FirstElem3.first_elem(xss) === [A, :a, 1, 3.2]) |> inspect() |> IO.puts()
(FirstElem3.first_elem([]) === []) |> inspect() |> IO.puts()
```

Módosítsa úgy a megoldásait, hogy üres lista esetén a lista feje helyett a nil atomot rakja be az eredménylistába.

```
defmodule FirstElem do
 @spec first_elem(xss :: [[any()]]) :: zs :: [any() | nil]
 def first_elem(xss) do
 ...
 end
end
xss = [[A,B],[:a,:b,:c,:d],[1],[],[3.2,2.1,1.0]]
(FirstElem.first_elem(xss) === [A,:a,1,nil,3.2]) |> inspect() |> IO.puts()
```

Melyik verzióját a legkönnyebb módosítania?

```
defmodule FirstElem4 do
 @spec first_elem(xss :: [[any()]]) :: zs :: [any()]
 def first_elem([]), do: []
 def first_elem([[] | xss]), do: [nil | first_elem(xss)]
 def first_elem([[x | _xs] | xss]), do: [x | first_elem(xss)]
end
```

```
xss = [[A, B], [:a, :b, :c, :d], [1], [], [3.2, 2.1, 1.0]]
(FirstElem4.first_elem(xss) === [A, :a, 1, nil, 3.2]) |> inspect() |> IO.puts()
```

```
defmodule FirstElem5 do
 @spec first_elem(xss :: [[any()]]) :: zs :: [any()]
 def first_elem(xss) do
  xss
  |> Enum.map(fn
   [] -> nil
   [x | _xs] -> x
  end)
 end
end
```

```
xss = [[A, B], [:a, :b, :c, :d], [1], [], [3.2, 2.1, 1.0]]
(FirstElem5.first_elem(xss) === [A, :a, 1, nil, 3.2]) |> inspect() |> IO.puts()
```

```
defmodule FirstElem6 do
 @spec first_elem(xss :: [[any()]]) :: zs :: [any()]
 def first_elem(xss) do
  for xs <- xss,
     do:
      (case xs do
        [] -> nil
        [x | _xs] -> x
       end)
 end
end
```

```
xss = [[A, B], [:a, :b, :c, :d], [1], [], [3.2, 2.1, 1.0]]
(FirstElem6.first_elem(xss) === [A, :a, 1, nil, 3.2]) |> inspect() |> IO.puts()
```

Elfajult mátrix első oszlopának transzponálása

Elfajult mátrixnak nevezünk egy olyan mátrixot, amelynek soraiban különböző számú elemek lehetnek. Egy ilyen mátrixot is listák listájaként, sorfolytonosan ábrázolunk. Írjon olyan függvényt, amelyik egy elfajult mátrix első oszlopát az eredménymátrix első sorává transzformál, a további sorokban pedig a bemenő mátrix sorai vannak a transzformált első elemek kivételével, az eredeti sorrend megtartásával. Ha egy sor üres, akkor első eleme helyett a nil atom kerüljön be az eredménymátrixba. Írjon több változatot, pl. (1) saját rekurzív függvényt, (2) az Enum.map/2 és/vagy magasabb rendű függvényekkel, (3) for-komprehenzióval. Használjon mintaillesztést, ahol csak lehet!

```
defmodule FirstColTranspose do
 @spec first_col_transpose(xss :: [[any()]]) :: zss :: [[any()]]
 def first_col_transpose(xss) do
```

```
  ...
 end
end
xss = [[A,B,C],[:a,:b,:c],[1,2,3],[3.2,2.1,1.0]]
zss = [[A, :a, 1, 3.2],[B,C],[:b,:c],[2,3],[2.1,1.0]]
(FirstColTranspose.first_col_transpose(xss) === zss) |> inspect() |> IO.puts()
xss = [[A,B],[:a,:b,:c,:d],[1],[3.2,2.1,1.0]]
zss = [[A, :a, 1, 3.2],[B],[:b,:c,:d],[],[2.1,1.0]]
(FirstColTranspose.first_col_transpose(xss) === zss) |> inspect() |> IO.puts()
(FirstColTranspose.first_col_transpose([]) === [[]]) |> inspect() |> IO.puts()
xss = [[A,B],[:a,:b,:c,:d],[1],[],[3.2,2.1,1.0]]
zss = [[A,:a,1,nil,3.2],[B],[:b,:c,:d],[],[],[2.1,1.0]]
(FirstColTranspose.first_col_transpose(xss) === zss) |> inspect() |> IO.puts()
```

Lehet olyan változatot is írni, amelyik csak egyszer halad végig a bemenő mátrixon. Rajta, írjon egyet! Vagy többet! :-)

