Deklaratív programozás 1. előadás

Kabódi László[1] Marussy Kristóf[2]
BME Számítástudományi és Információelméleti Tanszék
Mesterséges Intelligencia és Rendszertervezés Tanszék

[1] kabodi.laszlo@vik.bme.hu
[2] marussy@mit.bme.hu

2026. ősz

I. rész

Deklaratív programozás, követelmények, áttekintés

A tantárgy témája

- Deklaratív programozási nyelvek – gyakorlati megközelítésben
- Két fő irány:
  - funkcionális programozás **Elixir** nyelven,
  - logikai programozás **Prolog** nyelven.

Tartalom

Honlap, Elektronikus TanárSegéd, Teams csoport

- Honlap: `https://dp.iit.bme.hu`,
  a jelen félév honlapja: `https://dp.iit.bme.hu/dp-current`
- ETS, az Elektronikus TanárSegéd[1]
  `https://dps.iit.bme.hu/ets`, csak a tantárgy jelenlegi hallgatói tudnak belépni a Neptun-kódjukkal
- *Deklaratív programozás* Teams csoport

[1] Az ETS 2024-es új kiadása Hanák Dávidnak köszönhető. Forrás: `https://github.com/dhanak/ets/`

DP-követelmények: gyakorlatok

Gyakorlatok

- Az 1. oktatási héttől kezdve lesznek tantermi gyakorlatok, mégpedig az ötkredites VISZAD01 kurzus hallgatói számára minden héten, a háromkredites VISZAD00 kurzus hallgatói számára minden második héten. Részletes beosztás a honlapon.
- A gyakorlatok anyagát elektronikus formában a honlapon tesszük közzé.
- A tantermi gyakorlatokon nagyon ajánlott a hordozható számítógép (laptop) használata, különösen az FP-gyakorlatokon, ahol a *Livebook for Elixir* notebook-formában adjuk ki a feladatsorokat.
- További Elixir gyakorlási lehetőség az EDUX, Prolog gyakorlási lehetőség a PLWIN rendszerben (lásd honlap).

DP-követelmények: kis házi feladatok

Kis házi feladatok (KHF)

- 3 feladat funkcionális, 3 logikai programozási nyelven, ütemterv szerint.
- Kiírás a honlapon, beadás szintén elektronikus úton (ld. honlap, ETS).
- **Kötelező** legalább **két** FP és **két** LP KHF érvényes és sikeres beadása.
- A KHF-ek egyre összetettebbek és részben egymásra épülnek – érdemes minél előbb elkezdeni a KHF-ek beadását!
- Egy KHF beadása érvényes, ha az összes beadási tesztesetre jól fut le.
- A KHF-ek az ún. éles tesztelés során kapnak pontszámot (ezek az éles tesztesetek a beadási tesztesetekhez hasonlóak, de nem ugyanazok).
- Minden KHF sikeres megoldásáért – azaz az összes éles teszteset megoldásáért – 1-1 jutalompont (azaz a 100 alappont feletti pont) jár.
- Ha valaki mindhárom KHF-et megoldja az adott nyelven, azzal **megduplázza** a pontszámát, azaz 3 helyett 6 pontot kap.
- Minden **KHF**-nek külön határideje van, **pótlási lehetőség nincs**.
- A beadási határidejéig a KHF többször is beadható, az utolsót értékeljük.
- Mindenkinek el kell tudnia magyaráznia a megoldását az oktatóknak.

DP-követelmények: nagy házi feladat

Nagy házi feladat (NHF)

- Programozás mind funkcionális, mind logikai nyelven.
- Mindenkinek önállóan kell dolgoznia!
- Elvárás: hatékony (időlimit!), jól dokumentált („kommentezett”) programok.
- A programokhoz 5–10 oldalas fejlesztői dokumentáció PDF-ben.
- Az FP NHF kiírása 6. héten, az LP NHF-é 11. héten a honlapon.
- Az FP NHF beadása 8. héten, az LP NHF-é 13. héten az ETS-sel.
- A beadáskor és a pontozáskor most is külön-külön tesztsorozatot használunk (nehézségben hasonlókat, de nem azonosakat).
- Azok a programok, amelyek megoldják az éles tesztesetek 80%-át, *létraversenyen* vesznek részt (hatékonysági, gyorsasági plusz pontokért).
- Azok a hallgatók, akik **mindkét** nyelvből bejutnak a létraversenybe, és a kis házi feladatokra vonatkozó követelményeket is teljesítik, **megajánlott** jegyet kapnak.
- A beadási határidejéig az NHF többször beadható, az utolsót értékeljük.

DP-követelmények: nagy házi feladat (folyt.)

Nagy házi feladat (folyt.)

- Pontozása mindkét nyelvből:
  - helyes (azaz jó eredményt időkorláton belül adó) futás esetén a 10 teszteset mindegyikére 0,5-0,5 pont, összesen max. 5 pont;
  - a dokumentációra, a kód olvashatóságára, kommentezettségére max. 2 pont;
  - tehát nyelvenként összesen max. 7 pont szerezhető.
- Így az NHF súlya az osztályzatban: 14% (a 100 pontból 14).
- Az NHF beadása **nem kötelező, de ajánlott!** Hogy miért? Többek között
  - egy-egy nagyobb feladat megoldásával lehet igazán megérteni, megérezni a funkcionális, ill. a logikai programozási szemléletet;
  - mindkét NHF hatékony megoldásával megajánlott jegyet lehet szerezni, ami a külföldön tanulóknak, „home office”-ban dolgozóknak, sok zéhát íróknak stb. különösen hasznos lehet;
  - maguk a feladványok is érdekesek, például abból a szempontból, hogy hogyan lehet hatékonyan algoritmizálni őket;
  - lehetőséget adnak egy új szakmai területre, a korlátprogramozásba (constraint programming) való „belekóstolásra”.

DP-követelmények: zárthelyik

Nagyzárthelyik, pótzárthelyik (NZH-k, PZH-k)

- Két NZH-t tartunk, egyet a funkcionális, egyet a logikai részből.
- Két PZH-t tartunk, mindkét PZH-n bármelyik NZH-t lehet pótolni, akár mind a kettőt (de szűkebb időkerettel).
- Az FP-zéhákat terveink szerint gépteremben, *Livebook for Elixir* notebookkal kell megírni.
- Mind a funkcionális, mind a logikai részből **kötelező** a zárthelyi érvényes teljesítése, kivéve megajánlott jegy esetén (lásd alább).
- 40%-os szabály: **nyelvenként** a maximális pontszám 40%-a kell az érvényességhez.
- Zárthelyi időpontok: lásd a honlapon.
- A zárthelyik súlya az osztályzatban: 43%–43% (a 100 pontból max. 86).

DP-követelmények: megajánlott jegy, önálló feladatmegoldás

A megajánlott jegy feltételei

- Alapfeltételek: a KHF követelmények teljesítése; az NHF „megvédése”.
- Azok a programok, amelyek megoldják az éles tesztesetek 80%-át az időkorláton belül, *létraversenyen* vesznek részt
  - Jóval nehezebb létra-tesztesetek: verseny a hallgatók között, hogy kinek a programja hányat tud megoldani időre
- Jó (4): a nagy házi feladat mindkét nyelvből bejut a létraversenybe.
- Jeles (5): legalább 40%-os eredmény a létraversenyen, mindkét nyelvből.

DP-követelmények: megajánlott jegy, önálló feladatmegoldás

A megajánlott jegy feltételei

- Alapfeltételek: a KHF követelmények teljesítése; az NHF „megvédése”.
- Azok a programok, amelyek megoldják az éles tesztesetek 80%-át az időkorláton belül, *létraversenyen* vesznek részt
  - Jóval nehezebb létra-tesztesetek: verseny a hallgatók között, hogy kinek a programja hányat tud megoldani időre
- Jó (4): a nagy házi feladat mindkét nyelvből bejut a létraversenybe.
- Jeles (5): legalább 40%-os eredmény a létraversenyen, mindkét nyelvből.

Az NHF „megvédése” azt jelenti, hogy a hallgatónak személyes (vagy távjelenléti) formában el kell magyaráznia a tantárgy egy oktatójának, hogyan oldotta meg az NHF-et, és válaszolni kell tudnia az oktató kérdéseire.

Magától értetődő, hogy minden házi feladatot önállóan kell elkészíteni. Másolás esetén kötelesek vagyunk fegyelmi eljárást indítani, lásd a 3/2011. (III.23.) rektori utasításban a *"Beadandó feladat, szakdolgozat, diplomaterv elkészíttetése mással"* tételt.

DP-követelmények: IMSc-pontozás

- A tantárgyból kétféle módon szerezhető IMSc pont:
  - egyes zárthelyik során pluszfeladatok megoldásával (max. 13 pont),
  - a létraversenyen a megajánlott jeles érdemjegyhez szükséges 40%-os teljesítés felett minden további 10%-os teljesítésért mindkét nyelv esetén 1–1 pont (összesen max. 12 pont).
- A hallgató a fenti módokon szerzett pontok összegét kapja, de a VISZAD00 kurzus esetén legfeljebb 15 IMSc pontot.
- Az IMSc pontok gyűjtése teljesen független a tantárgyban szerezhető ZH és HF pontoktól. Ezen pontok megszerzése és a fakultatív feladatok megoldása nélkül is jeles szinten teljesíthetők a tantárgy követelményei.
- Az IMSc pontok megszerzése az IMSc programban nem résztvevő hallgatók számára is lehetséges.

Tartalom

Deklaratív programozás

- A „deklaratív programozás” jelentése (Wikipédia):
  (. . .) a deklaratív programozás egy programozási paradigma, (. . .) amely kifejezi a számítás logikáját anélkül, hogy leírná a vezérlési folyamatát.
  (. . .) declarative programming is a programming paradigm (. . .) that expresses the logic of a computation without describing its control flow.

Deklaratív programozás

- A „deklaratív programozás” jelentése (Wikipédia):
  (. . .) a deklaratív programozás egy programozási paradigma, (. . .) amely kifejezi a számítás logikáját anélkül, hogy leírná a vezérlési folyamatát.
  (. . .) declarative programming is a programming paradigm (. . .) that expresses the logic of a computation without describing its control flow.
- A „deklaratív” jelző értelmezése (Topszótár):
  - kijelentő, kinyilatkoztató
  - ellentmondást nem tűrő
- A „deklaratív” jelző a nyelvészetből származik, pl. „kijelentő mondat”.

Deklaratív programozás

- A „deklaratív programozás” jelentése (Wikipédia):
  (. . .) a deklaratív programozás egy programozási paradigma, (. . .) amely kifejezi a számítás logikáját anélkül, hogy leírná a vezérlési folyamatát.
  (. . .) declarative programming is a programming paradigm (. . .) that expresses the logic of a computation without describing its control flow.
- A „deklaratív” jelző értelmezése (Topszótár):
  - kijelentő, kinyilatkoztató
  - ellentmondást nem tűrő
- A „deklaratív” jelző a nyelvészetből származik, pl. „kijelentő mondat”.
- Milyen más mondatfajták vannak?

Deklaratív programozás

- A „deklaratív programozás” jelentése (Wikipédia):
  (. . .) a deklaratív programozás egy programozási paradigma, (. . .) amely kifejezi a számítás logikáját anélkül, hogy leírná a vezérlési folyamatát.
  (. . .) declarative programming is a programming paradigm (. . .) that expresses the logic of a computation without describing its control flow.
- A „deklaratív” jelző értelmezése (Topszótár):
  - kijelentő, kinyilatkoztató
  - ellentmondást nem tűrő
- A „deklaratív” jelző a nyelvészetből származik, pl. „kijelentő mondat”.
- Milyen más mondatfajták vannak?
  - kérdő,
  - felszólító (imperatív) stb.

