Tartalom

DP25a - 1. FP-gyakorlat, 2025-09-09

Jelölések, konvenciók

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.

Első lépések

Válassza ki, hogy a *Livebookot* (esetleg külön az *Elixirt* is) hogyan kívánja telepíteni a saját gépére: Docker konténerként, a Mac, ill. a Windows telepítőkkel, Debian/Ubuntu esetén az *apt*, a *mise* vagy az *asdf* csomagkezelővel. Az első előadás

prezentációs anyagában néhány dián összefoglaltuk a lehetőséget.

Kezdje a munkát a kiválasztott eszközök telepítésével, az ismerkedéssel. Hozzon létre Elixir-cellát a *Livebookban*, inditsa el egy terminálablakban az *iex*-et, írjon be néhány egyszerű kifejezést, hívjon meg közismert függvényeket a Kernel Elixir- vagy a :math Erlang-könyvtárból, figyelje meg a *szövegkiegészítés* (completion) és a *Livebookban* a *felugró súgó* működését. Ha elakad, kérjen segítséget a gyakorlatvezetőktől vagy társaitól.

Ez a fájl az első tantermi gyakorlat anyagát tartalmazza. Ha a feladatsort nem sikerül befejeznie a gyakorlaton, oldja meg otthon. Lehetőség van távkonzultációra: üzenjen nekünk a Teams csoportban!

Rekurzió

A deklaratív programozás alappillére a rekurzió. A rekurzió kétféle lehet: lineáris és elágazó (angolul linear recursion és 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.

Egyszerű feladatok listák feldolgozására

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.

Egy rekurzív adatszerkezet feldolgozására legalább két, esetleg több klózt írunk. Közöttük vannak 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.

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

Gyakran előfordul, hogy bizonyos listákon bizonyos műveleteket nem lehet elvégezni. Pl. egy üres listának egyetlen eleme sincs, bármelyik elemét is kérjük, nincs mit eredményül adni.

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 a visszatérési érték az :error atom;
- Elixir-stílusban a visszatérési érték típusa any() | nil, azaz siker esetén ugyancsak egy any() típusú value a visszatérési érték, meghiúsulás esetén pedig a nil atom.

Az alábbi feladatok megoldására **ne** használja az azonos vagy hasonló feladatokat megoldó könyvtári függvényeket a Kernel, List, Enum és más modulokból, pl. hd, tl, first, last, at, length, split, slice!

Azt kérjük, hogy most gyakorlásképpen *saját* – ahol kell, rekurzív – függvénydefiníciókat írjon.

1. Lista feje

Írjon függvényt egy lista fejének (első elemének) visszaadására! Ha a lista üres, Elixir-stílusban jelezze, azaz a nil atomot adja eredményül.

Súgó

Az első klóz a legalább egy elemű listára, a második pedig az üres listára illeszkedjen.

```
defmodule Head do
 @spec hd(xs :: [any()]) :: r :: (x :: any()) | nil
 # Ha xs nem üres, r == x, az xs lista feje, egyébként r == nil
 def hd(xs), do:
 ...
end
IO.puts(Head.hd([]) == nil)
IO.puts(Head.hd(Range.to_list(1..5)) == 1)
IO.puts(Head.hd(~c"almárium") == ?a)
```

2. Lista farka

Írjon függvényt egy lista farkának (az első elemét követő részlistájának) a visszaadására! Ha a lista üres, Elixir-stílusban jelezze, azaz a nil atomot adja eredményül.

Súgó

Az első klóz a legalább egy elemű listára, a második pedig az üres listára illeszkedjen.

```
defmodule Tail do
 @spec tl(xs :: [any()]) :: r :: (ts :: any()) | nil
 # Ha xs nem üres, r == ts, az xs lista farka, egyébként r == nil
 @spec tl(xs: [any()]) :: ts :: [any()] | nil
 # Az xs lista farka ts
 def tl(xs), do:
 ...
end
IO.puts(Tail.tl([]) == nil)
IO.puts(Tail.tl(Range.to_list(1..5)) == [2,3,4,5])
IO.puts(Tail.tl(~c"almárium") == ~c"lmárium")
```

3. Lista n-edik eleme

Írjon rekurzív függvényt egy lista n-edik elemének a visszaadására! (A lista indexelése 0-tól indul.) Ha a lista n-nél rövidebb, Elixir-stílusban jelezze, azaz a nil atomot adja eredményül.

