Deklaratív programozás

Szeredi Péter[1] Hanák Péter[2]
BME Számítástudományi és Információelméleti Tanszék

[1] szeredi@cs.bme.hu

[2] phanak@edu.bme.hu

2025. ősz

Az előadók köszönetüket fejezik ki Kápolnai Richárdnak, aki 2011 és 2017 között volt a funkcionális programozás előadója, valamint Hanák Dávidnak az ETS 2001-es kifejlesztéséért és 2024-es megújításáért.

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).
- A gyakorló, valamint a házi feladatok megoldását távkonzultációval is segítjük (Teams), ha lesz rá igény.

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 kódolnia (programoznia)!
- Elvárás: hatékony (időlimit!), jól dokumentált („kommentezett”) programok.
- A programokhoz 5–10 oldalas fejlesztői dokumentáció PDF-ben.
- Az FP NHF kiírása október, az LP NHF-é november elején a honlapon.
- Az FP NHF beadása október, az LP NHF-é november végén 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”.
- 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.
- 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 lasssúúú! $\rightsquigarrow$ 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:

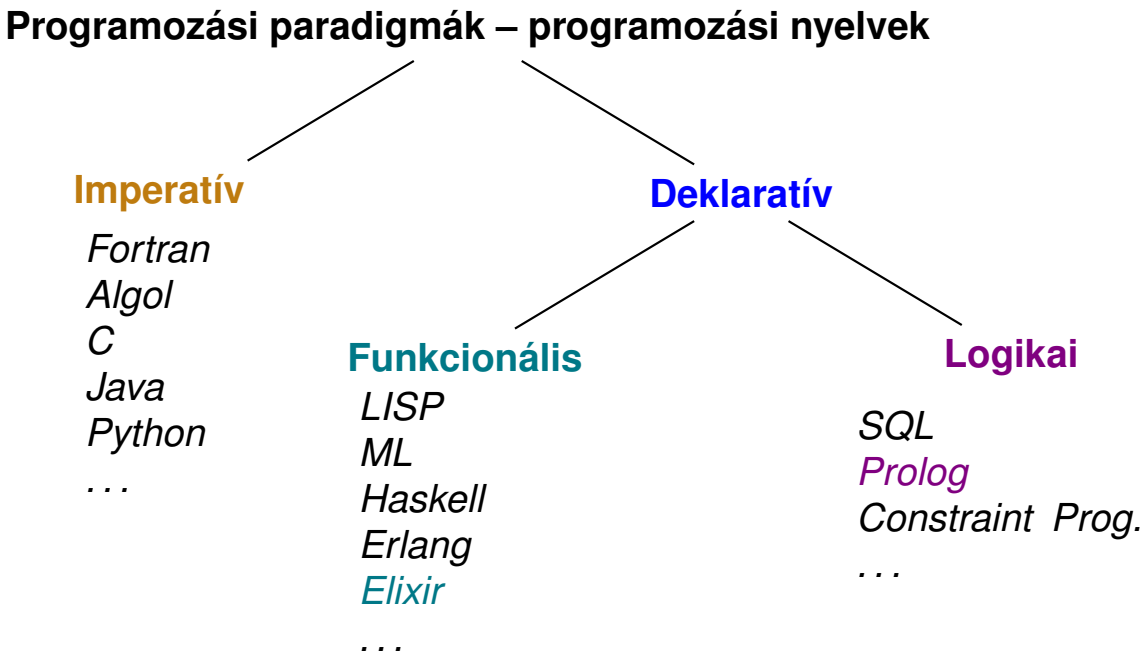
  MIT és nem HOGYAN

  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



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

1. 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 skatulyázhatók, 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).
```

- 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) ⇒ L = [1|C], app([], [2], C)  ⇒ 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
```

- Egy behelyettesítetlen változó többször is előfordulhat egy hívásban:

```
| ?- app(A, A, [1]).            ⟹   no
| ?- app(A, A, [1,1]).          ⟹   A = [1] ? ; no
```

- Az app/3 eljárás append/3 néven beépített eljárásként elérhető.

