

Tartalom

dp25a FP gy1 felvezető 2025-09-09

Kedvcsináló első lépések	1
Két láncolt lista összefűzése rekurzív függvénnyel (app/2)	2
Röviden a mintaillesztésről	3
Vissza a listák összefűzéséhez!	4
Faktoriális (jobbrekurzív is)	7
Egészlista elemeinek összege	8

dp25a FP gy1 felvezető 2025-09-09

Kedvcsináló első lépések

Talán a legalapvetőbb adatstruktúra a deklaratív nyelvekben a *láncolt lista*.



Az Elixir, az Erlang és a Prolog közös szintaxisa a láncolt listák jelölésére a következő:

- `[]` – üres lista,
- `[x|xs]` – olyan lista, amelynek `x` a *feje* (első eleme) és `xs` a *farka* (a fejet nem tartalmazó részlistája).

Az 1,2,3 egész számokból álló (láncolt) listát többféle módon is megadhatjuk, a tömörebb változatok nyilván könnyebben írhatók le és olvashatók:

`[1|2|3|[]]`, `[1,2|3]`, `[1,2,3]` stb.

`[1|2|3|[]]` `|> IO.inspect()`

`[1,2|3]` `#|> IO.inspect()`

`[1,2,3]`

Tipp: Húzza a kurzort az `IO`, illetve az `inspect` szavak fölé, ill. később más modul- és függvénynevek fölé, hogy többet tudjon meg róluk! Ha nem jelenne meg semmi, futtassa a kódot, azaz kattintson az Evaluate gombra.

A fenti kis példában a `'|>'` az ún. *pipe* operátor (mint a Linuxban a `|`): a bal oldalán álló kifejezés eredményét adja át a jobb oldalán álló függvénynek, mégpedig első paramétereként, ha több paramétere is lenne.

Mire való az `IO.inspect/1` függvény? Mivel az Elixir egy kifejezéssorozat kiértékelésekor csak az utolsó kifejezés értékét írja ki a terminálra, az `IO.inspect/1`-gyel vesszük rá arra, hogy a korábbi kifejezések értékét is kiírja.

Az `IO.inspect/1` – a többi `inspect` függvénnyel együtt – nagyon hasznos hibakeresési segédeszköz is, mert nemcsak kiírja a kapott kifejezés értékét, hanem eredményként vissza is adja.

```
[1 |> IO.inspect(), 2 |> IO.inspect(), 3 |> IO.inspect()] |> IO.inspect()
```

Az Elixirben egy függvényt három dolog azonosít: a neve, az *aritása* (azaz a paramétereinek száma), valamint annak a modulnak a neve, amelyikben az adott függvény definiálva van.

Például `String.slice/2` és a `String.slice/3` a `String` modul két azonos nevű függvényét azonosítja, az egyiknek két, a másiknak három paramétere van.

Amikor ugyanabban a modulban hivatkozunk egy függvényre, mint amelyikben definiálva van, akkor a modulnév elmaradhat.

Függvényt csak modulban lehet definiálni. Az Elixir értelmező programban (`iex`) modul nem definiálható, a *Livebookban* azonban igen.

```
def dummy, do: "halihó"
dummy()
```

```
defmodule Dummy do
```

```
  def dummy0, do: 2024          # exportált, paraméter nélkül
```

```
  def dummy1(p), do: dummy2(p)  # exportált fv.
```

```
  defp dummy2(p), do: IO.puts(p) # privát fv.
```

```
end
```

```
Dummy.dummy0
```

```
Dummy.dummy0 |> IO.inspect()
```

```
Dummy.dummy1('karakterlánc')  # karakterkódokból álló láncolt lista, nem sztring!
```

```
Dummy.dummy1(~c"karakterlista") # a ~c egy ún. szigil, speciális jelölés az Elixirben
```

```
Dummy.dummy1("sztring")
```

```
# Dummy.dummy2("sztring")
```

Két láncolt lista összefűzése rekurzív függvénnyel (`app/2`)

A deklaratív stílus érzékeltetésére *két lista összefűzését* mutatjuk be: egy Elixir-függvényt írunk `app` néven. Az `app/2` függvény kapcsán sok részletet mutatunk be az Elixir nyelvről, a rekurzióról, a rekurzió hatékonyságáról, a *Livebook* használatáról stb. Számos részletre később visszatérünk.