Deklaratív programozás

- A „deklaratív programozás” jelentése (Wikipédia):
  (. . .) a deklaratív programozás egy programozási paradigma, (. . .) amely kifejezi a számítás logikáját anélkül, hogy leírná a vezérlési folyamatát.
  (. . .) declarative programming is a programming paradigm (. . .) that expresses the logic of a computation without describing its control flow.
- A „deklaratív” jelző értelmezése (Topszótár):
  - kijelentő, kinyilatkoztató
  - ellentmondást nem tűrő
- A „deklaratív” jelző a nyelvészetből származik, pl. „kijelentő mondat”.
- Milyen más mondatfajták vannak?
  - kérdő,
  - felszólító (imperatív) stb.
- A számítógépek belső nyelve (gépi kódja) alapvetően felszólító jellegű: add hozzá, szorozd meg, ugorj, . . .
- A magas szintű programnyelvek többsége is imperatív:
  `while ... do ...`, `goto ...`, értékadás (írd felül a változó értékét) stb.

Deklaratív programozás (folyt.)

- Lehet-e pl. C-ben deklaratívan programozni?
  Igen, pl. ciklus helyett rekurzió használatával:

```
int fact(int n) {if (n > 0) return n * fact(n-1);
                 else return 1;
                }
```

- Igen, de ez lassú! $\leadsto$ Ún. jobbrekurzív (farokrekurzív, tail recursive) változata a ciklussal azonos hatékonyságú kóddá fordul.
- Mi az előnye a deklaratív szemléletnek?

Deklaratív programozás (folyt.)

- Lehet-e pl. C-ben deklaratívan programozni?
  Igen, pl. ciklus helyett rekurzió használatával:

```
int fact(int n) {if (n > 0) return n * fact(n-1);
                 else return 1;
                }
```

- Igen, de ez lassú! $\rightsquigarrow$ Ún. jobbrekurzív (farokrekurzív, tail recursive) változata a ciklussal azonos hatékonyságú kóddá fordul.
- Mi az előnye a deklaratív szemléletnek? A programkód sokkal közelebb áll a specifikációhoz, helyességéről sokkal könnyebb meggyőződni.
- A deklaratív megközelítés jelmondata:

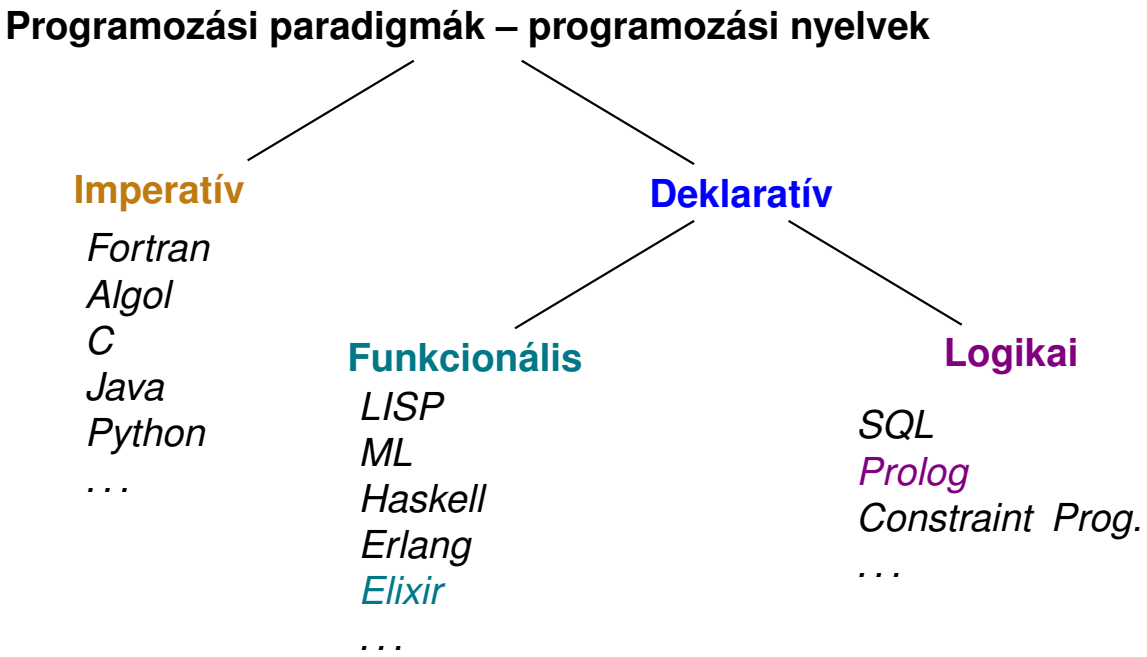
  MIT és nem HOGYAN

  vagy kicsit enyhítve:

  Inkább MIT, mint HOGYAN
  (WHAT rather than HOW)
- A deklaratív nyelvekben a változó a matematika változófogalmának felel meg: egyetlen, esetleg még ismeretlen értéket jelöl.

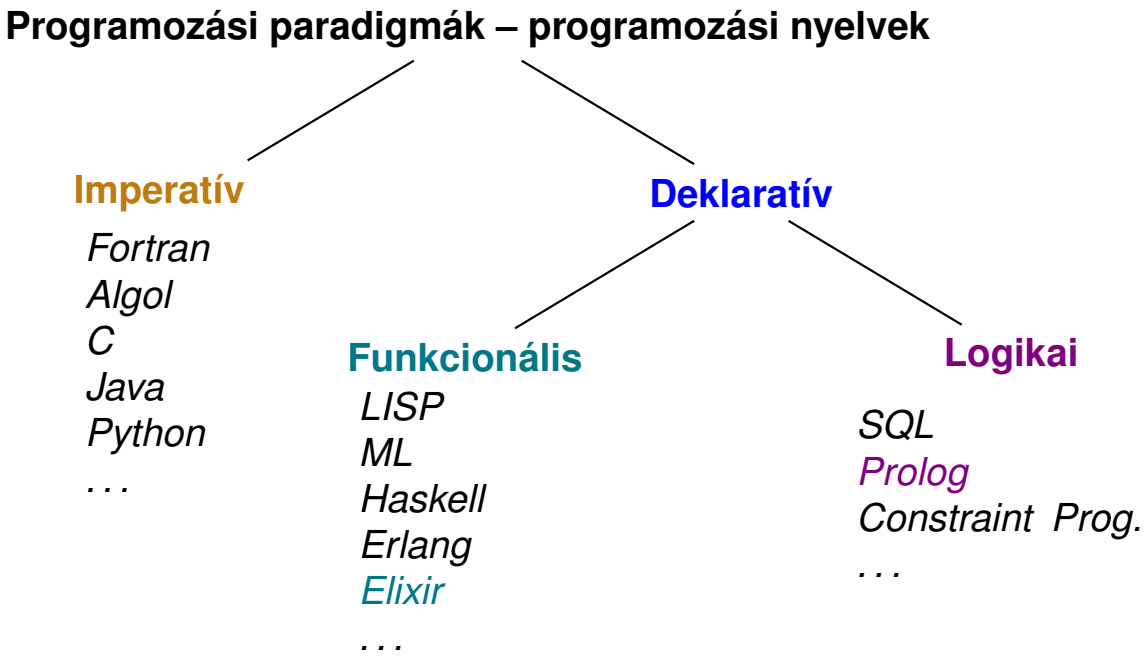
Funkcionális és logikai programozás

- A deklaratív programozás két fő ága egy-egy alapvető matematikai fogalomhoz kapcsolódik:
  - Funkcionális programozás (FP) – függvények
  - Logikai programozás (LP) – relációk



Funkcionális és logikai programozás

- A deklaratív programozás két fő ága egy-egy alapvető matematikai fogalomhoz kapcsolódik:
  - Funkcionális programozás (FP) – függvények
  - Logikai programozás (LP) – relációk



- A kurzus tárgya: az Elixir funkcionális és a Prolog logikai programozási nyelv.

Példa: Listák összefűzése Elixirben és Prologban

- Az Elixir és a Prolog nyelvek (közös) szintaxisa a láncolt listák jelölésére:
  - `[]` – üres lista
  - `[Head|Tail]` – egy olyan lista, amelynek feje `Head`, farka pedig `Tail`
- Példa: az `1`, `2`, `3` számokból álló lista: `[1|[2|[3|[]]]]`
- Ugyanez az adatstruktúra tömörebben is leírható: `[1,2,3]`

Példa: Listák összefűzése Elixirben és Prologban

- Az Elixir és a Prolog nyelvek (közös) szintaxisa a láncolt listák jelölésére:
  - `[]` – üres lista
  - `[Head|Tail]` – egy olyan lista, amelynek feje `Head`, farka pedig `Tail`
- Példa: az `1`, `2`, `3` számokból álló lista: `[1|[2|[3|[]]]]`
- Ugyanez az adatstruktúra tömörebben is leírható: `[1,2,3]`
- Írjunk egy `app` nevű kétargumentumú Elixir függvényt (`app/2`):

```
#   app(l1,   l2): l1 és l2 listák összefűzöttje (l1⊕l2)
```

Példa: Listák összefűzése Elixirben és Prologban

- Az Elixir és a Prolog nyelvek (közös) szintaxisa a láncolt listák jelölésére:
  - `[]` – üres lista
  - `[Head|Tail]` – egy olyan lista, amelynek feje `Head`, farka pedig `Tail`
- Példa: az `1`, `2`, `3` számokból álló lista: `[1|[2|[3|[]]]]`
- Ugyanez az adatstruktúra tömörebben is leírható: `[1,2,3]`
- Írjunk egy `app` nevű kétargumentumú Elixir függvényt (`app/2`):

```
#   app(l1,   l2): l1 és l2 listák összefűzöttje (l1⊕l2)
def app([],    b) do              b end       # [] ⊕ b = b
```

Példa: Listák összefűzése Elixirben és Prologban

- Az Elixir és a Prolog nyelvek (közös) szintaxisa a láncolt listák jelölésére:
  - `[]` – üres lista
  - `[Head|Tail]` – egy olyan lista, amelynek feje `Head`, farka pedig `Tail`
- Példa: az `1`, `2`, `3` számokból álló lista: `[1|[2|[3|[]]]]`
- Ugyanez az adatstruktúra tömörebben is leírható: `[1,2,3]`
- Írjunk egy `app` nevű kétargumentumú Elixir függvényt (`app/2`):

```
#   app(l1,   l2): l1 és l2 listák összefűzöttje (l1⊕l2)
def app([],    b) do             b end       # [] ⊕ b = b
def app([x|a], b) do [x|app(a,b)] end       # [x|a] ⊕ b = [x|a⊕b]
```

Példa: Listák összefűzése Elixirben és Prologban

- Az Elixir és a Prolog nyelvek (közös) szintaxisa a láncolt listák jelölésére:
  - `[]` – üres lista
  - `[Head|Tail]` – egy olyan lista, amelynek feje `Head`, farka pedig `Tail`
- Példa: az `1`, `2`, `3` számokból álló lista: `[1|[2|[3|[]]]]`
- Ugyanez az adatstruktúra tömörebben is leírható: `[1,2,3]`
- Írjunk egy `app` nevű kétargumentumú Elixir függvényt (`app/2`):

```
#   app(l1,   l2): l1 és l2 listák összefűzöttje (l1⊕l2)
def app([],    b) do              b end       # [] ⊕ b = b
def app([x|a], b) do [x|app(a,b)] end       # [x|a] ⊕ b = [x|a⊕b]
```