Súgó

Itt három klózt kell írnia: egyet az üres listára, egyet a megtalált elem visszaadására, egyet pedig arra, hogy rekurzív hívással folytassa a keresést.

```
defmodule Nth do
 @spec nth(xs :: [any()], n :: integer()) :: r :: (x :: any()) | nil
 # Ha xs elég hosszú, r == x, az xs n-edik eleme; egyébként r == nil (indexelés 0-tól)
 def nth(xs, n), do:
 ...
end
IO.puts(Nth.nth([], 5) == nil)
IO.puts(Nth.nth(Range.to_list(1..5), 4) == 5)
IO.puts(Nth.nth(Range.to_list(1..5), 5) == nil)
IO.puts(Nth.nth(~c"almárium", 3) == ?á)
IO.puts(Nth.nth(~c"almárium", -3) == nil)
```

A benchee Elixir-modul és a mix segítségével hasonlítsa össze a futási időket:

- egy hosszú lista első, középső és utolsó elemének elérése esetén,
- olyan klózsorrendek esetén, amikor az első, illetve az utolsó klóz illeszkedik az üres listára.

4. Lista hossza

Írjon rekurzív függvényt egy lista hosszának meghatározására! Ne használjon segédfüggvényt!

Súgó

Az első klóz a legalább egy elemű listákra, a második pedig az üres listára illeszkedjen.

```
defmodule Length do
 @spec len(xs :: [any()]) :: n :: integer()
 # Az xs lista hossza n
```

```
  def len(xs), do:
  ...
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 hosszának megállapítására most akkumulátort és segédfüggvényt használó *jobbrekurzív* függvényt írjon!

Súgó

Az akkumulátorban gyűjtse a listaelemek számát: amikor leszedi a soron következő elemet a lista elejéről, akkor a rekurzív hívásban az akkumulátor korábbi értékéhez adjon 1-et.

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

  @spec len(xs :: [any()], count :: integer()) :: n :: integer()
  # Az xs lista hossza és count összege n
  def len(xs, count), do:
    ...
end

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

5. Lista utolsó eleme

Írjon rekurzív függvényt egy lista utolsó elemének visszaadására! Ne használjon segédfüggvényt!

Esetek megkülönböztetése kölcsönös kizárással A 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* (angolul: *layered pattern):*

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

A lista farkára az xxs változóval lehet hivatkozni. Ha az x2 és xs változókat *nem használjuk* a klóz törzsében, akkor erre az Elixir fordító figyelmeztetni fog. A figyelmeztetést úgy kerülhetjük el, hogy a változó nevét aláhúzásjellel (_) kezdjük.

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

Az aláhúzásjellel kezdődő nevek helyett elég aláhúzásjelet írni:

def fun([x1 | xxs = [_|_]) ...
def fun([x]) ...,

a beszédes nevek azonban segítik a megértést, utalnak a változó szerepére.

Ha a lista második elemére vagy a harmadik elemtől kezdődő farkára akarunk hivatkozni a klóz törzsében, akkor inkább 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
IO.puts(Last.last(~c"Élvezed?") == ??)
IO.puts(Last.last([]))
```

Most írja meg a függvényt 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
IO.puts(LastEx.last(~c"Élvezed?") == ??)
IO.puts(LastEx.last([]) == nil)
```

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

6. Lista k-adik elemétől induló, n hosszú részlistája

Írjon rekurzív függvényt egy lista olyan n hosszú részlistájának a visszaadására, amelyik a k-adik elemtől kezdődik (a lista indexelése 0-tól indul)! Ha nincs ilyen hosszú részlistája, Elixir-típusú hibajelzéssel térjen vissza. Ne használjon segédfüggvényt! A klózok sorrendjének megválasztásával törekedjék hatékony megoldásra.

Gondolja át, hányféle esetet kell megkülönböztetnie!

Súgó

1. Kiszedtük a lista k-adik elemével kezdődő, n hosszú részlistát.
2. Elhagytuk a lista első $k-1$ elemét, kigyűjthetjük a következő n darab elemet.
3. Elhagyjuk a lista első $k-1$ elemét.
4. Elfogytak az elemek a listából, mielőtt kiszedtük volna az n hosszú részlistát.