Az `xs` és `ys` listák összefűzése azt jelenti, hogy az `xs` lista összes elemét az `ys` *elé* fűzzük az elemek eredeti sorrendjének megőrzésével: `xs @ ys`.

Mivel a lista láncolt (ún. lineárisan rekurzív) adatstruktúra, csak a lista *első* elemét érjük el közvetlenül, a lista utolsó – és bármely közbülső – elemét csak úgy, ha előbb az összes előtte álló elemet eltávolítjuk.

Egyszerű a dolgunk, ha az első lista – a továbbiakban `xs` – üres, ugyanis ekkor a második listát – a továbbiakban `ys` – kell változtatás nélkül visszaadnunk.

```
defmodule App0 do
  # app(xs, ys): xs és ys listák összefűzöttje zs == (xs @ ys)
  # [] @ ys == ys
  def app([], ys), do: ys
end
```

```
App0.app([], ~c"abc")
```

MI történik, ha az `App0.app/2` függvénynek első paraméterként nem üres listát adunk át?

```
App0.app([1,2,3], ~c"abc")
```

Az Elixir – más funkcionális, ill. deklaratív nyelvekhez hasonlóan – *mintaillesztést* használ a különféle esetek felismerésére, szétválasztására. A függvények formális paraméterei tehát olyan *minták*, amelyekre az aktuális

paramétereknek illeszkedniük kell. Ha nem fedünk le minden esetet, azt az Elixir *dialyzer* eszköze az elemzés során jelzi. (A *dialyzer*rel később ismerkedünk meg.)

Röviden a mintaillesztésről



- A konstansok csak velük azonos értékre illeszkednek, pl. `123`, `[]`, `[1,2,3]`, `~c"abc"`, `"abc"`, `:eof`, `nil`, `true`.
- A kötött, azaz értékkel rendelkező változók csak velük azonos értékre illeszkednek.
- A kötetlen, azaz értékkel nem rendelkező változók tetszőleges értékre illeszkednek.
 - Pl. a `[x | xs]` minta a legalább egyelemű listákra illeszkedik: `x` felveszi a lista fejének, `xs` pedig a farkának az értékét; az utóbbi lehet üres is.
 - Pl. a `[x1, x2 | xs]` minta a legalább kételemű listára illeszkedik: `x1` felveszi a lista balról az első, `x2` balról a második elemének értékét, `xs` pedig a farkának értékét; az utóbbi lehet üres is.
 - Pl. a `[_ | xs]` minta is a legalább egyelemű listákra illeszkedik, de a lista fejét a hivatkozhatatlan `_`-hoz, a *névtelen változóhoz* köti, azaz eldobja, a farkát pedig az `xs`-hez köti.
 - Pl. a `[_, _ | _]` minta is a legalább kételemű listákra illeszkedik, de e listaelemek értékével nem kezd semmit.
- Kifejezés **nem lehet** minta, pl. `1+x`, `1+2`, `round(x)`, `round(5.3)`.
- Változóhoz értéket kétféle módon köthetünk:
 - az `=` mintaillesztés vagy kötés operátorral (nem azonos az értékadással!),
 - paraméterátadással függvényhívás esetén.

```
123 = 123
```

```
123 = 321
```

```
~c"abc" = [97, 98, 99]
```

```
:eof = :eof
```

```
nil = nil
```

```
[x | xs] = ~c"abc" |> IO.inspect  
{x, xs}
```

A `{...}` az ún. ennes (tuple) jelölése az Elixirben. Az ennes elemei vesszővel vannak elválasztva. Ennesként tudunk két vagy több értéket paraméterként átadni, eredményként visszakapni.

```
[_ | xs] = ~c"abc" |> IO.inspect  
xs
```

```
[_, _ | xs] = [0 | ~c"abc"] |> IO.inspect  
xs
```

Mi történt itt, miért lett `[0, 97, 98, 99]` a `[0 | ~c"abc"]` kifejezésből?

