Tartalom

DP25a - 2. FP-gyakorlat, 2025-09-23

Bal-, jobb- és törzsrekurzió

A jobb- és a törzsrekurzióra több példát láttunk már. Most olyan egyszerű feladatok következnek, amelyek a jobb- és a balrekurzió különbségére mutatnak rá.

Mivel a megírandó függvények két vagy több klózból fognak állni, nézzük meg, mit tudunk tenni, ha a hívásra illeszkedő klózt nem lehet csupán mintaillesztéssel kiválasztani.

Ha nem megy mintaillesztéssel...

Már tudjuk, hogy mintában csak *tömör*, azaz kiértékelhető kifejezés lehet, változót tartalmazó nem. Ilyesmit tehát nem írhatunk le:

```
...
def fac(n >= 0), do: ...
```

A hasonló esetek elég gyakoriak, ezért a mintát ún. *őrrel* egészíthetjük ki.

Az őrt a függvényfejben a paraméter(eke)t követő when kulcsszó vezeti be, ami után *őrkifejezésnek* kell állnia. Az őrkifejezésre vissza fogunk térni, elöljáróban annyit, hogy csak őrként *(guard)* definiált, garantáltan *mellékhatás nélküli* könyvtári függvényeket hívhatunk meg benne, más függvényeket nem, így saját függvényeket sem.

```
...
 def fac(n) when n >= 0, do: ...
...
```

Kiírás rekurzív hívás előtt és után

Írjon lineárisan rekurzív függvényeket az alábbi feladatok megoldására direkt rekurzióval. 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. Használjon őrt a minta szerinti feltétel kiegészítésére.

```
defmodule UptoBy31 do
 @spec upto_by_3(n :: integer()) :: :ok
 def upto_by_3(n) do
  IO.puts(i)
  ...
 end
end
UptoBy31.upto_by_3(20)
```

Mint már tudjuk, jobbrekurziónak (terminális, ritkábban farokrekurziónak - angolul: tail recursion) nevezzük a rekurzív hívást, ha egy klóz utolsó és egyetlen rekurzív hívása, melynek 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 UptoBy32 do
 @spec upto_by_3(n :: integer()) :: :ok
 def upto_by_3(n) do
  ...
  IO.puts(i)
 end
end
UptoBy3.upto_by_32(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.

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

Az n-tő lefelé való haladásnak az a járulékos előnye itt és hasonló esetekben, hogy a végállomás a 0 (esetleg más előre tudható konstans) lesz, így lehet csak mintaillesztést használni, ami hatékonyabb, mintha őrt is használnánk.

Ha tehát őrt használt volna most is, cserélje le pusztán mintaillesztésre. Használjon segédfüggvényt.

Összeg testvéries elosztása

A harmadik FP-előadáson tárgyaltuk ezt a feladatot:

"Adott pénzérméket úgy kell elosztani két ember között, hogy a két összeg különbségének abszolút értéke a lehető legkisebb legyen (CEOI'1995 versenyfeladat).

Ha az érmék értékét tartalmazó lista ez: [28, 7, 11, 8, 9, 7, 27], akkor egyikük a [9, 11, 28] érméket kapja, melyek összege 48, másikuk a többit, melyek összege 49."

A bemutatott megoldás először kigyűjti az összes olyan részlistát, melyek összege nem nagyobb a teljes listaösszeg felénél, majd ezekből válogatja ki a maximális összegűeket.

Írjon egy vagy több olyan változatot, amelyek már menet közben eldobják az aktuális maximumnál kisebb részlistákat. Ezután próbáljon olyan megoldást írni, amely az egyszer már kiszámolt összegű részlisták összegét nem számolja ki újra. Hasonlítsa össze az egyes változatok futási idejét a benchee segítségével, és becsülje meg a tárigényüket.

Műveletek listákon

Írjon többféle megoldást a feladatokra: saját rekurzív függvényekkel, különféle könyvtári függvények felhasználásával, for-komprehenzióval.

Hasonlítsa össze a megoldások futási idejét a benchee-vel, próbáljon hatékonyabb kódot írni pl. jobbrekurzióval.

Használja a kino_explorert-t nyomkövetésre, hibakeresésre.

Próbálja ki a dialyxir modult, azaz a *dialyzer* segédeszközt, ellenőrizze programjaiban a típus- és függvényspecifikációk helyességét.

A *dialyzert* nem tudja *Livebook* cellában futtatni, mert a *dialyzer* a lefordított BEAM-kódot elemzi, a *Livebook* pedig nem menti el a BEAM-kódot a háttértárba. Ezért hozzon létre egy *mix* projektet, másolja ki az elemzendő programrészeket a *Livebook* cellá(k)ból, és mentse el ezt a kódot a *mix* projekt fájlrendszerén belül a lib mappában egy .ex típusnevű fájlba. Írja be a *mix* függőségei közé a dialyxir-t, töltse le és fordítsa le az új modulokat. Ezután már futtathatja a *dialyzert*. A harmadik FP-előadáson néhány dián össze van foglalva a dialyxir telepítése és használata. Itt is olvashat róla: https://hexdocs.pm/dialyxir/readme.html.

A *mix* parancssoros használatához az *elixirt* telepítenie kell a saját gépére, vagy *dockerből* kell tudni futtatnia. Az első FP-előadás diái segítenek az *elixir* telepítésben, az *elixir* és a *mix* használatban, ha eddig még nem használta parancssorból az *elixirt* vagy a *mixet*.

L1. Lista kettévágása

Írjon függvényt egy lista kettévágására! Írhat segédfüggvényt, használhat akkumulátort és jobbrekurziót, használhatja a for-jelölést.

Ne használja az Enum.split, *Enum.take* és Enum.drop* függvények semelyik változatát 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))
```

L2. Lista adott feltételt kielégítő elemeiből álló prefixuma

Írjon függvényt egy lista adott feltételt kielégítő prefixumának előállítására. Írhat segédfüggvényt, használhat akkumulátort és jobbrekurziót, használhatja a for-jelölést.

Ne használja az Enum.split, *Enum.take* és Enum.drop* függvények semelyik változatát 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")
```

L3. Lista minden n-edik elemének kihagyásával létrejövő lista

Írjon függvényt egy olyan lista létrehozására, amelyikből a paraméterként átadott lista minden n-edik eleme, a nulladiktól kezdve, ki van hagyva. (A listák indexelése, mint tudjuk, a 0-val kezdődik.)

Ne használja az Enum.split, *Enum.take* és Enum.drop* függvények semelyik változatát 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

Ha nem ír segédfüggvényt és nincs más ötlete, használhatja a for-jelölést, generátoraként a ../2, szűrőjeként a rem/2 függvényeket, a listaelemek elérésére pedig a Enum.at/2 függvényt.

L4. 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.

L5. 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
```

L6. 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. Írhat segédfüggvényt és akkumulátort nem használó, valamint akkumulátoros segédfüggvényt használó változatot. Próbáljon meg egyéb változatokat is írni, pl. a for-jelöléssel és 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([]) === [])
```