```
defmodule Slice do
 @spec slice(xs :: [any()], k :: integer(), n :: integer()) :: r :: (ss :: [any()]) | nil
 # Ha xs elég hosszú, r == ss, az xs k-tól induló, n hosszú részlistája; egyébként r == nil
 # Indexelés 0-tól
 def slice(xs, k, n), do:
 ...
end
IO.puts(Slice.slice([], 0, 5) == nil)
IO.puts(Slice.slice(Range.to_list(1..5), 1, 3) == [2,3,4])
IO.puts(Slice.slice(Range.to_list(1..5), 4, 1) == [5])
IO.puts(Slice.slice(~c"almárium", 3, 3) == ~c"ári")
IO.puts(Slice.slice(~c"almárium", -3, 3) == nil)
```

A benchee Elixir-modul és a mix segítségével mérje meg a futási időket különféle paraméterezések mellett, továbbá profilírozással nézze meg, hogy melyik klóz, illetve hívott függvény használja el a legtöbb futási időt. Hasonlítsa össze a saját Slice.slice/3 függvényének futási idejét az Enum.slice/3 függvény futási idejével különféle szélsőséges paraméterezések mellett.

7. Tagsági vizsgálat

Írjon rekurzív függvényt annak eldöntésére, hogy egy érték benne van-e egy listában. Ne használjon segédfüggényt, ügyeljen a hatékonyságra.

```
defmodule Member do
  @spec member?(xs :: [any()], e :: any()) :: b :: boolean()
  # b == true, ha e benne van xs-ben, egyénként false
  def member?(xs, e), do:
  ...
end
(Member.member?(~c"A szó elszáll", ?ó) == true) |> IO.inspect()
(Member.member?([~c"A szó", ~c"elszáll", ~c"az írás", ~c"megmarad."], ~c"elszáll")
  == true) |> IO.inspect()
(Member.member?([1.2, ?v, "str", false], false) == true) |> IO.inspect()
(Member.member?([1.2, ?v, "str", false], "str") == true) |> IO.inspect()
(Member.member?([1.2, ?v, "str", false], ~c"str") == false) |> IO.inspect()
(Member.member?([], []) == false) |> IO.inspect()
```

8. Prímvizsgálat

Írjon rekurzív programot annak eldöntésére, hogy egy egész szám prím-e. Feltételezheti, hogy a függvényt egészszám-paraméterrel hívjuk. (Az első előadás egyik Prolog-példája volt ez a feladat.) Segédfüggvényt használhat. Ügyeljen a hatékonyságra.

```
defmodule Prime do
  @spec prime?(x :: integer()) :: b :: boolean()
  # b == true, ha x prím
  def prime?(x), do:
  ...
end
(Prime.prime?(17) == true) |> IO.inspect()
(Prime.prime?(18) == false) |> IO.inspect()
(Prime.prime?(197_628) == false) |> IO.inspect()
(Prime.prime?(1_000_003) == true) |> IO.inspect()
(Prime.prime?(1_213_457) == false) |> IO.inspect()
(Prime.prime?(179_424_691) == true) |> IO.inspect()
```

További gyakorló feladatok

```
defmodule Fp1Gy do
  @spec revapp(xs :: [integer()], ys :: [integer()]) :: zs :: [integer()]
  # zs == xs fordítottja ys elé fűzve
  def revapp(xs, ys) do
    ...
  end

  @spec rev(xs :: [integer()]) :: zs :: [integer()]
  # zs == xs fordítottja
  def rev(xs) do
    ...
  end

  @spec diff(xs :: [integer()], ys :: [integer()]) :: zs :: [integer()]
  # zs == xs és ys különbsége, azaz xs azon elemei, melyek nincsenek benne ys-ben
  # a listában az azonos értékű elemeket külön elemeknek tekintjük,
  # azaz az ilyen lista zsák (bag), nem halmaz (set)
  def diff(xs, ys) do
    ...
```

```
  end
end
```