2. példa (Prolog): egy szám prím voltának ellenőrzése

- Írjunk egy `prime(P)` eljárást, amely eldönti hogy `P` prim-e.
- Az alábbi kód egy „végrehajtható specifikáció”.
- A jobboldalon angol nyelvű kommentként, a baloldalon Prolog kódként:

```
prime(P) :-                          % P is a prime if
  integer(P), P > 1,                 %   P is an integer and P > 1 and
  P1 is P-1,                         %   P1 = P-1 and
  \+                                 %   it is not the case that
    (                                %     (   there exists an I such that:
      between(2, P1, I),             %         2 =< I =< P1  and
      P mod I =:= 0                  %         P modulo I equals 0
    ).                               %     ).
```

- Az `X is Expr` egy beépített eljárás (BIP), amely az `Expr` aritmetikai kifejezést kiértékeli és az eredményt `X`-be rakja.
- Az `Expr1 =:= Expr2` BIP mindkét argumentumát (`Expr1` és `Expr2`) kiértékeli, és pontosan akkor sikerül, ha ezek az eredmények egyenlők; analóg a helyzet az `Expr1 > Expr2` esetén, stb.
- A `between(From, To, Int)` könyvtári eljárás `Int`-ben felsorolja a `From` és `To` egészek közé eső egész számokat (a határokat is beleértve).

3. példa (Prolog): adott értékű kifejezés előállítása

- Nevezzünk alapkifejezésnek egy olyan aritmetikai kifejezést amely csak a négy alapműveletet (+, -, *, /) tartalmazza
- A feladat: írjunk programot a következő feladvány megoldására: Adott számokból építsünk egy adott értékű alapkifejezést! (Feltételezhető, hogy az adott számok mind különböznek.)
- Példa: keressünk egy olyan aritmetikai kifejezést, amely az 1, 3, 4, 6 számokból áll, és értéke 24 (nehéz feladat!)
- Pontosítás:
  - A számok nem „tapaszthatók” össze hosszabb számokká
  - Mindegyik adott számot pontosan egyszer kell felhasználni, sorrendjük tetszőleges lehet
  - Nem minden alapműveletet kell felhasználni, egyfajta alapművelet többször is előfordulhat
  - Zárójelek tetszőlegesen használhatók
- Példák a fenti szabályoknak megfelelő, az 1, 3, 4, 6 számokból felépített aritmetikai kifejezésekre: $1 + 6 * (3 + 4)$, $(1 + 3)/4 + 6$

Adott értékű kifejezés előállítása – a Prolog kód

Kék/narancs színnel jelezzük a beépített/könyvtári eljárásokat

```
% Kif az L listabeli számokból  felépített, Ertek értékű kifejezés.
levelek_ertek_kif(L, Ertek, Kif) :-
    permutation(L, PL),       % PL az L levéllista permutációja,
    levelek_kif(PL, Kif),     % Kif egy PL levéllistáju alapkifejezés,
    catch(Kif =:= Ertek, _H, % Kif értéke Ertek, ha a kiértékelés hibát
          fail).              % dob (0-val osztás miatt), hiúsuljunk meg!

% Kif egy tetsz. olyan alapkifejezés, amelynek levéllistája az adott L
levelek_kif(L, Kif) :-
    L = [Kif].                % Ha L egyelemű, akkor Kif legyen az elem
levelek_kif(L, Kif) :-
    append(L1, L2, L),        % L-t bontsuk szét L1 ⊕ L2-re
    L1 \= [], L2 \= [],       % ahol sem L1, sem L2 nem []
    levelek_kif(L1, K1),      % K1 legyen egy tetsz. L1 levlistájú kif.
    levelek_kif(L2, K2),      % K2 legyen egy tetsz. L2 levlistájú kif.
    member(Op, [+,-,*,/]),    % Op legyen egy tetsz. alapművelet
    Kif =.. [Op,K1,K2].       % Kif legyen egy olyan kétargumentumú kif.
                              % melynek művelete Op, operandusai K1 és K2
```

Tartalom

Gyerekkori előzmények

- 1956/57, Áldás u. általános iskola, 2. osztály,
  alsó sor, balszélről: Simonyi Károly (később: Microsoft alapító),
  Horvai György (később akadémikus, BME rektorhelyettes 1997–2004);
  felső sor jobbszélről: Szeredi Péter

- 1963/64, Varsó, 9. osztály, első találkozás az „informatikával”: Michał Misiurewicz (lásd pl. `https://en.wikipedia.org/wiki/Misiurewicz_point`) osztálytársam épített egy „tic-tac-toe”-t, azaz 3x3-as amőbát játszó gépet (pl. `https://brainplay.com/p/tic-tac-toe`), kizárólag elemek, dugaszok, égők felhasználásával. (jó lenne rekonstruálni. . .)

