Tartalom

DP24a - 3. FP gyakorlat, 2024-09-23

Műveletek listákon

Ahol csak lehet, javasoljuk a for-jelölés használatát.

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))
```

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")
```

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.

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.

Lista egymást követő két-két eleméből képzett párok listája

Írjon olyan rekurzív függvényt, amelyik egy lista 0. és 1., 2. és 3., 4. és 5. 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()]
 # Az xs lista egymás után álló elemeiből képzett párok listája zs
 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
```

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([]) === [])
```

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. Írhat segédfüggvényt és akkumulátort nem használó, valamint akkumulátoros segédfüggvényt használó változatot. Feltétlenül írjon egyéb változatot is for-jelöléssel (meg fog lepődni!).

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

Súgó

Ha a for-jelölés 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.

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.

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 Repeated.repeated/1 függvényt is.

Természetes szám valódi osztói

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

Írjon olyan 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
def proper_divisors(i), do: for ...
(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.

Összetett számok

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

Írjon olyan 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
def composite_numbers(i), do: for ...
(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)
```

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 = fn(n) -> for i <- 4..n, composite?.(i), do: i end
(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
def probes(i), do: ...
@spec composite?(i :: integer()) :: b :: boolean()
# b igaz, ha i összetett szám
def composite?(i), do: ...
@spec composite_numbers_faster(i :: integer()) :: ns :: [integer()]
# Az i-nél nem nagyobb összetett számok listája ns
def composite_numbers_faster(i), do: ...
```

A megoldást, ha a javaslatunkat megfogadta, három kis – esetleg több – lépésre bontotta, mindegyikre egy-egy 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.