```
defmodule FirstColTranspose1 do
 @spec first_col_transpose(xss :: [[any()]]) :: zss :: [[any()]]
 def first_col_transpose(xss) do
  [
   for(
    xs <- xss,
    do:
     case xs do
      [] -> nil
      [x | _xs] -> x
     end
   )
   | for(
     xs <- xss,
     do:
      case xs do
       [] -> []
       [_x | xs] -> xs
      end
    )
  ]
 end
end

xss = [[A, B, C], [:a, :b, :c], [1, 2, 3], [3.2, 2.1, 1.0]]
zss = [[A, :a, 1, 3.2], [B, C], [:b, :c], [2, 3], [2.1, 1.0]]
(FirstColTranspose1.first_col_transpose(xss) === zss) |> inspect() |> IO.puts()
xss = [[A, B], [:a, :b, :c, :d], [1], [3.2, 2.1, 1.0]]
zss = [[A, :a, 1, 3.2], [B], [:b, :c, :d], [], [2.1, 1.0]]
(FirstColTranspose1.first_col_transpose(xss) === zss) |> inspect() |> IO.puts()
```

```
(FirstColTranspose1.first_col_transpose([]) === [[]]) |> inspect() |> IO.puts()
xss = [[A, B], [:a, :b, :c, :d], [1], [], [3.2, 2.1, 1.0]]
zss = [[A, :a, 1, nil, 3.2], [B], [:b, :c, :d], [], [], [2.1, 1.0]]
(FirstColTranspose1.first_col_transpose(xss) === zss) |> inspect() |> IO.puts()

defmodule FirstColTranspose2 do
 @spec first_col_transpose(xss :: [[any()]]) :: zss :: [[any()]]
 def first_col_transpose(xss) do
  {zs, zss} =
   xss
   |> Enum.map(fn
    [x | xs] -> {x, xs}
    [] -> {nil, []}
   end)
   |> Enum.unzip()

  [zs | zss]
 end
end

xss = [[A, B, C], [:a, :b, :c], [1, 2, 3], [3.2, 2.1, 1.0]]
zss = [[A, :a, 1, 3.2], [B, C], [:b, :c], [2, 3], [2.1, 1.0]]
(FirstColTranspose2.first_col_transpose(xss) === zss) |> inspect() |> IO.puts()
xss = [[A, B], [:a, :b, :c, :d], [1], [3.2, 2.1, 1.0]]
zss = [[A, :a, 1, 3.2], [B], [:b, :c, :d], [], [2.1, 1.0]]
(FirstColTranspose2.first_col_transpose(xss) === zss) |> inspect() |> IO.puts()
(FirstColTranspose2.first_col_transpose([]) === [[]]) |> inspect() |> IO.puts()
xss = [[A, B], [:a, :b, :c, :d], [1], [], [3.2, 2.1, 1.0]]
zss = [[A, :a, 1, nil, 3.2], [B], [:b, :c, :d], [], [], [2.1, 1.0]]
(FirstColTranspose2.first_col_transpose(xss) === zss) |> inspect() |> IO.puts()

defmodule FirstColTranspose3 do
 @spec first_col_transpose(xss :: [[any()]]) :: zss :: [[any()]]
 def first_col_transpose([]), do: [[]]

 def first_col_transpose([[x | xs] | xss]) do
  [ys | yss] = first_col_transpose(xss)
  [[x | ys] | [xs | yss]]
 end

 def first_col_transpose([[] | xss]) do
  [ys | yss] = first_col_transpose(xss)
  [[nil | ys] | [[] | yss]]
 end
end