Középiskolás évek

- 1964/65, Budapest, II. Rákóczi Ferenc Gimnázium, II. E osztály – Simonyi Károly/Charles ismét osztálytársam, Ural 2 számítógépprogramokat hoz be az osztályba, kilyuggatott fotófilmen.
- 1965. szeptember (gimn. III. osztály): fodrász az osztályfőnöki órán – Charles magántanuló lesz, egy tanév alatt elvégzi a III.-IV. osztályt, 17 évesen érettségit tesz, majd a dán Regnecentralen céghez megy dologozni, de az év végén nem jön haza, az USA-ba megy egyetemre.
- Vissza 1965 szeptemberéhez: beiratkozom a BME könyvtárba, kikölcsönzöm az Algol 60 programozási nyelv leírását. Még azon az őszön egy osztálytársammal együtt bemegyünk a NIM IGÜSZI *számoló*központba, ahol Pázmány Béla nagy szeretettel fogad minket.
- Elkezdek programozni az Elliott 803/B *számoló*gépen, először Autókódban, aztán gépi kódban is – „feltalálom” az asszemblert, első publikációm az érettségi évében: *Szeredi Péter. A BEEL szimbolikus programozási rendszer az ELLIOTT 803/B számológépre. NIM IGÜSZI Számítástechnikai Közlemények, 9:36-47, 1967.*

Egyetemi és korai NIM IGÜSZI-s évek

- 1967-1972: ELTE matematikus szak, mellette „heti 18 óra” munka a NIM IGÜSZI-ben: rendszerszoftverek, fordítóprogramok írása.
- Az egyetem utolsó két évében ún. egyéni képzésben mat. logikát és axiomatikus halmazelméletet tanultam, szakdolgozatom a Cohen módszerről szólt. Emellett az Algol 68 nyelv fordítási módszereivel is foglalkoztam, találtam egy hibát az Algol 68 jelentésben, így a nevem 1973-ban megjelent a *Revised Report on Algol 68* jelentésben (lásd az `szp incestuous` Google keresést).
- Megszerezvén a diplomámat, a NIM IGÜSZI-ben álltam munkába. Az Algol 68 egyik szerzője, C.H.A. Koster kidolgozott egy CDL (Compiler Description Language) nevű programozási nyelvet (`https://en.wikipedia.org/wiki/Compiler_Description_Language`), ezt használtuk a magyar EMG 830/840 számítógépek rendszerszoftvereinek fejlesztésére. Emellett gépi tételbizonyítási módszerekkel is foglalkoztunk.

A Prolog korszak kezdete

- A tételbizonyítás kapcsán találkoztunk a Prolog logikai programozási nyelvvel, amelyet 1972-73-ban Marseille-ben fejlesztettek ki. A Fortran nyelvű Prolog interpreter az Edinburgh-i egyetemről érkezett hozzánk, de ezt szóhossz-problémák miatt nem lehetett életre kelteni.
- Rendelkezésünkre állt D.H.D. Warren diasora a Prologról, lásd `https://www.softwarepreservation.org/projects/prolog/edinburgh/doc/Warren-What_is_Prolog-1974.pdf`. Ennek utolsó három diája „Example to illustrate how Prolog is implemented” szolgált az általam 1975 tavaszán CDL-ben megírt – a világon második – Prolog megvalósítás alapjául. 1979-ben megindult a második magyar Prolog rendszer, az MProlog fejlesztése a Számítástechnikai Koordinációs Intézetben (SzKI), CDL2 nyelven.
- A 1970-es évek második felében Magyarország a Prolog alkalmazások nagyhatalma lett. Darvas Ferenc vezetésével gyógyszerkutatási, Futó Iván csoportjában szimulációs alkalmazásokat fejlesztettek, Márkusz Zsuzsanna előadást tartott az 1977-es IFIP kongresszuson Prolog alapú lakástervezésről. Az első 100 fő fölötti részvételű Prolog konferenciát 1980-ban, Debrecenben rendeztük. Lásd még:

  `https://www.softwarepreservation.org/projects/prolog/nim_iguszi`

Az 1980 utáni időszak

- Az 1980-as évek elején Japán bejelentette az „5. generációs számítógép-rendszerek” projektet (Fifth Generation Computer Project) `https://en.wikipedia.org/wiki/Fifth_Generation_Computer_Systems`. Ennek hatására az MProlog, mint az első ipari minőségű Prolog, jelentős eladásokra tett szert, Európában, Japánban és Amerikában is.
- 1982-ben British Council ösztöndíjasként fél évet töltöttem az Egyesült Királyságban. Angliai tartózkodásom végén meghívtak előadóként egy komoly konferenciára, előadásomról a Computer Weekly is beszámolt.
- 1987–1990 között Warren professzor csoportjában dolgoztam a manchesteri és bristoli egyetemen, a párhuzamos (többprocesszoros) Prolog implementációk témakörében. 1999-ben fél évet dologoztam Svédországban a SICStus Prolog rendszer továbbfejlesztésén.
- 1990 és 2004 között az IQSOFT Rt.-nél dolgoztam, főleg nemzetközi kutatási projekteken. 1994-től félállásban, 2004-től 2012-es nyugdíjazásomig teljes állásban dolgoztam a BME Számítástudományi és Információelméleti tanszékén – azóta ugyanezt csinálom nyugdíjasként.

Rákóczis osztálytalálkozó a Gresham Palotában, 2024. június

II. rész

Elixir: fő jellemzői, telepítés, használat

Tartalom

Az Elixir nyelv fő jellemzői

- Funkcionális
- A nyelvben minden kifejezés
- Rekurzió és magasabb rendű függvények (ciklusok helyett)
- Mintaillesztés
- Nincs semmi megosztva, a processzek üzenetekkel kommunikálnak
- Teljes körű Unicode támogatás, UTF-8 sztringek, karakterláncok
- Erlang függvények hívhatók Elixirből, Elixir függvények Erlangból
- Metaprogramozás, polimorfizmus támogatása
- Dokumentálás támogatása (Markdown formatting language)
- Beépített eszközkészlet támogatja a fejlesztést

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`
  - **asdf** verziókezelő (több verzió telepíthető vele párhuzamosan).
  - Különféle csomagkezelők: Debian, Ubuntu, Windows, Mac Os (nem mindig a legfrissebb verziót telepítik).
  - **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`.
  - Mivel az *asdf* – és így a *mise* is – forrásfájlokat tölt le és fordít, különféle programokra, könyvtárakra van szükségük, különösen ezekre: `git`, `g++`, `libncurses-dev`, `automake`, `autoconf`, `libssl-dev`.

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

Elixir telepítése *mise*-zel

A *mise*-t[3] a `https://mise.jdx.dev/getting-started.html` weblapon leírtak szerint kell telepíteni; többféle eljárás lehetséges.
Ezután érdemes telepíteni a *mise* egy segédcsomagját, majd pedig az Erlang *rebar* nevű csomagkezelőjét, az Erlangot és az Elixirt az alábbi parancsokkal:

```
mise use -g usage          # install mise completions
mise use -g rebar@latest   # install rebar
mise use -g erlang@latest  # install erlang
mise use -g elixir@latest  # install elixir
```

Ha nem a *latest* verziót akarjuk telepíteni, helyette a verziószámot kell megadni. Az elérhető pluginok, ill. plugin-verziók a `mise plugins ls-remote`, ill. a `mise ls-remote` paranccsal listázhatók, pl. `mise ls-remote erlang@27`.
Még aktiválnunk kell a *mise*-t meg a parancskiegészítő módot, ehhez, Linux és Bash használatot feltételezve, a *.bashrc* fájlba az alábbi két sort kell beírni:

```
eval "$(mise activate bash)"
eval "$(mise complete bash)"
```

majd újra beolvasni a *.bashrc*-t: `source .bashrc`.

[3] *mise* ejtése: míz, *mise en place* (gasztronómiai) jelentése: minden elő van készítve

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.
- A függőségek telepítése után további parancsokat kell futtatni, lásd: `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`

Livebook for Elixir: `app/2`, `fac/1`, `sum/1`

Az Elixir-nyelvű funkcionális programozásról szóló első előadáson a *Livebook* segítségével tesszük meg az első lépéseket, ismerkedünk meg a funkcionális programozás alapjaival: három függvényt (`app/2`, `fac/1`), `sum/1` tárgyalunk meg részletesen, elemezzük a kódjukat és a működésüket.
A Livebook-példák interaktív Markdown formátumban innen tölthetők le:

- `https://dp.iit.bme.hu/dp25a/dp25a-fp1ea.livemd`,

képekkel együtt:

- `https://dp.iit.bme.hu/dp25a/dp25a-fp1ea.zip`,

illetve olvasható-nyomtatható PDF-ként:

- `https://dp.iit.bme.hu/dp25a/dp25a-fp1ea.pdf`.