- Az `app` függvénynek egy 3-argumentumú Prolog eljárás (másnéven predikátum) felel meg (`app/3`), a 3. argumentum az Elixir fv. eredménye):

```
%   app(L1, L2, L12): L1 és L2 listák összefűzöttje L12 (L1⊕L2=L12)
```

Példa: Listák összefűzése Elixirben és Prologban

- Az Elixir és a Prolog nyelvek (közös) szintaxisa a láncolt listák jelölésére:
  - `[]` – üres lista
  - `[Head|Tail]` – egy olyan lista, amelynek feje `Head`, farka pedig `Tail`
- Példa: az `1`, `2`, `3` számokból álló lista: `[1|[2|[3|[]]]]`
- Ugyanez az adatstruktúra tömörebben is leírható: `[1,2,3]`
- Írjunk egy `app` nevű kétargumentumú Elixir függvényt (`app/2`):

```
#   app(l1,   l2): l1 és l2 listák összefűzöttje (l1⊕l2)
def app([],    b) do              b end       # [] ⊕ b = b
def app([x|a], b) do [x|app(a,b)] end       # [x|a] ⊕ b = [x|a⊕b]
```

- Az `app` függvénynek egy 3-argumentumú Prolog eljárás (másnéven predikátum) felel meg (`app/3`), a 3. argumentum az Elixir fv. eredménye):

```
%   app(L1, L2, L12): L1 és L2 listák összefűzöttje L12 (L1⊕L2=L12)
    app([],    B,              B).      % [] ⊕ B = B
```

Példa: Listák összefűzése Elixirben és Prologban

- Az Elixir és a Prolog nyelvek (közös) szintaxisa a láncolt listák jelölésére:
  - `[]` – üres lista
  - `[Head|Tail]` – egy olyan lista, amelynek feje `Head`, farka pedig `Tail`
- Példa: az `1`, `2`, `3` számokból álló lista: `[1|[2|[3|[]]]]`
- Ugyanez az adatstruktúra tömörebben is leírható: `[1,2,3]`
- Írjunk egy `app` nevű kétargumentumú Elixir függvényt (`app/2`):

```
#   app(l1,   l2): l1 és l2 listák összefűzöttje (l1⊕l2)
def app([],    b) do              b end       # [] ⊕ b = b
def app([x|a], b) do [x|app(a,b)] end       # [x|a] ⊕ b = [x|a⊕b]
```

- Az `app` függvénynek egy 3-argumentumú Prolog eljárás (másnéven predikátum) felel meg (`app/3`), a 3. argumentum az Elixir fv. eredménye):

```
%   app(L1, L2, L12): L1 és L2 listák összefűzöttje L12 (L1⊕L2=L12)
    app([],    B,              B).          % [] ⊕ B = B
    app([X|A], B,    [X|       C]) :-       % [X|A] ⊕ B = [X|C] ha
```

Példa: Listák összefűzése Elixirben és Prologban

- Az Elixir és a Prolog nyelvek (közös) szintaxisa a láncolt listák jelölésére:
  - `[]` – üres lista
  - `[Head|Tail]` – egy olyan lista, amelynek feje `Head`, farka pedig `Tail`
- Példa: az `1`, `2`, `3` számokból álló lista: `[1|[2|[3|[]]]]`
- Ugyanez az adatstruktúra tömörebben is leírható: `[1,2,3]`
- Írjunk egy `app` nevű kétargumentumú Elixir függvényt (`app/2`):

```
#   app(l1,   l2): l1 és l2 listák összefűzöttje (l1⊕l2)
def app([],    b) do              b end       # [] ⊕ b = b
def app([x|a], b) do [x|app(a,b)] end       # [x|a] ⊕ b = [x|a⊕b]
```

- Az `app` függvénynek egy 3-argumentumú Prolog eljárás (másnéven predikátum) felel meg (`app/3`), a 3. argumentum az Elixir fv. eredménye):

```
%   app(L1, L2, L12): L1 és L2 listák összefűzöttje L12 (L1⊕L2=L12)
    app([],    B,              B).          % [] ⊕ B = B
    app([X|A], B,    [X|       C]) :-       % [X|A] ⊕ B = [X|C] ha
           app(A, B, C).                    %      A ⊕ B = C
```

Példa: Listák összefűzése Elixirben és Prologban

- Az Elixir és a Prolog nyelvek (közös) szintaxisa a láncolt listák jelölésére:
  - `[]` – üres lista
  - `[Head|Tail]` – egy olyan lista, amelynek feje `Head`, farka pedig `Tail`
- Példa: az `1`, `2`, `3` számokból álló lista: `[1|[2|[3|[]]]]`
- Ugyanez az adatstruktúra tömörebben is leírható: `[1,2,3]`
- Írjunk egy `app` nevű kétargumentumú Elixir függvényt (`app/2`):

```
#   app(l1,   l2): l1 és l2 listák összefűzöttje (l1⊕l2)
def app([],    b) do              b end       # [] ⊕ b = b
def app([x|a], b) do [x|app(a,b)] end       # [x|a] ⊕ b = [x|a⊕b]
```

- Az `app` függvénynek egy 3-argumentumú Prolog eljárás (másnéven predikátum) felel meg (`app/3`), a 3. argumentum az Elixir fv. eredménye):

```
%   app(L1, L2, L12): L1 és L2 listák összefűzöttje L12 (L1⊕L2=L12)
    app([],    B,          B).          % [] ⊕ B = B
    app([X|A], B,    [X|       C]) :-   % [X|A] ⊕ B = [X|C] ha
           app(A, B, C).                %      A ⊕ B = C
```

- Az eljáráshívások (a függvényhívásokkal ellentétben) nem ágyazhatók egymásba, ezért van szükség a `C` segédváltozóra.
- **DE**: az `app/3` Prolog eljárás jobbrekurzív (azaz ciklussá fordul!)

Az app/3 Prolog eljárás használata

- Prologban megengedett, hogy egy adatstruktúrában behelyettesítetlen változó szerepeljen:

```
app([],    B, B).
app([X|A], B, L) :- L = [X|C], app(A, B, C).
```

Az `app/3` Prolog eljárás használata

- Prologban megengedett, hogy egy adatstruktúrában behelyettesítetlen változó szerepeljen:

```
app([],    B, B).
app([X|A], B, L) :- L = [X|C], app(A, B, C).
```

- A Prolog változó pointerként is felfogható: pl. `app` először felépíti az eredménylista első láncszemét (`[X|C]`), majd az `app` jobbrekurzív hívásával kitölti az eredménylista `C` által mutatott farkát

Az `app/3` Prolog eljárás használata

- Prologban megengedett, hogy egy adatstruktúrában behelyettesítetlen változó szerepeljen:

```
app([],    B, B).
app([X|A], B, L) :- L = [X|C], app(A, B, C).
```

- A Prolog változó pointerként is felfogható: pl. `app` először felépíti az eredménylista első láncszemét (`[X|C]`), majd az `app` jobbrekurzív hívásával kitölti az eredménylista `C` által mutatott farkát, pl.

`app([1], [2], L)` $\Rightarrow$ `L = [1|C], app([], [2], C)` $\Rightarrow$ `L = [1|[2]] = [1,2]`

Az `app/3` Prolog eljárás használata

- Prologban megengedett, hogy egy adatstruktúrában behelyettesítetlen változó szerepeljen:

```
app([],    B, B).
app([X|A], B, L) :- L = [X|C], app(A, B, C).
```

- A Prolog változó pointerként is felfogható: pl. `app` először felépíti az eredménylista első láncszemét (`[X|C]`), majd az `app` jobbrekurzív hívásával kitölti az eredménylista `C` által mutatott farkát, pl.

`app([1], [2], L)` $\Rightarrow$ `L = [1|C], app([], [2], C)` $\Rightarrow$ `L = [1|[2]] = [1,2]`

- Az `app/3` eljárás nemcsak összefűzésre használható:

```
| ?- app([1,2], [3,4], L).        ⟹
```

Az `app/3` Prolog eljárás használata

- Prologban megengedett, hogy egy adatstruktúrában behelyettesítetlen változó szerepeljen:

```
app([],    B, B).
app([X|A], B, L) :- L = [X|C], app(A, B, C).
```

- A Prolog változó pointerként is felfogható: pl. `app` először felépíti az eredménylista első láncszemét (`[X|C]`), majd az `app` jobbrekurzív hívásával kitölti az eredménylista `C` által mutatott farkát, pl.

`app([1], [2], L)` $\Rightarrow$ `L = [1|C], app([], [2], C)` $\Rightarrow$ `L = [1|[2]] = [1,2]`

- Az `app/3` eljárás nemcsak összefűzésre használható:

```
| ?- app([1,2], [3,4], L).      ⟹    L = [1,2,3,4] ?
```

Az app/3 Prolog eljárás használata

- Prologban megengedett, hogy egy adatstruktúrában behelyettesítetlen változó szerepeljen:

```
app([],    B, B).
app([X|A], B, L) :- L = [X|C], app(A, B, C).
```

- A Prolog változó pointerként is felfogható: pl. app először felépíti az eredménylista első láncszemét ([X|C]), majd az app jobbrekurzív hívásával kitölti az eredménylista C által mutatott farkát, pl.

app([1], [2], L) $\Rightarrow$ L = [1|C], app([], [2], C) $\Rightarrow$ L = [1|[2]] = [1,2]

- Az app/3 eljárás nemcsak összefűzésre használható:

```
| ?- app([1,2], [3,4], L).      ⟹   L = [1,2,3,4] ? ; no
```

Az app/3 Prolog eljárás használata

- Prologban megengedett, hogy egy adatstruktúrában behelyettesítetlen változó szerepeljen:

```
app([],    B, B).
app([X|A], B, L) :- L = [X|C], app(A, B, C).
```

- A Prolog változó pointerként is felfogható: pl. app először felépíti az eredménylista első láncszemét ([X|C]), majd az app jobbrekurzív hívásával kitölti az eredménylista C által mutatott farkát, pl.

app([1], [2], L) $\Rightarrow$ L = [1|C], app([], [2], C) $\Rightarrow$ L = [1|[2]] = [1,2]

- Az app/3 eljárás nemcsak összefűzésre használható:

```
| ?- app([1,2], [3,4], L).        ⟹   L = [1,2,3,4] ? ; no
| ?- app([1,2], B, [1,2,3,4]).    ⟹
```

Az app/3 Prolog eljárás használata

- Prologban megengedett, hogy egy adatstruktúrában behelyettesítetlen változó szerepeljen:

```
app([],    B, B).
app([X|A], B, L) :- L = [X|C], app(A, B, C).
```

- A Prolog változó pointerként is felfogható: pl. app először felépíti az eredménylista első láncszemét ([X|C]), majd az app jobbrekurzív hívásával kitölti az eredménylista C által mutatott farkát, pl.

app([1], [2], L) $\Rightarrow$ L = [1|C], app([], [2], C) $\Rightarrow$ L = [1|[2]] = [1,2]

- Az app/3 eljárás nemcsak összefűzésre használható:

```
| ?- app([1,2], [3,4], L).      ⟹   L = [1,2,3,4] ? ; no
| ?- app([1,2], B, [1,2,3,4]).  ⟹   B = [3,4] ?
```

Az app/3 Prolog eljárás használata

- Prologban megengedett, hogy egy adatstruktúrában behelyettesítetlen változó szerepeljen:

```
app([],    B, B).
app([X|A], B, L) :- L = [X|C], app(A, B, C).
```

- A Prolog változó pointerként is felfogható: pl. app először felépíti az eredménylista első láncszemét ([X|C]), majd az app jobbrekurzív hívásával kitölti az eredménylista C által mutatott farkát, pl.

app([1], [2], L) $\Rightarrow$ L = [1|C], app([], [2], C) $\Rightarrow$ L = [1|[2]] = [1,2]

- Az app/3 eljárás nemcsak összefűzésre használható:

```
| ?- app([1,2], [3,4], L).        ⟹   L = [1,2,3,4] ? ; no
| ?- app([1,2], B, [1,2,3,4]).    ⟹   B = [3,4] ? ; no
| ?- app([1,2], B, [1,3,4,5]).    ⟹
```

Az `app/3` Prolog eljárás használata

- Prologban megengedett, hogy egy adatstruktúrában behelyettesítetlen változó szerepeljen:

```
app([],    B, B).
app([X|A], B, L) :- L = [X|C], app(A, B, C).
```

- A Prolog változó pointerként is felfogható: pl. `app` először felépíti az eredménylista első láncszemét (`[X|C]`), majd az `app` jobbrekurzív hívásával kitölti az eredménylista `C` által mutatott farkát, pl.