Ha egy egész számokból álló listában csupa megjeleníthető, azaz vezérlő vagy nyomtatható ASCII-kódú karakterként értelmezhető szám van (7..13, 32..126, 127), akkor az ilyen listát az Elixir a `~c"..."` karakterlista-jelöléssel írja ki, ellenkező esetben a karakterkódokat tartalmazó listaként.

```
[_xs] = ~c"Ábc" |> IO.inspect  
xs
```

```
defmodule BadPattern do  
  def fun(x < 0), do: x  
end
```

```
defmodule BadPattern do  
  def fun(round(x)), do: x  
end
```

```
1+2 = 1+2
```

Vissza a listák összefűzéséhez!



Ha tehát `xs`, azaz az első lista üres, akkor a már bemutatott megoldás működik. A következő cellában megismételjük a definíciót, azonban a korábban használt rövidített függvénydefiníció (`..., do: ...`) helyett a teljes függvénydefinícióval (`... do ... end`).

Ha ezt a cellát kiértékeljük az azonos nevű modult tartalmazó cella kiértékelése után, akkor az Elixir hibát jelez.

```
defmodule App0 do  
  # app(xs, ys): xs és ys listák összefűzöttje zs == (xs @ ys)  
  # [] @ ys == ys  
  def app([], ys) do  
    ys  
  end  
end
```

A *Livebook*-ban az Elixir-cellák különálló modulok, így ha egy modulnak nevet adunk az egyik cellában, akkor ugyanazt a nevet már nem használhatjuk egy másikban. Ennek az az előnye, hogy a korábbi cellákban definiált függvényeket a modulnév megadásával hívhatjuk a későbbi cellákból.

Ha az `xs` nem üres, akkor ahhoz, hogy az `ys` elé fűzzük, rendre le kell emelni és félre kell rakni az elemeit, amíg csak üressé nem válik.

Hová tegyük ezeket az elemeket? Ha *rekurzív módon* hívja meg a függvény saját magát, akkor azok átmenetileg automatikusan a hívási verembe kerülnek.

Amikor rekurzív függvényt írunk, abból indulunk ki, hogy a függvény valamilyen egyszerűbb adatstruktúrára – pl. egy paraméterként kapott lista farkára – elvégzi, amit elvárunk tőle, és ezután már csak a lista fejével kell valamit kezdenie, pl. a rekurzív hívás eredményeként kapott lista elé fűznie.

A mintaillesztés lehetőséget ad arra, hogy az előforduló eseteket világosan elkülönítsük egymástól a kódban. Két lista összefűzése esetén alapvetően két esetet érdemes megkülönböztetünk (persze lehetne többet is, de felesleges lenne) az első paraméter, `xs` értéke szerint:

1. `xs` nem üres,
2. `xs` üres.

A függvény definálásakor minden előforduló esetre egy-egy ún. *klózt* írunk. Minden klóz független a többitől: a paramétereik neve lehet azonos, de lehet különböző is; egy klóz törzsében csak a saját paramétereire lehet hivatkozni.

```
defmodule App1 do  
  # app(xs, ys): xs és ys listák összefűzöttje zs == (xs @ ys)  
  
  # [z | zs] @ ys = [z | zs @ ys]  
  def app([z | zs] = zzs, ys) do  
    [z | app(zs, ys)]  
  end
```

```
# [] @ ys == ys
def app([], ys) do
  ys
end
```

end

A $[z | zs] = zzs$ mintát réteges mintánaki (layered pattern) nevezzük, mert lehetővé teszi, hogy a függvény, pontosabban a klóz törzsében a teljes listára zss-sel, a fejére z-vel, a farkára zs-sel hivatkozzunk.

Most csak a magyarázat kedvéért alkalmaztuk a réteges mintát, hogy utalni tudjunk a teljes paraméterre is, ne csak a komponenseire.

```
defmodule App2 do
  # app(xs, ys): xs és ys listák összefűzöttje zs == (xs @ ys)
  def app([z | zs] = _zss, ys) do
    [z | app(zs, ys)]
  end

  def app([], ys) do
    ys
  end
end
```

Ha az `app/2` függvény hívásakor az első paraméter egy üres lista, akkor az Elixir a második klózt értékeli ki, ha pedig nem üres az a lista, akkor az első – a mintaillesztés működik!