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)> 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().
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)> 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
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#
# 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
$
```

III. rész

Hasznos segédeszközök: mix, benchee

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[4]:

```
~/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.)

[4] 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![5]

```
~/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 ...
```

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

Egészlista elemeinek összege: `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

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.
- Az `iex`-nek az indításakor megadhatjuk a projektünk elérési útjait és függőségeit: `iex -S mix`
- 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:
dp25a-fp1ea-sum-benchee.livemd,
dp25a-fp1ea-sum-benchee.pdf.

IV. rész

Műveletek listákon

Tartalom

Műveletek listákon 1

- A lista láncolt lineáris adatstruktúra, ezért olcsó az első elemét (a *fejét*) és az összes többi elemét (a *farkát*) megkapni, de drága az utolsó elemét elérni, mert végig kell gyalogolni a listán
- A funkcionális nyelvekben, a többi adatstruktúrával egyezően, a lista nem frissíthető, az Elixirben sem: amikor a lista egy elemét le akarjuk cserélni, akkor *másolatot* kell készítenünk a lecserélendő elem előtti részlistáról
- A másolás során a lecserélendő elem előtti összes elemet félre kell raknunk, majd a lecserélendő elem utáni *megosztott* farokrész elé be kell fűznünk az új elemet, ezt követően pedig a félrerakott elemeket egyesével be kell fűznünk az új elemet már tartalmazó listarész elé
- Vagyis a lista adott elemi utáni farkáról nem készül másolat, mert az Elixir *megosztja* a lista farkát a régi és az új elemet tartalmazó listák között
- A lista annyira megkerülhetetlen adatstruktúra a funkcionális nyelvekben, hogy már eddig is sok példát láttunk a használatára. A következő dián összefoglaljuk a leggyakoribb listaműveleteket
- A sztring ugyan nem listaként van ábrázolva az Elixirben, de a használata hasonló, ezért a leggyakoribb sztringműveleteket is összefoglaljuk később egy dián

Műveletek listákon 2

- Lista feje, farka, hossza: `hd(xs)`, `tl(xs)`, `length(xs)`[6]
- Két lista összefűzése (konkatenációja): `xs ++ ys`, eredménye `xs` összes eleme `ys` elé fűzve az eredeti sorrendben
- Két lista különbsége: `xs -- ys`, eredménye `xs` azon elemeinek listája az eredeti sorrendben, amelyek nincsenek benne `ys`-ben
- Tagsági vizsgálat: `x in xs` eredménye `true`, ha `x` eleme `xs`-nek
- Példák

```
iex> [:a, 'a', [65]] ++ [1+2, 2/1, 'a'] # 65 == ?A
[:a, 'a', 'A', 3, 2.0, 'a']
iex> Enum.to_list(1..100000) ++ [100001] # rossz hatékonyságú!
[1, 2, 3, 4, 5, 6, 7, 8, ...100001]
iex> [:a, 'a', [65], 'a'] -- ["A", 2/1, 'a']
[:a, 'A', 'a']
iex> [:a, 'a', [65], 'a'] -- ["A", 2/1, 'a', :a, :a, :a]
['A', 'a']
iex> [1, 2, 3] -- [1.0, 2] # szigorú egyenlőség: 1 ≠ 1.0
[1, 3]
iex> "A" in ["A", 2/1, 'a', :a, :a, :a]
true
```

[6] `hd/1`, `tl/1`, `length/1`, `++/2`, `--/2`, `in/2` a Kernel modulban vannak definiálva

Műveletek listákon 3

Nézzünk további hasznos függvényeket a `List` modulból!