`app([1], [2], L)` $\Rightarrow$ `L = [1|C], app([], [2], C)` $\Rightarrow$ `L = [1|[2]] = [1,2]`

- Az `app/3` eljárás nemcsak összefűzésre használható:

```
| ?- app([1,2], [3,4], L).        ⟹   L = [1,2,3,4] ? ; no
| ?- app([1,2], B, [1,2,3,4]).    ⟹   B = [3,4] ? ; no
| ?- app([1,2], B, [1,3,4,5]).    ⟹   no
```

Az app/3 Prolog eljárás használata

- Prologban megengedett, hogy egy adatstruktúrában behelyettesítetlen változó szerepeljen:

```
app([],    B, B).
app([X|A], B, L) :- L = [X|C], app(A, B, C).
```

- A Prolog változó pointerként is felfogható: pl. app először felépíti az eredménylista első láncszemét ([X|C]), majd az app jobbrekurzív hívásával kitölti az eredménylista C által mutatott farkát, pl.

app([1], [2], L) $\Rightarrow$ L = [1|C], app([], [2], C) $\Rightarrow$ L = [1|[2]] = [1,2]

- Az app/3 eljárás nemcsak összefűzésre használható:

```
| ?- app([1,2], [3,4], L).        ⟹   L = [1,2,3,4] ? ; no
| ?- app([1,2], B, [1,2,3,4]).    ⟹   B = [3,4] ? ; no
| ?- app([1,2], B, [1,3,4,5]).    ⟹   no
| ?- app(A, B, [1,2]).            ⟹
```

Az app/3 Prolog eljárás használata

- Prologban megengedett, hogy egy adatstruktúrában behelyettesítetlen változó szerepeljen:

```
app([],    B, B).
app([X|A], B, L) :- L = [X|C], app(A, B, C).
```

- A Prolog változó pointerként is felfogható: pl. app először felépíti az eredménylista első láncszemét ([X|C]), majd az app jobbrekurzív hívásával kitölti az eredménylista C által mutatott farkát, pl.

app([1], [2], L) $\Rightarrow$ L = [1|C], app([], [2], C) $\Rightarrow$ L = [1|[2]] = [1,2]

- Az app/3 eljárás nemcsak összefűzésre használható:

```
| ?- app([1,2], [3,4], L).        ⟹   L = [1,2,3,4] ? ; no
| ?- app([1,2], B, [1,2,3,4]).    ⟹   B = [3,4] ? ; no
| ?- app([1,2], B, [1,3,4,5]).    ⟹   no
| ?- app(A, B, [1,2]).            ⟹   A = [], B = [1,2] ? ;
                                       A = [1],  B = [2] ? ;
                                       A = [1,2], B = [] ? ; no
```

II. rész

Elixir: fő jellemzői, telepítés, használat

Tartalom

Az Elixir nyelv fő jellemzői

- Funkcionális
- A nyelvben minden kifejezés
  (nincs statement / expression megkülönböztetés)
- Mintaillesztés
- Rekurzió és magasabb rendű függvények (ciklusok helyett)
- Dinamikusan típusos
- Nincs semmi megosztva, a processzek üzenetekkel kommunikálnak
- Erlang függvények hívhatók Elixirből, Elixir függvények Erlangból

Mintaillesztés

- Mintakifejezés, röviden minta: termhez hasonló olyan kifejezés, amelyben nincs függvénykifejezés, de lehet benne szabad változó
- Egy szabad változó mindenre illeszkedik, az értékét az illeszkedő kifejezés megfelelő részéhez *kötjük* a sikeres illesztéskor

Mintaillesztés

- Mintakifejezés, röviden minta: termhez hasonló olyan kifejezés, amelyben nincs függvénykifejezés, de lehet benne szabad változó
- Egy szabad változó mindenre illeszkedik, az értékét az illeszkedő kifejezés megfelelő részéhez *kötjük* a sikeres illesztéskor

- Egyszerű változó kötés: `x = 5`, az `x` mintát illeszti az 5 kifejezésre

Mintaillesztés

- Mintakifejezés, röviden minta: termhez hasonló olyan kifejezés, amelyben nincs függvénykifejezés, de lehet benne szabad változó
- Egy szabad változó mindenre illeszkedik, az értékét az illeszkedő kifejezés megfelelő részéhez *kötjük* a sikeres illesztéskor

- Egyszerű változó kötés: `x = 5`, az `x` mintát illeszti az 5 kifejezésre
- `[hd|tl] = valami()`: `hd` és `tl` változókat köti a `valami()` által visszaadott lista fejéhez és farkához
  - hibával leáll, ha a `valami()` üres listával (vagy bármi mással) tér vissza: `(MatchError) no match of right hand side value`

Mintaillesztés

- Mintakifejezés, röviden minta: termhez hasonló olyan kifejezés, amelyben nincs függvénykifejezés, de lehet benne szabad változó
- Egy szabad változó mindenre illeszkedik, az értékét az illeszkedő kifejezés megfelelő részéhez *kötjük* a sikeres illesztéskor

- Egyszerű változó kötés: `x = 5`, az `x` mintát illeszti az 5 kifejezésre
- `[hd|tl] = valami()`: `hd` és `tl` változókat köti a `valami()` által visszaadott lista fejéhez és farkához
  - hibával leáll, ha a `valami()` üres listával (vagy bármi mással) tér vissza: `(MatchError) no match of right hand side value`
- Mintaillesztés `case` kifejezéssel

```
case valami() do
  [hd|tl] -> ... # Ha nem üres lista
  [] -> ... # Ha üres lista
  _ -> ... # Bármi más
end
```

Mintaillesztés

- Mintakifejezés, röviden minta: termhez hasonló olyan kifejezés, amelyben nincs függvénykifejezés, de lehet benne szabad változó
- Egy szabad változó mindenre illeszkedik, az értékét az illeszkedő kifejezés megfelelő részéhez *kötjük* a sikeres illesztéskor

- Egyszerű változó kötés: `x = 5`, az `x` mintát illeszti az 5 kifejezésre
- `[hd|tl] = valami()`: `hd` és `tl` változókat köti a `valami()` által visszaadott lista fejéhez és farkához
  - hibával leáll, ha a `valami()` üres listával (vagy bármi mással) tér vissza: `(MatchError) no match of right hand side value`
- Mintaillesztés `case` kifejezéssel és őrfeltétellel

```
case valami() do
  [hd|tl] when is_integer(hd) -> hd * 2 # Ha nem üres lista
  [] -> 0 # Ha üres lista
  _ -> nil # Bármi más: hibajelzés
end
```

- Függvényhívás: az aktuális paramétereket *illesztjük* a formális paraméterekre az egyes *klózokban*

Rekurzió

- Lineáris rekurzió: egy rekurzív hívás a függvényben

```
def fac(0), do: 1 # 1. klóz: Alapeset
def fac(n): do: n * fac(n - 1) # 2. klóz: Rekurzív eset
```

Rekurzió

- Lineáris rekurzió: egy rekurzív hívás a függvényben

```
def fac(0), do: 1 # 1. klóz: Alapeset
def fac(n): do: n * fac(n - 1) # 2. klóz: Rekurzív eset
```

- Jobbrekurzív (farokrekurzív, tail recursive): rekurzív hívás visszatérési pozícióban → hatékonyabb gépi kód, nem kell stack

```
def fac(n), do: fac(n, 1) # Segédfüggvény meghívása akkumulátor paraméterrel
def fac(0, a), do: a
def fac(n, a), do: fac(n - 1, n * a)
```

Rekurzió

- Lineáris rekurzió: egy rekurzív hívás a függvényben

```
def fac(0), do: 1 # 1. klóz: Alapeset
def fac(n): do: n * fac(n - 1) # 2. klóz: Rekurzív eset
```

- Jobbrekurzív (farokrekurzív, tail recursive): rekurzív hívás visszatérési pozícióban → hatékonyabb gépi kód, nem kell stack

```
def fac(n), do: fac(n, 1) # Segédfüggvény meghívása akkumulátor paraméterrel
def fac(0, a), do: a
def fac(n, a), do: fac(n - 1, n * a)
```

- Elágazó rekurzíció: több rekurzív hívás, sokszor nem hatékony

```
def fib(n) when n <= 1, do: n # 1. klóz őrfeltétellel: Alapeset
def fib(n), do: fib(n - 1) + fib(n - 2) # 2. klóz: Rekurzív eset
```

Rekurzió

- Lineáris rekurzió: egy rekurzív hívás a függvényben

```
def fac(0), do: 1 # 1. klóz: Alapeset
def fac(n): do: n * fac(n - 1) # 2. klóz: Rekurzív eset
```

- Jobbrekurzív (farokrekurzív, tail recursive): rekurzív hívás visszatérési pozícióban → hatékonyabb gépi kód, nem kell stack

```
def fac(n), do: fac(n, 1) # Segédfüggvény meghívása akkumulátor paraméterrel
def fac(0, a), do: a
def fac(n, a), do: fac(n - 1, n * a)
```

- Elágazó rekurzíció: több rekurzív hívás, sokszor nem hatékony

```
def fib(n) when n <= 1, do: n # 1. klóz őrfeltétellel: Alapeset
def fib(n), do: fib(n - 1) + fib(n - 2) # 2. klóz: Rekurzív eset
```

- Lineáris rekurzív adatszerkezetek: pl. `[...]` *egyszeresen láncolt listák*
  - Alapeset: `[]` üres lista
  - Rekurzív eset: legalább egy elemű `[H|T]`, ahol `T` is egy lista (`[]` vagy legalább egy elemű)

Rekurzió

- Lineáris rekurzió: egy rekurzív hívás a függvényben

```
def fac(0), do: 1 # 1. klóz: Alapeset
def fac(n): do: n * fac(n - 1) # 2. klóz: Rekurzív eset
```

- Jobbrekurzív (farokrekurzív, tail recursive): rekurzív hívás visszatérési pozícióban → hatékonyabb gépi kód, nem kell stack

```
def fac(n), do: fac(n, 1) # Segédfüggvény meghívása akkumulátor paraméterrel
def fac(0, a), do: a
def fac(n, a), do: fac(n - 1, n * a)
```

- Elágazó rekurzíció: több rekurzív hívás, sokszor nem hatékony

```
def fib(n) when n <= 1, do: n # 1. klóz őrfeltétellel: Alapeset
def fib(n), do: fib(n - 1) + fib(n - 2) # 2. klóz: Rekurzív eset
```

- Lineáris rekurzív adatszerkezetek: pl. `[...]` *egyszeresen láncolt listák*
  - Alapeset: `[]` üres lista
  - Rekurzív eset: legalább egy elemű `[H|T]`, ahol `T` is egy lista (`[]` vagy legalább egy elemű)
- Elágazó rekurzív adatszerekezetek: pl. bináris fa
- Algebrai módszer: rekurzív adatszerekezetek feldolgozása mintaillesztéssel rekurzív függvényekkel