xss = [[A, B, C], [:a, :b, :c], [1, 2, 3], [3.2, 2.1, 1.0]]
```

```
zss = [[A, :a, 1, 3.2], [B, C], [:b, :c], [2, 3], [2.1, 1.0]]
(FirstColTranspose3.first_col_transpose(xss) === zss) |> inspect() |> IO.puts()
xss = [[A, B], [:a, :b, :c, :d], [1], [3.2, 2.1, 1.0]]
zss = [[A, :a, 1, 3.2], [B], [:b, :c, :d], [], [2.1, 1.0]]
(FirstColTranspose3.first_col_transpose(xss) === zss) |> inspect() |> IO.puts()
(FirstColTranspose3.first_col_transpose([]) === [[]]) |> inspect() |> IO.puts()
xss = [[A, B], [:a, :b, :c, :d], [1], [], [3.2, 2.1, 1.0]]
zss = [[A, :a, 1, nil, 3.2], [B], [:b, :c, :d], [], [], [2.1, 1.0]]
(FirstColTranspose3.first_col_transpose(xss) === zss) |> inspect() |> IO.puts()
```

Lista adott indexű eleme Elixir-stílusú hibajelzéssel

Írjon olyan függvényt mintaillesztéssel, amelyik egy lista adott indexű elemét adja eredményül, vagy a nil atomot, ha nincs ilyen indexű elem a listában! Az indexelés 0-val kezdődik, és balról jobbra halad.

```
defmodule AtEx do
 @spec at(xs :: [any()], n :: integer()) :: r :: (x :: any()) | nil
 # Az xs lista n-edik eleme e (indexelés 0-tól, balról jobbra)
 def at(xs, n) do
 ...
 end
end
IO.puts(AtEx.at(~c"abcdefghi", 0) == ?a)
IO.puts(AtEx.at(~c"abcdefghi", 3) == ?d)
IO.puts(AtEx.at(~c"abcdefghi", 9) == nil)
IO.puts(AtEx.at(~c"", 5) == nil)
```

Ne használja a List.at függvényeket!

```
defmodule AtEx do
 @spec at(xs :: [any()], n :: integer()) :: r :: (x :: any()) | nil
 # Az xs lista n-edik eleme e (indexelés 0-tól, balról jobbra)
 def at([], _), do: nil
 def at([x | _xs], 0), do: x
 def at([_x | xs], n), do: at(xs, n - 1)
end

IO.puts(AtEx.at(~c"abcdefghi", 0) == ?a)
IO.puts(AtEx.at(~c"abcdefghi", 3) == ?d)
IO.puts(AtEx.at(~c"abcdefghi", 9) == nil)
IO.puts(AtEx.at(~c"", 5) == nil)
```

Listák utolsó elemeinek listája

Írjon olyan rekurzív függvényt a LastEr.last/1 függvény felhasználásával, amelyik paraméterként egy listák listáját kapja meg, és a belső listák utolsó eleméből álló listát adja eredményül. Ha egy belső lista üres, hagyja figyelmen kívül. Ha

az összes belső lista üres, vagy a függvényt magát egy üres listára alkalmazzuk, az eredmény az üres lista legyen.

```
defmodule LastOfListsEr do
 @spec last_of_lists(xss :: [[any()]]) :: zs :: [any()]
 # xss utolsó elemeinek listája zs
 def last_of_lists(xss) do
 ...
 end
end
IO.puts(LastOfListsEr.last_of_lists([]) == [])
IO.puts(LastOfListsEr.last_of_lists([[]]) == [])
IO.puts(LastOfListsEr.last_of_lists([[], ~c""]) == [])
IO.puts(LastOfListsEr.last_of_lists([~c"apple", ~c"peach", ~c"", ~c"orange"]) == ~c"ehe")
```

Súgó

A LastEr.last/1 függvény visszatérési értékeit mintaillesztéssel azonosítsa egy case szerkezetben a rekurzív hívásokhoz.