Az első klóz törzsében, a rekurzív hívásban tehát, ahogy mondtuk, az `app/2` függvényt a paraméterként kapott `_zss` lista farkára, azaz `zs`-re alkalmazzuk, feltételezve, hogy képes a `zs` listát az `ys` elé fűzni. Amikor ebből a hívásból visszatér, akkor már csak a – verembe félretett – `z`-t kell a rekurzív hívás eredményeként kapott lista elé fűznie (`[z | app(zs, ys)]`).

Két dolgot kell még belátnunk.

A egyik az, hogy a rekurzió nem végtelen. Ez biztosan teljesül a jelen esetben, mert a listák véges hosszúságúak az Elixirben, a rekurzív hívás pedig a kapott `_zss`-nél mindig eggyel rövidebb `zs`-re alkalmazza `app/2`-t. A `_zss` lista tehát egyre rövidül, és amikor üressé válik, akkor a második klóz kiértékelésére kerül sor: ennek törzsében nincs rekurzív hívás, a függvény a rekurzív hívások során változás nélkül továbbadott `ys`-sel tér vissza.

A másik, amit be kell látnunk, az, hogy a függvény azt csinálja, amit elvárunk tőle. Az egyértelmű, hogy az első klózban, amikor a `_zss` már üressé vált, a rekurzív hívás az `ys` listával tér vissza, ez elé fűzi `_zss` utolsó elemét, amit a veremből vesz elő. A következő visszatéréskor már az így kibővített lista elé fűzi `_zss` utolsó előtti elemét, majd a `_zss` hátulról harmadik elemét, s.í.t., amíg csak ki nem ürül a verem. A végeredmény valóban az lesz, amit várunk.

```
App2.app([1,2,3], [4,5])
```

Az `app/2` függvényt tömörebben is definiálhatjuk, rövidített függvényjelöléssel. Ezt a formát akkor célszerű alkalmazni, amikor a klóz törzse egyetlen kifejezésből áll.

A függvények definiálásakor célszerű a paramétereknek és magának a függvénynek a típusát is specifikálni. Ez egyrészt javítja a függvény dokumentáltságát és ezáltal az olvashatóságát, másrészt lehetővé teszi, hogy a dyalizer segédprogrammal ellenőrizzük a függvény típushelyességét.

A `#`-tel kezdődő *deklaratív* fejkomment szintén a program olvashatóságát javítja: a fejkommenttel a bemenő paraméter(ek) és a függvény visszatérési értéke közötti kapcsolatot fejezzük ki deklaratív módon, azaz (lehetőleg) a *mi*-re, és ne a *hogyan*-ra adjunk választ.

```
defmodule App3 do
  @spec app(xs :: [integer()], ys :: [integer()]) :: zs :: [integer()]
  # xs és ys listák összefűzöttje zs
  def app([x|xs], ys), do: [x|app(xs, ys)]
  def app([], ys), do: ys
end
```

```
App3.app([], []) # E kifejezés értékével nem kezdünk semmit, nem is jelenik meg sehol.
```

```
App3.app([], [1, 2, 3])
```

```
IO.inspect(App3.app([], [])) # Itt sem kezdünk vele semmit, de kiírjuk a már látott
```

```
IO.inspect(App3.app([], [1, 2, 3])) # IO.inspect/1 függvénnyel, direkt paraméterátadással.
```


Mielőtt további példákat néznénk meg az App3.app/2 használatára, vizsgáljuk meg, hogy lefedtünk-e minden lehetséges esetet a függvénydefinícióban.

Az első paraméter kétféle mintára illeszkedhet a két klózban: [] vagy [x|xs] – üres és legalább egyelemű – ami valóban lefedi az első paraméter összes lehetséges értékét.

A második paraméter csak egyféle mintára illeszkedhet mindkét klózban, az ys változóra. Mivel egy kötetlen változó mindenre illeszkedik, a második paraméter összes lehetséges értékét is lefedtük.

Vagyis a függvény két klózával valóban lefedtünk minden lehetséges esetet.