Magasabb rendű függvények

- A függvény is érték: változóhoz köthető, adatstruktúra eleme lehet, függvény eredménye lehet, paraméterként átadható stb.
  Azaz: a függvény is ún. *first class citizen, teljes jogú polgár*

Magasabb rendű függvények

- A függvény is érték: változóhoz köthető, adatstruktúra eleme lehet, függvény eredménye lehet, paraméterként átadható stb.
  Azaz: a függvény is ún. *first class citizen, teljes jogú polgár*

```
> L.map [1, 2, 3], fn(x) -> 2 * x end
[2, 4, 6]
```

Magasabb rendű függvények

- A függvény is érték: változóhoz köthető, adatstruktúra eleme lehet, függvény eredménye lehet, paraméterként átadható stb.
  Azaz: a függvény is ún. *first class citizen, teljes jogú polgár*

```
> L.map [1, 2, 3], fn(x) -> 2 * x end
[2, 4, 6]
```

- Egyre gyakoribb az objektumorientált nyelvekben is
  - Java: `List.of(1, 2, 3).stream().map(x -> 2 * x).toList()`
  - C#: `new List<int>{1, 2, 3}.Select(x => 2 * x)`

Magasabb rendű függvények

- A függvény is érték: változóhoz köthető, adatstruktúra eleme lehet, függvény eredménye lehet, paraméterként átadható stb.
  Azaz: a függvény is ún. *first class citizen, teljes jogú polgár*

```
> L.map [1, 2, 3], fn(x) -> 2 * x end
[2, 4, 6]
```

- Egyre gyakoribb az objektumorientált nyelvekben is
  - Java: `List.of(1, 2, 3).stream().map(x -> 2 * x).toList()`
  - C#: `new List<int>{1, 2, 3}.Select(x => 2 * x)`
- Ciklusok helyett: szétválaszthatjuk az adatszerkezet rekurzív bejárását az elvégzendő műveletektől

```
defmodule L do # Egyszeresen láncolt lista bejárás
  def map([], _f), do: []
  def map([hd|tl], f), do: [f.(hd)|map(tl, f)]
end
```

Magasabb rendű függvények

- A függvény is érték: változóhoz köthető, adatstruktúra eleme lehet, függvény eredménye lehet, paraméterként átadható stb.
  Azaz: a függvény is ún. *first class citizen, teljes jogú polgár*

```
> L.map [1, 2, 3], fn(x) -> 2 * x end
[2, 4, 6]
```

- Egyre gyakoribb az objektumorientált nyelvekben is
  - Java: `List.of(1, 2, 3).stream().map(x -> 2 * x).toList()`
  - C#: `new List<int>{1, 2, 3}.Select(x => 2 * x)`
- Ciklusok helyett: szétválaszthatjuk az adatszerkezet rekurzív bejárását az elvégzendő műveletektől

```
defmodule L do # Egyszeresen láncolt lista bejárás
  def map([], _f), do: []
  def map([hd|tl], f), do: [f.(hd)|map(tl, f)]
end
defmodule T do # Bináris fa bejárás
  def map(nil, _f), do: nil
  def map({x, left, right}, f), do: {f.(x), map(left, f), map(right, f)}
end
```

Dinamikus típusok

Az Elixir

- *erősen típusos:* minden értéknek pontosan egy futásidejű típusa van
- *dinamikus típusellenőrzéssel:* a típusokat nem kötelező megadni a kódban, egy változóhoz nem feltétlenül csak egy típus tartozik

Dinamikus típusok

Az Elixir

- *erősen típusos:* minden értéknek pontosan egy futásidejű típusa van
- *dinamikus típusellenőrzéssel:* a típusokat nem kötelező megadni a kódban, egy változóhoz nem feltétlenül csak egy típus tartozik
- DE: típus specifikációk megadhatók dokumentációs céllal
  - *@spec tl(xs :: [any()]) :: ts :: [any()] | nil*
    „A `tl` függvény argumentuma egy `xs` tetszőleges elemű lista, visszatérési értéke (`ts`) egy szintén tetszőleges elemű lista vagy `nil`.”

Dinamikus típusok

Az Elixir

- *erősen típusos:* minden értéknek pontosan egy futásidejű típusa van
- *dinamikus típusellenőrzéssel:* a típusokat nem kötelező megadni a kódban, egy változóhoz nem feltétlenül csak egy típus tartozik
- DE: típus specifikációk megadhatók dokumentációs céllal
  - *@spec tl(xs :: [any()]) :: ts :: [any()] | nil*
    „A `tl` függvény argumentuma egy `xs` tetszőleges elemű lista, visszatérési értéke (`ts`) egy szintén tetszőleges elemű lista vagy `nil`.”
  - *@spec nth(xs :: [any()], n :: integer()) :: r :: any() | nil*

Dinamikus típusok

Az Elixir

- *erősen típusos:* minden értéknek pontosan egy futásidejű típusa van
- *dinamikus típusellenőrzéssel:* a típusokat nem kötelező megadni a kódban, egy változóhoz nem feltétlenül csak egy típus tartozik
- DE: típus specifikációk megadhatók dokumentációs céllal
  - *@spec tl(xs :: [any()]) :: ts :: [any()] | nil*
    „A `tl` függvény argumentuma egy `xs` tetszőleges elemű lista, visszatérési értéke (`ts`) egy szintén tetszőleges elemű lista vagy `nil`.”
  - *@spec nth(xs :: [any()], n :: integer()) :: r :: any() | nil*
    „Az `nth` függvény argumentumai egy `xs` tetszőleges elemű lista és egy `n` egész szám, visszatérési értéke (`r`) vagy egy tetszőleges típusú érték vagy `nil`.”

Folyamatok üzenetekkel kommunikálnak

- Nincs folyamatok (processzek) között megosztott memória
- Kommunikáció üzenetküldés segítségével (aktor modell)

Folyamatok üzenetekkel kommunikálnak

- Nincs folyamatok (processzek) között megosztott memória
- Kommunikáció üzenetküldés segítségével (aktor modell)

```
worker = spawn(fn -> # Új folyamat indítása
  receive do # Várakozás üzenetre
    {:hello, from} -> # Válasz küldése from felé
      send(from, {:reply, "Hello from the worker process!"})
  end
end)
```

Folyamatok üzenetekkel kommunikálnak

- Nincs folyamatok (processzek) között megosztott memória
- Kommunikáció üzenetküldés segítségével (aktor modell)

```
worker = spawn(fn -> # Új folyamat indítása
  receive do # Várakozás üzenetre
    {:hello, from} -> # Válasz küldése from felé
      send(from, {:reply, "Hello from the worker process!"})
  end
end)
send(worker, {:hello, self()}) # Üzenet küldése válaszcímmel
```

Folyamatok üzenetekkel kommunikálnak

- Nincs folyamatok (processzek) között megosztott memória
- Kommunikáció üzenetküldés segítségével (aktor modell)

```
worker = spawn(fn -> # Új folyamat indítása
  receive do # Várakozás üzenetre
    {:hello, from} -> # Válasz küldése from felé
      send(from, {:reply, "Hello from the worker process!"})
  end
end)
send(worker, {:hello, self()}) # Üzenet küldése válaszcímmel
receive do # Várakozás a worker folyamat válaszára
  {:reply, message} -> # Üzenet fogadása mintaillesztéssel
    IO.puts(message)
end
```

Folyamatok üzenetekkel kommunikálnak

- Nincs folyamatok (processzek) között megosztott memória
- Kommunikáció üzenetküldés segítségével (aktor modell)

```
worker = spawn(fn -> # Új folyamat indítása
  receive do # Várakozás üzenetre
    {:hello, from} -> # Válasz küldése from felé
      send(from, {:reply, "Hello from the worker process!"})
  end
end)
send(worker, {:hello, self()}) # Üzenet küldése válaszcímmel
receive do # Várakozás a worker folyamat válaszára
  {:reply, message} -> # Üzenet fogadása mintaillesztéssel
    IO.puts(message)
end
```

- Válasz egymásutáni üzenetekre rekurzív függvények segítségével

Erlang és Elixir

- Erlang Open Telecom Platform (OTP): az Ericsson által fejlesztett nyílt forráskódú rendszer masszívan párhuzamos, elosztott, megbízható alkalmazások fejlesztésére
  - Kihasználja az üzenetküldést és azt, hogy nincs megosztott memória vagy változó módosítás
  - Akár függvények is átküldhetők a szerverek között

Erlang és Elixir

- Erlang Open Telecom Platform (OTP): az Ericsson által fejlesztett nyílt forráskódú rendszer masszívan párhuzamos, elosztott, megbízható alkalmazások fejlesztésére
  - Kihasználja az üzenetküldést és azt, hogy nincs megosztott memória vagy változó módosítás
  - Akár függvények is átküldhetők a szerverek között
- Erlang: az OTP platform eredeti funkcionális programozási nyelve, Prologhoz hasonló szintaxis

Erlang és Elixir

- Erlang Open Telecom Platform (OTP): az Ericsson által fejlesztett nyílt forráskódú rendszer masszívan párhuzamos, elosztott, megbízható alkalmazások fejlesztésére
  - Kihasználja az üzenetküldést és azt, hogy nincs megosztott memória vagy változó módosítás
  - Akár függvények is átküldhetők a szerverek között
- Erlang: az OTP platform eredeti funkcionális programozási nyelve, Prologhoz hasonló szintaxis
- BEAM: az Erlang/OTP virtuális gépe

Erlang és Elixir

- Erlang Open Telecom Platform (OTP): az Ericsson által fejlesztett nyílt forráskódú rendszer masszívan párhuzamos, elosztott, megbízható alkalmazások fejlesztésére
  - Kihasználja az üzenetküldést és azt, hogy nincs megosztott memória vagy változó módosítás
  - Akár függvények is átküldhetők a szerverek között
- Erlang: az OTP platform eredeti funkcionális programozási nyelve, Prologhoz hasonló szintaxis
- BEAM: az Erlang/OTP virtuális gépe
- Elixir: funkcionális programozási nyelv a BEAM platforma, modern (Ruby-ra hasonlító) szintaxissal

Erlang és Elixir

- Erlang Open Telecom Platform (OTP): az Ericsson által fejlesztett nyílt forráskódú rendszer masszívan párhuzamos, elosztott, megbízható alkalmazások fejlesztésére
  - Kihasználja az üzenetküldést és azt, hogy nincs megosztott memória vagy változó módosítás
  - Akár függvények is átküldhetők a szerverek között
- Erlang: az OTP platform eredeti funkcionális programozási nyelve, Prologhoz hasonló szintaxis
- BEAM: az Erlang/OTP virtuális gépe
- Elixir: funkcionális programozási nyelv a BEAM platforma, modern (Ruby-ra hasonlító) szintaxissal

$$\text{BEAM} : \text{Erlang} : \text{Elixir} \approx \text{JVM} : \text{Java} : \text{Kotlin}$$

$$\text{Jupyter} \approx \text{Livebook}$$

Tartalom

Elixir: használat, telepítés

- Elixir homokozó: `https://dps.iit.bme.hu/edux/`.[2]
- Az Elixir előtt a megfelelő verziójú Erlangot is telepíteni kell.
- Telepítés: `https://elixir-lang.org/install.html`

- Összetettebb esetekhez: különféle csomagkezelők: Debian, Ubuntu, Windows, macOS
  - **asdf** verziókezelő (több verzió telepíthető vele párhuzamosan).
  - **mise** verziókezelő (*asdf*-en alapul, ugyancsak több verziót tud kezelni): `https://mise.jdx.dev/getting-started.html`.
    Az Erlangot a **mise** alapból ismeri (lásd: *Plugins*, Core Plugins), az Elixir telepítése itt van leírva:
    `https://github.com/mise-plugins/mise-elixir`.