```
defmodule LastOfListsEr do
 @spec last_of_lists(xss :: [[any()]]) :: zs :: [any()]
 # xss utolsó elemeinek listája zs
 def last_of_lists([]), do: []

 def last_of_lists([xs | xss]) do
  case LastEr.last(xs) do
   {:ok, x} -> [x | last_of_lists(xss)]
   :error -> last_of_lists(xss)
  end
 end
end

IO.puts(LastOfListsEr.last_of_lists([]) == [])
IO.puts(LastOfListsEr.last_of_lists([[]]) == [])
IO.puts(LastOfListsEr.last_of_lists([[], ~c""]) == [])
IO.puts(LastOfListsEr.last_of_lists([~c"apple", ~c"peach", ~c"", ~c"orange"]) == ~c"ehe")
```

Írja meg a függvény egy változatát a LastEx.last/1 függvény alkalmazásával.

```
defmodule LastOfListsEx do
 @spec last_of_lists(xss :: [[any()]]) :: zs :: [any()]
 # xss utolsó elemeinek listája zs
 def last_of_lists(xss) do
 ...
 end
end
IO.puts(LastOfListsEx.last_of_lists([]) == [])
IO.puts(LastOfListsEx.last_of_lists([[]]) == [])
```

```
IO.puts(LastOfListsEx.last_of_lists([[], ~c""]) == [])
css = [~c"now", ~c"bye", ~c"", ~c"hell", ~c"cell", ~c""]
(LastOfListsEx.last_of_lists(css) == ~c"well") |> IO.puts()
```

Mire kell ügyelnie a case kifejezésben ebben a változatban, amire a LastEr.last/1 függvény alkalmazásakor nem kellett? Itt miért igen, ott miért nem?

Súgó

A minták sorrendjére a LastEx.last/1 függvény visszatérési értékeinek mintaillesztéssel való azonosításakor.

```
defmodule LastOfListsEx do
 @spec last_of_lists(xss :: [[any()]]) :: zs :: [any()]
 # xss utolsó elemeinek listája zs
 def last_of_lists([]), do: []

 def last_of_lists([xs | xss]) do
  case LastEx.last(xs) do
   nil -> last_of_lists(xss)
   x -> [x | last_of_lists(xss)]
  end
 end
end

IO.puts(LastOfListsEx.last_of_lists([]) == [])
IO.puts(LastOfListsEx.last_of_lists([[]]) == [])
IO.puts(LastOfListsEx.last_of_lists([[], ~c""]) == [])
css = [~c"now", ~c"bye", ~c"", ~c"hell", ~c"cell", ~c""]
(LastOfListsEx.last_of_lists(css) == ~c"well") |> IO.puts()
```

Listák utolsó előtti elemeinek listája

Írjon olyan rekurzív függvényt a LastEx.last/1 vagy az AtEx.at/2függvény felhasználásával, amelyik listák listáját kapja paraméterként, és a belső listák utolsó előtti eleméből álló listát adja eredményül. Ha egy belső lista üres, hagyja figyelmen kívül. Ha az összes belső lista üres, vagy a függvényt magát egy üres listára alkalmazzuk, az eredmény az üres lista legyen.

```
defmodule LastButOnesEx do
 @spec last_but_ones(xss :: [[any()]]) :: zs :: [any()]
 # xss utolsó előtti elemeinek listája zs
 def last_but_ones(xss) do
 ...
 end
end
IO.puts(LastButOnesEx.last_but_ones([]) == [])
IO.puts(LastButOnesEx.last_but_ones([[]]) == [])
```

```
IO.puts(LastButOnesEx.last_but_ones([[], ~c""]) == [])
IO.puts(LastButOnesEx.last_but_ones([~c"bye", ~c"tiles", ~c"", ~c"list"]) == ~c"yes")

defmodule LastButOnesEx do
 @spec last_but_ones(xss :: [[any()]]) :: zs :: [any()]
 # xss utolsó előtti elemeinek listája zs
 def last_but_ones([]), do: []

 def last_but_ones([xs | xss]) do
  case AtEx.at(xs, length(xs) - 2) do
   nil -> last_but_ones(xss)
   x -> [x | last_but_ones(xss)]
  end
 end
end

IO.puts(LastButOnesEx.last_but_ones([]) == [])
IO.puts(LastButOnesEx.last_but_ones([[]]) == [])
IO.puts(LastButOnesEx.last_but_ones([[], ~c""]) == [])
IO.puts(LastButOnesEx.last_but_ones([~c"bye", ~c"tiles", ~c"", ~c"list"]) == ~c"yes")
```