App3.app/2 második klóza illeszkedik az üres listára. A rekurzió során erre a klózra csak egyszer kerül sor, ezért ez észszerű döntésnek tűnik. Vajon van-e észrevehető különbség a futási időben, ha a két klóz sorrendjét megcseréljük (lásd App4.app/2)?

```
defmodule App4 do
  @spec app(xs :: [integer()], ys :: [integer()]) :: zs :: [integer()]
  # xs és ys listák összefűzöttje zs
  def app([], ys), do: ys
  def app([x|xs], ys), do: [x|app(xs, ys)]
end
```

```
App4.app(Range.to_list(1..25_000_000), [])
:ok
```

```
App3.app(Range.to_list(1..25_000_000), [])
:ok
```

Térjünk vissza pár percre az App3.app/2 függvény alkalmazásához. Most már nem a függvény működésével, hanem az eredmények megjelenítésével fogunk foglalkozni.

```
IO.inspect(App3.app([], []))
IO.inspect(App3.app([5, 6, 7], [1, 2, 3]))
IO.inspect(App3.app([7, 10, 12], [97, 98, 99])) # Ha kiírható a karakterkód, akkor úgy is látjuk

IO.inspect(App3.app([], []))
IO.inspect(App3.app([5, 6, 7], [1, 2, 3]))
IO.inspect(App3.app([7, 10, 12], [97, 98, 99]), charlists: :as_lists) # Ha számként szeretnénk látni

App3.app([7, 10, 12], [97, 98, 99]) |> IO.inspect(charlists: :as_lists)

App3.app([7, 10, 12], [97, 98, 99]) |> inspect(charlists: :as_lists)

App3.app([7, 10, 12], [97, 98, 99]) |> inspect(charlists: :as_charlists)

App3.app([7, 10, 12], [97, 98, 99]) |> inspect(charlists: :as_charlists) |> IO.puts()
```

Az előző négy Elixir-cellában lévő kódok és így az eredmények megjelenítése kismértékben különböznek egymástól.

A négy cella közül az elsőben az IO.inspect/2, a többiben a Kernel modulban definiált inspect/2 függvényt alkalmazzuk. (A Kernel modul neve elhagyható.) A negyedik cellában az inspect/2 eredményét még az IO.puts/1 függvényen is átengedjük.

Mi a különbség az IO.inspect/2 és a Kernel.inspect/2 között? Az első a kapott kifejezést kiírja, értékét változatlan formában továbbadja. A második is kiírja a kapott kifejezést, de az értékét sztringgé konvertálva adja eredményül.

Az inspect függvények működését többféle opcióval lehet befolyásolni: az itt használt charlists: :as_charlists opció a lista elemeit karakterként, a charlists: :as_lists opció pedig számként jeleníti meg.

Az IO.puts/1 a kapott kifejezést sztringgé alakítva írja ki, eredményül az :ok atomot adja vissza.

A 0..127 tartományba eső ASCII-kódú karakterek megjelenési formáját pl. így nézhetjük meg. Láthatjuk, hogy 7..13, 27..27 és 32..126 tartományon kívül eső kódú karakterek a hexadecimális kódjukkal jelennek meg.

```
(for code <- 0..127, do: code) |> IO.inspect(charlists: :as_charlists)
```

```
[127] |> IO.inspect(charlists: :as_charlists)
```

Sok dologról esett szó már eddig is: modulok és függvények definiálásáról, rekurzióról, mintaillesztésről, különféle típusú adatokról, a |>, .. és más operátorokról, a for kompehenzióról stb. A következő hetekben alaposabban megismerkedünk ezekkel a nyelvi konstrukciókkal, fogalmakkal.

Faktoriális (jobbrekurzív is)

A rekurzió szokásos iskolapéldája az $n!$ kiszámítása. Matematikai definíciója:

$$0! = 1$$

$$n! = n \cdot (n - 1)!, \text{ ha } n > 0$$

Az első változat a matematikai definíciót másoló, rekurzív függvény. A második klózban alkalmazott rekurziót angolul *body recursion*-nek nevezik – magyarul *törzsrekurzió*nak mondhatjuk –, ha hangsúlyozni akarják, hogy a rekurzív hívás eredményével még további műveletet kell végezni a függvény törzsében.