[2] Köszönet Gergely Viktornak az EDUX-ért! Forrás: `https://github.com/vikger/edux/`

Livebook for Elixir telepítése

- A *Livebook* a Python Jupyterhez hasonló notebook. Telepítéséről itt lehet olvasni: `https://livebook.dev/#install`.
- Macre és Windowsra telepítőkkel lehet fölrakni.
- Linuxon csak kicsit bonyolultabb (hála az Elixir *escript* funkciójának), a leírás itt található: `https://github.com/livebook-dev/livebook#direct-installation-with-elixir`.
- Ha az Elixirt és Erlangot a *mise*-zel telepítettük, akkor az Erlang függőségeit, amelyek az említett szakaszban vannak felsorolva (`inets`, `os_mon`, `runtime_tools`, `ssl`, `xmerl`), nem az operációs rendszer, hanem az Erlang már említett *rebar* csomagkezelőjével kell telepíteni:
  - `rebar3 local install inets` stb.
- Futtatás csomagkezelőből telepítés után: `https://github.com/livebook-dev/livebook#escript`.

Ha nem akarunk a telepítésekkel bajlódni, akkor használhatjuk a dockerizált Elixirt és Livebookot. Részletek a következő dián.

Elixir: használat Docker konténerekkel

- Elixir Docker: `https://elixir-lang.org/install.html#docker`
  - Run iex (interactive mode): `docker run -it --rm elixir`
  - Run elixir (compiler & iex): `docker run -it --rm elixir bash`
- Elixir notebook, azaz *Livebook for Elixir:* `https://livebook.dev/`
  - Docker: `https://github.com/livebook-dev/livebook#docker`
  - Lokális tárhely a `-v $(pwd):/data` opcióval megadott mappában; `$(pwd)` használata esetén abban a mappában, ahol a `docker ...` parancsot kiadjuk.
    Tulajdonosi jogok beállítása a `-u $(id -u):$(id -g)` opcióval.

```
docker run -p 8080:8080 -p 8081:8081 --rm --pull always \
-u $(id -u):$(id -g) -v $(pwd):/data ghcr.io/livebook-dev/livebook
```

  - Már letöltött konténer gyorsindítása frissítés nélkül:

```
docker run -p 8080:8080 -p 8081:8081 --rm \
-u $(id -u):$(id -g) -v $(pwd):/data ghcr.io/livebook-dev/livebook
```

Elixir: szövegszerkesztők

- Editors for Elixir: `https://github.com/elixir-editors`
  - vim-elixir for VIM:
    `https://github.com/elixir-editors/vim-elixir`
  - emacs-elixir for Emacs:
    `https://github.com/elixir-editors/emacs-elixir`
  - language-elixir for Atom:
    `https://github.com/elixir-editors/language-elixir`
  - Visual Studio Code with ElixirLS:
    `https://thinkingelixir.com/elixir-in-vs-code/`
  - IntelliJIdea with intellij-elixir plugin:
    `https://github.com/KronicDeth/intellij-elixir`

Elixir felvezető tutorial Livebookban

Az Elixir az előadásban szereplő jellegzetességeinek további bemutatásához *Livebook* segítségével készítettünk egy interaktív bemutatót.
A Livebook-példák interaktív Markdown formátumban innen tölthetők le:

- `https://dp.iit.bme.hu/dp26a/gy/dp26a-fp1gyfel.livemd`.

Mivel egyetlen programozási nyelvet sem lehet lineárisan, minden részletre kitérve ismertetni, megtanulni, ezért a fontos dolgokra később is visszatérünk.

A következő diákon az Elixir interaktív használatát mutatjuk be kisebb példákon.

Tartalom

Interactive Elixir (REPL: read-eval-print loop)

```
$ iex
Erlang/OTP ...
Interactive Elixir ...
  press Ctrl+C to exit
  (type h() ENTER for help)
iex(1)>
```

Interactive Elixir (REPL: read-eval-print loop)

```
$ iex
Erlang/OTP ...
Interactive Elixir ...
  press Ctrl+C to exit
  (type h() ENTER for help)
iex(1)> 3.2 + 2.1 * 2
7.4
```

Interactive Elixir (REPL: read-eval-print loop)

```
$ iex
Erlang/OTP ...
Interactive Elixir ...
  press Ctrl+C to exit
  (type h() ENTER for help)
iex(1)> 3.2 + 2.1 * 2
7.4
iex(2)> :atom
:atom
```

Interactive Elixir (REPL: read-eval-print loop)

```
$ iex
Erlang/OTP ...
Interactive Elixir ...
  press Ctrl+C to exit
  (type h() ENTER for help)
iex(1)> 3.2 + 2.1 * 2
7.4
iex(2)> :atom
:atom
iex(3)> Atom
Atom
```

Interactive Elixir (REPL: read-eval-print loop)

```
$ iex
Erlang/OTP ...
Interactive Elixir ...
  press Ctrl+C to exit
  (type h() ENTER for help)
iex(1)> 3.2 + 2.1 * 2
7.4
iex(2)> :atom
:atom
iex(3)> Atom
Atom
iex(4)> "string"
"string"
```

Interactive Elixir (REPL: read-eval-print loop)

```
$ iex
Erlang/OTP ...
Interactive Elixir ...
  press Ctrl+C to exit
  (type h() ENTER for help)
iex(1)> 3.2 + 2.1 * 2
7.4
iex(2)> :atom
:atom
iex(3)> Atom
Atom
iex(4)> "string"
"string"
iex(5)> {:ennes,:%,A,:':',9.8}
{:ennes, :%, A, :":", 9.8}
```

Interactive Elixir (REPL: read-eval-print loop)

```
$ iex
Erlang/OTP ...
Interactive Elixir ...
  press Ctrl+C to exit
  (type h() ENTER for help)
iex(1)> 3.2 + 2.1 * 2
7.4
iex(2)> :atom
:atom
iex(3)> Atom
Atom
iex(4)> "string"
"string"
iex(5)> {:ennes,:%,A,:':',9.8}
{:ennes, :%, A, :":", 9.8}
iex(6)> [:lista,:%,A,:':',9.8]
[:lista, :%, A, :":", 9.8]
```

Interactive Elixir (REPL: read-eval-print loop)

```
$ iex
Erlang/OTP ...
Interactive Elixir ...
  press Ctrl+C to exit
  (type h() ENTER for help)
iex(1)> 3.2 + 2.1 * 2
7.4
iex(2)> :atom
:atom
iex(3)> Atom
Atom
iex(4)> "string"
"string"
iex(5)> {:ennes,:%,A,:':',9.8}
{:ennes, :%, A, :":", 9.8}
iex(6)> [:lista,:%,A,:':',9.8]
[:lista, :%, A, :":", 9.8]
iex(7)> i :':'
```

Interactive Elixir (REPL: read-eval-print loop)

```
$ iex
Erlang/OTP ...
Interactive Elixir ...
  press Ctrl+C to exit
  (type h() ENTER for help)
iex(1)> 3.2 + 2.1 * 2
7.4
iex(2)> :atom
:atom
iex(3)> Atom
Atom
iex(4)> "string"
"string"
iex(5)> {:ennes,:%,A,:':',9.8}
{:ennes, :%, A, :":", 9.8}
iex(6)> [:lista,:%,A,:':',9.8]
[:lista, :%, A, :":", 9.8]
iex(7)> i :':'
...Data type
    Atom...
```

Interactive Elixir (REPL: read-eval-print loop)

```
$ iex
Erlang/OTP ...
Interactive Elixir ...
  press Ctrl+C to exit
  (type h() ENTER for help)
iex(1)> 3.2 + 2.1 * 2
7.4
iex(2)> :atom
:atom
iex(3)> Atom
Atom
iex(4)> "string"
"string"
iex(5)> {:ennes,:%,A,:':',9.8}
{:ennes, :%, A, :":", 9.8}
iex(6)> [:lista,:%,A,:':',9.8]
[:lista, :%, A, :":", 9.8]
iex(7)> i :':'
...Data type
    Atom...
```

```
iex(8)> Ctrl+C
```

Interactive Elixir (REPL: read-eval-print loop)

```
$ iex
Erlang/OTP ...
Interactive Elixir ...
  press Ctrl+C to exit
  (type h() ENTER for help)
iex(1)> 3.2 + 2.1 * 2
7.4
iex(2)> :atom
:atom
iex(3)> Atom
Atom
iex(4)> "string"
"string"
iex(5)> {:ennes,:%,A,:':',9.8}
{:ennes, :%, A, :":", 9.8}
iex(6)> [:lista,:%,A,:':',9.8]
[:lista, :%, A, :":", 9.8]
iex(7)> i :':'
...Data type
    Atom...
```

```
iex(8)> Ctrl+C
BREAK: (a)bort (A)bort with dump (c)ontinue
   (p)roc info (i)nfo (l)oaded (v)ersion
   (k)ill (D)b-tables (d)istribution
```

Interactive Elixir (REPL: read-eval-print loop)

```
$ iex
Erlang/OTP ...
Interactive Elixir ...
  press Ctrl+C to exit
  (type h() ENTER for help)
iex(1)> 3.2 + 2.1 * 2
7.4
iex(2)> :atom
:atom
iex(3)> Atom
Atom
iex(4)> "string"
"string"
iex(5)> {:ennes,:%,A,:':',9.8}
{:ennes, :%, A, :":", 9.8}
iex(6)> [:lista,:%,A,:':',9.8]
[:lista, :%, A, :":", 9.8]
iex(7)> i :':'
...Data type
    Atom...
```

```
iex(8)> Ctrl+C
BREAK: (a)bort (A)bort with dump (c)ontinue
   (p)roc info (i)nfo (l)oaded (v)ersion
   (k)ill (D)b-tables (d)istribution
Ctrl+C
$
```

Interactive Elixir (REPL: read-eval-print loop)

```
$ iex
Erlang/OTP ...
Interactive Elixir ...
  press Ctrl+C to exit
  (type h() ENTER for help)
iex(1)> 3.2 + 2.1 * 2
7.4
iex(2)> :atom
:atom
iex(3)> Atom
Atom
iex(4)> "string"
"string"
iex(5)> {:ennes,:%,A,:':',9.8}
{:ennes, :%, A, :":", 9.8}
iex(6)> [:lista,:%,A,:':',9.8]
[:lista, :%, A, :":", 9.8]
iex(7)> i :':'
...Data type
    Atom...
```

```
iex(8)> Ctrl+C
BREAK: (a)bort (A)bort with dump (c)ontinue
   (p)roc info (i)nfo (l)oaded (v)ersion
   (k)ill (D)b-tables (d)istribution
Ctrl+C
$

iex(8>) Ctrl+G
User switch command
-->
```

Interactive Elixir (REPL: read-eval-print loop)

```
$ iex
Erlang/OTP ...
Interactive Elixir ...
  press Ctrl+C to exit
  (type h() ENTER for help)
iex(1)> 3.2 + 2.1 * 2
7.4
iex(2)> :atom
:atom
iex(3)> Atom
Atom
iex(4)> "string"
"string"
iex(5)> {:ennes,:%,A,:':',9.8}
{:ennes, :%, A, :":", 9.8}
iex(6)> [:lista,:%,A,:':',9.8]
[:lista, :%, A, :":", 9.8]
iex(7)> i :':'
...Data type
    Atom...
```

```
iex(8)> Ctrl+C
BREAK: (a)bort (A)bort with dump (c)ontinue
   (p)roc info (i)nfo (l)oaded (v)ersion
   (k)ill (D)b-tables (d)istribution
Ctrl+C
$

iex(8>) Ctrl+G
User switch command
--> h
  c [nn]              - connect to job
  i [nn]              - interrupt job
  k [nn]              - kill job
  j                   - list all jobs
  s [shell]           - start local shell
  r [node [shell]]    - start remote shell
  q                   - quit erlang
  ? | h               - this message
  -->
```

Interactive Elixir (REPL: read-eval-print loop)

```
$ iex
Erlang/OTP ...
Interactive Elixir ...
  press Ctrl+C to exit
  (type h() ENTER for help)
iex(1)> 3.2 + 2.1 * 2
7.4
iex(2)> :atom
:atom
iex(3)> Atom
Atom
iex(4)> "string"
"string"
iex(5)> {:ennes,:%,A,:':',9.8}
{:ennes, :%, A, :":", 9.8}
iex(6)> [:lista,:%,A,:':',9.8]
[:lista, :%, A, :":", 9.8]
iex(7)> i :':'
...Data type
    Atom...
```

```
iex(8)> Ctrl+C
BREAK: (a)bort (A)bort with dump (c)ontinue
   (p)roc info (i)nfo (l)oaded (v)ersion
   (k)ill (D)b-tables (d)istribution
Ctrl+C
$

iex(8>) Ctrl+G
User switch command
--> h
  c [nn]              - connect to job
  i [nn]              - interrupt job
  k [nn]              - kill job
  j                   - list all jobs
  s [shell]           - start local shell
  r [node [shell]]    - start remote shell
  q                   - quit erlang
  ? | h               - this message
  --> q
$
```

Interactive Elixir (IEx): parancsok

```
iex(1)> h().
```

Interactive Elixir (IEx): parancsok

```
iex(1)> h().
Welcome to Interactive Elixir.
...
c/1             - compiles a file
c/2             - compiles a file and writes bytecode to the given path
cd/1            - changes the current directory
clear/0         - clears the screen
exports/1       - shows all exports (functions + macros) in a module
h/1             - prints help for the given module, function or macro
i/0             - prints information about the last value
i/1             - prints information about the given term
ls/0            - lists the contents of the current directory
ls/1            - lists the contents of the specified directory
pwd/0           - prints the current working directory
r/1             - recompiles the given module's source file
v/0             - retrieves the last value from the history
v/1             - retrieves the nth value from the history
...
To learn more about IEx as a whole, type h(IEx).
```

Saját program fordítása, futtatása

fpea.ex – Faktoriális

```
defmodule Fpea do
# Fájlnév csupa kisbetűvel, szavak között aláhúzás (snake_case)
# Modulnév egybe, szavak nagy kezdőbetűvel (BumpyCase, CamelCase)
  @spec fac(n::integer) :: f::integer  # Típusspecifikáció
  # f = n! (azaz f az n faktoriálisa)   # Fejkomment
  def fac(0), do: 1                     # ha az n=0 mintaillesztés sikeres
  def fac(n), do: n * fac(n-1)          # ha az n=0 mintaillesztés sikertelen
end
```

```
iex(1)>
```

Saját program fordítása, futtatása

fpea.ex – Faktoriális

```
defmodule Fpea do
# Fájlnév csupa kisbetűvel, szavak között aláhúzás (snake_case)
# Modulnév egybe, szavak nagy kezdőbetűvel (BumpyCase, CamelCase)
  @spec fac(n::integer) :: f::integer  # Típusspecifikáció
  # f = n! (azaz f az n faktoriálisa)   # Fejkomment
  def fac(0), do: 1                     # ha az n=0 mintaillesztés sikeres
  def fac(n), do: n * fac(n-1)          # ha az n=0 mintaillesztés sikertelen
end
```

```
iex(1)> c "fpea.ex"   # fordítás
[Fpea]
iex(2)>
```

Saját program fordítása, futtatása

fpea.ex – Faktoriális

```
defmodule Fpea do
# Fájlnév csupa kisbetűvel, szavak között aláhúzás (snake_case)
# Modulnév egybe, szavak nagy kezdőbetűvel (BumpyCase, CamelCase)
  @spec fac(n::integer) :: f::integer  # Típusspecifikáció
  # f = n! (azaz f az n faktoriálisa)   # Fejkomment
  def fac(0), do: 1                     # ha az n=0 mintaillesztés sikeres
  def fac(n), do: n * fac(n-1)          # ha az n=0 mintaillesztés sikertelen
end
```

```
iex(1)> c "fpea.ex"   # fordítás
[Fpea]
iex(2)> Fpea.fac(5)   # futtatás
120
iex(3)>
```

Saját program fordítása, futtatása

fpea.ex – Faktoriális

```
defmodule Fpea do
# Fájlnév csupa kisbetűvel, szavak között aláhúzás (snake_case)
# Modulnév egybe, szavak nagy kezdőbetűvel (BumpyCase, CamelCase)
  @spec fac(n::integer) :: f::integer  # Típusspecifikáció
  # f = n! (azaz f az n faktoriálisa)   # Fejkomment
  def fac(0), do: 1                     # ha az n=0 mintaillesztés sikeres
  def fac(n), do: n * fac(n-1)          # ha az n=0 mintaillesztés sikertelen
end
```

```
iex(1)> c "fpea.ex"   # fordítás
[Fpea]
iex(2)> Fpea.fac(5)   # futtatás
120
iex(3)> fac(5)          # a modulnevet ki kell írni
** (CompileError) iex:3: undefined function fac/1
iex(4)>
```

Saját program fordítása, futtatása

fpea.ex – Faktoriális

```
defmodule Fpea do
# Fájlnév csupa kisbetűvel, szavak között aláhúzás (snake_case)
# Modulnév egybe, szavak nagy kezdőbetűvel (BumpyCase, CamelCase)
  @spec fac(n::integer) :: f::integer  # Típusspecifikáció
  # f = n! (azaz f az n faktoriálisa)   # Fejkomment
  def fac(0), do: 1                     # ha az n=0 mintaillesztés sikeres
  def fac(n), do: n * fac(n-1)          # ha az n=0 mintaillesztés sikertelen
end
```

```
iex(1)> c "fpea.ex"   # fordítás
[Fpea]
iex(2)> Fpea.fac(5)   # futtatás
120
iex(3)> fac(5)          # a modulnevet ki kell írni
** (CompileError) iex:3: undefined function fac/1
iex(4)> Fpea.fac 5    # argumentum körül a zárójel sokszor elhagyható
120
```

Elixir docker – IEx

```
# tárhely a docker konténerben
$ docker run -it --rm -w /home elixir
```

Elixir docker – IEx

```
# tárhely a docker konténerben
$ docker run -it --rm -w /home elixir
Erlang/OTP ...
Interactive Elixir ...
iex(1)>
```

Elixir docker – IEx

```
# tárhely a docker konténerben
$ docker run -it --rm -w /home elixir
Erlang/OTP ...
Interactive Elixir ...
iex(1)> pwd
/home
iex(2)>
```

Elixir docker – IEx

```
# tárhely a docker konténerben
$ docker run -it --rm -w /home elixir
Erlang/OTP ...
Interactive Elixir ...
iex(1)> pwd
/home
iex(2)> Enum.map 1..5, fn(x) -> -x end
[-1, -2, -3, -4, -5]
iex(3)>
```

Elixir docker – IEx

```
# tárhely a docker konténerben
$ docker run -it --rm -w /home elixir
Erlang/OTP ...
Interactive Elixir ...
iex(1)> pwd
/home
iex(2)> Enum.map 1..5, fn(x) -> -x end
[-1, -2, -3, -4, -5]
iex(3)> Ctrl+C kétszer

# tárhely a hoszton
$
```

Elixir docker – IEx

```
# tárhely a docker konténerben
$ docker run -it --rm -w /home elixir
Erlang/OTP ...
Interactive Elixir ...
iex(1)> pwd
/home
iex(2)> Enum.map 1..5, fn(x) -> -x end
[-1, -2, -3, -4, -5]
iex(3)> Ctrl+C kétszer

# tárhely a hoszton
$ docker run -it --rm -v "$PWD":/home -w /home elixir
iex(1)>
```

Elixir docker – IEx

```
# tárhely a docker konténerben
$ docker run -it --rm -w /home elixir
Erlang/OTP ...
Interactive Elixir ...
iex(1)> pwd
/home
iex(2)> Enum.map 1..5, fn(x) -> -x end
[-1, -2, -3, -4, -5]
iex(3)> Ctrl+C kétszer

# tárhely a hoszton
$ docker run -it --rm -v "$PWD":/home -w /home elixir
iex(1)>ls
fpea.ex      fpea.exs      ...
iex(2)>
```

Elixir docker – IEx

```
# tárhely a docker konténerben
$ docker run -it --rm -w /home elixir
Erlang/OTP ...
Interactive Elixir ...
iex(1)> pwd
/home
iex(2)> Enum.map 1..5, fn(x) -> -x end
[-1, -2, -3, -4, -5]
iex(3)> Ctrl+C kétszer

# tárhely a hoszton
$ docker run -it --rm -v "$PWD":/home -w /home elixir
iex(1)>ls
fpea.ex      fpea.exs      ...
iex(2)> c "fpea.ex"
[Fpea]
iex(3)>
```

Elixir docker – IEx

```
# tárhely a docker konténerben
$ docker run -it --rm -w /home elixir
Erlang/OTP ...
Interactive Elixir ...
iex(1)> pwd
/home
iex(2)> Enum.map 1..5, fn(x) -> -x end
[-1, -2, -3, -4, -5]
iex(3)> Ctrl+C kétszer

# tárhely a hoszton
$ docker run -it --rm -v "$PWD":/home -w /home elixir
iex(1)>ls
fpea.ex      fpea.exs      ...
iex(2)> c "fpea.ex"
[Fpea]
iex(3)> exports Fpea
fac/1
4>
```

Elixir docker – IEx

```
# tárhely a docker konténerben
$ docker run -it --rm -w /home elixir
Erlang/OTP ...
Interactive Elixir ...
iex(1)> pwd
/home
iex(2)> Enum.map 1..5, fn(x) -> -x end
[-1, -2, -3, -4, -5]
iex(3)> Ctrl+C kétszer

# tárhely a hoszton
$ docker run -it --rm -v "$PWD":/home -w /home elixir
iex(1)>ls
fpea.ex      fpea.exs      ...
iex(2)> c "fpea.ex"
[Fpea]
iex(3)> exports Fpea
fac/1
4> Fpea.fac 5
120
```

Elixir docker – bash

```
# tárhely a hoszton
$ docker run -it --rm -v "$PWD":/home -w /home elixir /bin/bash
root@...:/home#
#
```

Elixir docker – bash

```
# tárhely a hoszton
$ docker run -it --rm -v "$PWD":/home -w /home elixir /bin/bash
root@...:/home#
# ls *.ex *.exs
fpea.ex      fpea.exs
#
```

Elixir docker – bash

```
# tárhely a hoszton
$ docker run -it --rm -v "$PWD":/home -w /home elixir /bin/bash
root@...:/home#
# ls *.ex *.exs
fpea.ex      fpea.exs
# iex fpea.ex
```

Elixir docker – bash

```
# tárhely a hoszton
$ docker run -it --rm -v "$PWD":/home -w /home elixir /bin/bash
root@...:/home#
# ls *.ex *.exs
fpea.ex      fpea.exs
# iex fpea.ex
Erlang/OTP ...
Interactive Elixir ...
iex(1)>
```

Elixir docker – bash

```
# tárhely a hoszton
$ docker run -it --rm -v "$PWD":/home -w /home elixir /bin/bash
root@...:/home#
# ls *.ex *.exs
fpea.ex      fpea.exs
# iex fpea.ex
Erlang/OTP ...
Interactive Elixir ...
iex(1)> Fpea.fac 5
120
iex(2)>
```

Elixir docker – bash

```
# tárhely a hoszton
$ docker run -it --rm -v "$PWD":/home -w /home elixir /bin/bash
root@...:/home#
# ls *.ex *.exs
fpea.ex      fpea.exs
# iex fpea.ex
Erlang/OTP ...
Interactive Elixir ...
iex(1)> Fpea.fac 5
120
iex(2)> Ctrl+C kétszer
$root@...:/home#
```

Elixir docker – bash

```
# tárhely a hoszton
$ docker run -it --rm -v "$PWD":/home -w /home elixir /bin/bash
root@...:/home#
# ls *.ex *.exs
fpea.ex      fpea.exs
# iex fpea.ex
Erlang/OTP ...
Interactive Elixir ...
iex(1)> Fpea.fac 5
120
iex(2)> Ctrl+C kétszer
$root@...:/home# ^D
```

Elixir docker – bash

```
# tárhely a hoszton
$ docker run -it --rm -v "$PWD":/home -w /home elixir /bin/bash
root@...:/home#
# ls *.ex *.exs
fpea.ex      fpea.exs
# iex fpea.ex
Erlang/OTP ...
Interactive Elixir ...
iex(1)> Fpea.fac 5
120
iex(2)> Ctrl+C kétszer
$root@...:/home# ^D exit
$
```

Tartalom

Projektszervezés Elixirben 1: `mix`

Foglalkozzunk egy kicsit az Elixir projektek szervezésével.