A második, *jobbrekurzív* – angolul: *tail recursive* – változat kevésbé szigorúan követi a matematikai definíciót, emiatt nehezebb megérteni, nehezebb hozzá kifejező, pontos fejkommentet írni. A jobbrekurziót magyarul szokták még *terminális rekurzió*nak, ritkábban *farokrekurzió*nak is nevezni, mert a rekurzív hívás az adott klózban az utolsó – befejező, lezáró – hívás, az eredményét változatlanul vissza kell adni, már semmiféle műveletet nem végzünk vele.

```
defmodule Fac do
  @spec fac(n :: integer()) :: f :: integer() # Típuspecifikáció
  # f = n! (azaz f az n faktoriálisa) # Fejkomment

  # ha az n=0 mintaillesztés sikeres
  def fac(0), do: 1

  # ha az n=0 mintaillesztés sikertelen
  def fac(n), do: n * fac(n - 1)
end

Fac.fac(5)
Fac.fac(0)
Fac.fac(1)
Fac.fac(100_000)
```

A jobbrekurzív változathoz egy plusz paraméterre van szükségünk. Ezt *akkumulátornak* szokták nevezni, mert a részeredményeket gyűjtjük benne – ahelyett, hogy a még elvégzendő műveleteket az argumentumaikkal együtt a verembe tennénk.

Az akkumulátor tehát egy segédparaméter, amit jelen esetben arra használunk, hogy a részletszorzatokat adjuk át benne a rekurzív hívásban.

Az akkumulátornak az első híváskor adunk értéket: a *fac/1* hívja meg az 1 kezdőértékkel a *fac/2*-t. Ha a *fac/1*-et 0-val hívjuk meg, akkor az eredménynek 1-nek kell lennie. Ezért a *fac/2* első klózáának a 0 esetén 1-et kell visszaadnia, nincs szükség rekurzív hívásra. Ha az első paraméter, n , nem 0, akkor kerül sor a második klózra: a rekurzív hívásban az első paraméter értékét eggyel csökkentjük ($n - 1$), a második paraméterben pedig n -nel megszorozzuk az eddig már összegyűjtött részletszorzatot ($n * a$), így alakul majd ki a $n \cdot (n - 1) \cdot \dots \cdot 1$ eredmény.

```
defmodule FacJobbrek do
  # Típuspecifikáció
  @spec fac(n :: integer()) :: f :: integer()
  # f = n! (azaz f az n faktoriálisa)
  def fac(n), do: fac(n, 1)

  @spec fac(n :: integer(), a :: integer()) :: f :: integer()
  defp fac(0, a), do: a
  defp fac(n, a), do: fac(n - 1, n * a)
end

FacJobbrek.fac(5)
FacJobbrek.fac(0)
FacJobbrek.fac(1)
FacJobbrek.fac(100_000)
```

Tapasztalható-e lényeges különbség a törzsrekurzív és a jobbrekurzív függvény futási ideje között?

A jobbrekurzív manapság főleg olyan esetekben indokolt, amikor két vagy több processz üzenetet küld egymásnak, és végtelen jobbrekurzív hívásban várnak a válaszüzenetre.

Egészlista elemeinek összege

```
defmodule Sum do
```

```
  @spec sum1(xs::[integer()]) :: sum::integer()
```

```
  # xs elemeinek összege sum
```

```
  # üres listára az első klóz illeszkedik
```

```
  def sum1([]), do: 0
```

```
  def sum1([x|xs]), do: x + sum1(xs)
```

```
  # üres listára a második klóz illeszkedik
```

```
  def sum2([x|xs]), do: x + sum2(xs)
```

```
  def sum2([]), do: 0
```

```
  # jobbrekurzív
```

```
  def sum3(xs), do: sumi(xs, 0)
```

```
  defp sumi([x|xs], sum), do: sumi(xs, sum+x)
```

```
  defp sumi([], sum), do: sum
```

```
end
```

```
1..1000 |> Range.to_list() |> Sum.sum1() |> IO.inspect()
```

```
1..1000 |> Range.to_list() |> Sum.sum2() |> IO.inspect()
```

```
1..1000 |> Range.to_list() |> Sum.sum3() |> IO.inspect()
```

```
1..90_000_000 |> Range.to_list() |> Sum.sum1() |> IO.inspect()
```

```
1..90_000_000 |> Range.to_list() |> Sum.sum2() |> IO.inspect()
```

```
1..90_000_000 |> Range.to_list() |> Sum.sum3() |> IO.inspect()
```