- Az Elixirhez sokféle modul van, a gyakran használtak (pl. `Kernel`, `Enum`, `List`, `String`) az Elixir-alapcsomag része, a többit (pl. `Benchee`) utólag kell telepíteni, ha és amikor szükség van rájuk.
- Magának a fordítónak (`elixir`, `elixirc`, `iex`) is, a moduloknak is több verziójuk van, az újabb verziók nem mindig kompatibilisek a korábbiakkal: a függőségeket kezelni kell.
- Általában egy saját projekt is több modulból áll, plusz a teszteléshez használt adatokból, segédprogramokból – célszerű ezeket is jól áttekinthetően, rendben tartani.
- Ahhoz, hogy az Elixirhez kidolgozott segédeszközöket használni tudjuk, be kell tartani a konvenciókat – nemcsak a névadásra, hanem például a fájlokat tároló mappák szerkezetére vonatkozóakat is.
- *Elixir projektek kezelésére* készült a `mix`. A `mix` az Elixir-csomag része.

  `https://elixir-lang.org/getting-started/mix-otp/introduction-to-mix.html`

Projektszervezés Elixirben 2: `mix`

Indítsuk el `mix`-et a `~/tmp/` mappában:

```
  ...$ cd ~/tmp
  ~/tmp$ mix --help
Mix is a build tool for Elixir

Usage: mix [task]

Examples:

    mix             - Invokes the default task (mix run) in a project
    mix new PATH    - Creates a new Elixir project at the given path
    mix help        - Lists all available tasks
    mix help TASK   - Prints documentation for a given task

The --help and --version options can be given instead of a task for
usage and versioning information.
```

Használhatjuk a dokkeres Elixirt is:

```
  ~$ docker run -it --rm -v "$PWD":/home -w /home elixir /bin/bash
  .../home#
```

Projektszervezés Elixirben 3: `mix`

Hozzunk létre egy új projektet egy új mappában. A projekt és a mappa neve legyen `fp`. Az `fp/lib` mappában is létrejön egy fájl `fp.ex` néven, benne az *Fp* modul-sablonnal – ez lenne a neve a `--module` opció nélkül is.

```
# mix new fp --module Fp
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/fp.ex
* creating test
* creating test/test_helper.exs
* creating test/fp_test.exs

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:
    cd fp
    mix test
Run "mix help" for more commands.

# ls -F fp
lib/  mix.exs  README.md  test/
```

Projektszervezés Elixirben 4: `mix`

Nézzük, mi van a `mix.exs` fájlban[3]:

```
~/tmp$ cd fp
~/tmp/fp$ cat mix.exs
defmodule Fp.MixProject do
  use Mix.Project
  def project do
    [
      app: :fp,
      version: "0.1.0",
      elixir: "~> 1.18",
      start_permanent: Mix.env() == :prod,
      deps: deps()
    ]
  end
  # Run "mix help compile.app" to learn about applications.
  def application do
    [
      extra_applications: [:logger]
    ]
  end
```

(Folytatás a következő dián.)

[3] Mi más lenne, mint Elixir-kód. :-)

Projektszervezés Elixirben 5: `mix`

(Az előző dia folytatása.)

```
 # Run "mix help deps" to learn about dependencies.
 defp deps do
  [
   # {:dep_from_hexpm, "~> 0.3.0"},
   # {:dep_from_git, git: "https://github.com/elixir-lang/my_dep.git", tag: ...}
  ]
 end
end
```

- `mix.exs` két publikus (`def`) és egy privát (`defp`) függvényt definiál.
- `project` a projektkonfigurációról tárol adatokat, `appication`-nel pedig egy applikációs fájlt lehet generálni – ezek részleteibe nem megyünk bele.
- A `deps` privát függvény törzsében kell leírni a függőségeket, megadni a kívánt modulok nevét és paramétereit.
- Egy új modult fogunk megadni, a `Benchee`-t, ami – ahogy a neve is sugallja – benchmarkingra használható: `https://github.com/bencheeorg/benchee`
- A következő dián a `mix.exs` fájl végét láthatjuk ismét az új függőséggel.

Projektszervezés Elixirben 6: `mix`

```
# Run "mix help deps" to learn about dependencies.
defp deps do
 [
  {:benchee, "~> 1.0", only: :dev},
  # {:dep_from_hexpm, "~> 0.3.0"},
  # {:dep_from_git, git: "https://github.com/elixir-lang/my_dep.git", tag: ...}
 ]
end
```

Telepítsük és fordítsuk le az új modult és függőségeit![4]

```
~/tmp/fp$ mix do deps.get + deps.compile
Resolving Hex dependencies...
Resolution completed in 0.051s
New:
  benchee 1.4.0
  deep_merge 1.0.0
  statistex 1.1.0
...
Compiling ...
```

[4] Az új modulok az adott projekt részei lesznek, lokálisak, nem globálisak.

Egészlista elemeinek összege: `lib/sum.ex`

Alább látható egy egészlisták összegét kiszámoló függvény három változatban. `sum1` első, `sum2` második klóza illeszkedik az üres listára, `sum3` pedig egy jobbrekurzív segédfüggvényt hív meg. Van-e különbség a hatékonyságukban?

```
defmodule Sum do
  def sum1([]), do: 0
  def sum1([x|xs]), do: x + sum1(xs)

  def sum2([x|xs]), do: x + sum2(xs)
  def sum2([]), do: 0

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

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

# A fájl végére írt kifejezéseket az iex automatikusan ki fogja értékelni
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()
```

Fordítás, futtatás `mix`-szel: `mix compile`; `iex -S mix`

- Fordítani a `mix compile`-lal lehet, a `_build/dev/lib/fp/ebin/` mappába kerül a lefordított fájl, pl. a `sum.ex` esetében `Elixir.Sum.beam` néven.
- `iex` indítása a `mix` projekt konfigurációjával és függőségeivel:
  (lásd `mix help`, `elixir --help`)
  ```
  iex -S mix # Starts IEx and runs the default task
  ```
- Betölteni, újratölteni egyszerű a programunkat az `r` parancssal (korrektebben: az `r` helper funkcióval):
  ```
  iex> r Sum
  ```
- A `Sum` modulban definiált függvények meghívása:
  ```
  iex> Sum.sum1 [1,2,3,4,5]
  15
  ```
- A modult záró `end` után lehetnek olyan függvényhívások, melyeket az `iex` betöltéskor kiértékel; az `IO.inspect` ezek eredményét kiírja, pl.
  ```
  1..1000 |> Range.to_list() |> Sum.sum1() |> IO.inspect()
  ```
  Ha újratöltjük a programot az `r` helper függvénnyel, az eredmény megjelenik a képernyőn:
  ```
  iex> r Sum
  ...
  500500
  {:reloaded, [Sum]}
  ```

Mérések, profilozás: `benchee`

- Már láttuk, hogyan kell telepíteni a `benchee` modult.
- Most nézzünk egy példát a használatára, vizsgáljuk meg a `Sum.sum1/1`, `Sum.sum2/1`, `Sum.sum3/1` függvények működését.
- A `Benchee.run/1` függvényt kell meghívni egy fájlban, pl. a `benchee_sum.exs`-ben. Ennek egy szótár a paramétere, amiben a függvényhívásokat kell megadni egy-egy névtelen függvény törzsében:

```
Benchee.run(%{"sum1" => fn -> 1..10_000 |> Enum.to_list() |> Sum.sum1() end,
              ...
              }
            )
```

- Az elemzést a `mix run lib/benchee_sum.exs`-szel indítjuk el.
- Ha azt is szeretnénk tudni, hogy a függvényeink által meghívott függvények milyen gyakran és mennyi ideig futnak, akkor a `profile_after` opciót kell megadnunk, így:

```
Benchee.run(%{"sum1" => fn -> 1..10_000 |> Enum.to_list() |> Sum.sum1() end,
              ...
            },
            profile_after: true
            )
```

Benchmarking eredmények (részletek): `sum.ex`

- ```
  Operating System: Linux
  CPU Information: Intel(R) Core(TM) i5-8365U CPU @ 1.60GHz
  Number of Available Cores: 8
  Available memory: 38.81 GB
  Elixir 1.18.4
  Erlang 28.0.1
  JIT enabled: true

  Benchmark suite executing with the following configuration:
  warmup: 2 s
  time: 5 s
  ...
  Estimated total run time: 21 s
  ...
  Name           ips        average  deviation         median         99th %
  sum3       16.26 K       61.50 µs    ±16.25%       63.65 µs       79.93 µs
  sum2        8.87 K      112.80 µs    ±18.45%       97.88 µs      175.92 µs
  sum1        8.26 K      121.01 µs    ±18.40%      105.16 µs      187.74 µs

  Comparison:
  sum3       16.26 K (2. klóz illeszkedik az üres listára a jobbrekurzív segédfüggvényben)
  sum2        8.87 K - 1.83x slower +51.30 µs (2. klóz illeszkedik az üres listára)
  sum1        8.26 K - 1.97x slower +59.52 µs (1. klóz illeszkedik az üres listára)
  ```
  ```

Benchmarking eredmények (részletek): `sum.ex`

A `Benchee` modul használatára példát az első előadás segédanyaga tartalmaz:
dp26a-fp1ea-sum-benchee.livemd,
dp26a-fp1ea-sum-benchee.pdf.

1. gyakorlat

- Kedden 10:15 az IB027 és IE007 termekben
- Felkészüléshez ajánlott
  - Erlang, Elixir, Livebook telepítés
    (a gyakorlaton segítünk, ha valaki elakadt)
  - Elixir felvezető tutorial:
    `https://dp.iit.bme.hu/dp26a/gy/dp26a-fp1gyfel.livemd`
  - Benchee használat példa: `https://dp.iit.bme.hu/dp26a/ea/dp26a-fp1ea-sum-benchee.livemd`
- Gyakorlat anyaga:
  `https://dp.iit.bme.hu/dp26a/gy/dp26a-fp1gy.livemd`