- Lista első / utolsó eleme; ha nincs, `default` vagy `nil`:
  `first(list, default \\ nil)`, `last(list, default \\ nil)`
- Egy elem első előfordulásának törlésével kapott lista:
  `delete(list, elem)`
- Adott pozíciójú elem törlésével / beszúrásával / cseréjével kapott lista:
  `delete_at(list, index)`, `insert_at(list, index, value)`,
  `replace_at(list, index, value)`, `update_at(list, index, fun)`
  Indexelés 0-tól, negatív index a lista végéről indul. `update_at/3` a `fun` függvényt alkalmazza az adott pozíciójú elemre.
- Lista kilapításával / kilapítása után a `tail` elé fűzésével kapott lista:
  `flatten(list)`, `flatten(list, tail)`
- Elem többszörözésével kapott lista: `duplicate(elem, n)`
- Listák listájából ennesek listája: `zip(list_of_lists)`
- Két kis példa:

```
iex> List.zip(['abc', 'defgh', 'ijkl'])
[{97, 100, 105}, {98, 101, 106}, {99, 102, 107}]
iex> List.flatten(['abc', [['defgh']], ['ijkl']], 'zzz')
  'abcdefghijklzzz'
```

Műveletek listákon 4

Milyen hasznos, gyakran használt függvények vannak még listákra?

- Konverziós függvények (ld. 129. dia), pl. `List.to_string`, `Tuple.to_list`
- Van három tesztelő függvény is:
  - `improper?(list)` igaz, ha `list` nem valódi lista[7]
  - `starts_with?(list, prefix)` igaz, ha `list` `prefix`-szel kezdődik
  - `ascii_printable?(list, n \\ :infinity)` igaz, ha `list` első `n` karaktere 7-bites ASCII-kódolású és nyomtatható, beleértve a vezérlő karaktereket is (`\a`, `\b`, `\t`, `\n`, `\v`, `\f`, `\r`, `\e`)

Az `Enum` modul függvényei is alkalmazhatók listákra:

- Lista megfordításával / megfordítása után a `tail` elé fűzésével kapott lista: `reverse(list)`, `reverse(list, tail)`
- Lista adott indexű eleme, ha nincs ilyen, `default` vagy `nil`: `at(list, index, default \\ nil)`

Példák

```
iex> List.starts_with? 'almafa', [?a,?l]
true
iex> (Enum.at (Enum.reverse 'almafa'), 3) === ?m
true
```

[7] Egy lista nem valódi, ha egy listakonstruktorban a farok *nem* lista, pl. `[1,2|3]`, `[:a,:b|nil]`

Műveletek listákon 5

További függvények az `Enum` modulból:

- Lista legkisebb / legnagyobb eleme:
  `min(list, sorter \\ &<=/2, empty_fallback \\ fn -> raise(Enum.EmptyError) end)`
  `max/3` paraméterezése hasonló, `&<=/2` helyett `&>=/2`-vel.
  Ha `list` üres, a 3. paraméterként átadott *függvény* aktivizálódik.
- Lista `n` elemű eleje, `n` elem utáni farka: `take(list, n)`, `drop(list, n)`
  Ha `n` negatív, az elemeket a lista végéről kezdve emeli le / dobja el.
- Lista részlistája: `slice(list, range)` a `range` tartományba eső indexű elemek listája / `slice(list, start, n)` a `start` indextől kezdődő `n` elemű részlista. Ha `range`, ill. `start` negatív, indexelés a lista végéről.

Példák

```
iex> {(Enum.max 'mióta') === ?ó, (Enum.min [], fn -> 0 end)}
{true, 0}
iex> xs='indulakutyasatyukaludni'; {(Enum.take xs,-5), (Enum.drop xs,5)}
{'ludni', 'akutyasatyukaludni'}
iex> {(Enum.slice xs, 6..10), (Enum.slice xs, -10..-7)}
{'kutya', 'tyuk'}
```

Műveletek listákon 6

És még néhány függvény az `Enum` modulból:

- Lista kettévágva: `split(list, n)` ugyanaz, csak rövidebben mint `{(take list, n), (drop list, n)}`
- Lista rendezve alapértelmezés / `fun` függény szerint: `sort(list)`, `sort(list, fun)`
- Lista többszörös értékek nélkül: `uniq(list)`

Példák

```
iex> xs='indulakutyasatyukaludni';[(Enum.split xs,5),(Enum.split xs,-5)]
[{'indul', 'akutyasatyukaludni'}, {'indulakutyasatyuka', 'ludni'}]
iex> Enum.sort xs
'aaaaddiikkllnnsttuuuuyy'
iex> Enum.sort xs, &>=/2
'yyuuuuttsnnllkkiiddaaaa'
iex> Enum.uniq xs
'indulaktys'
```

Az `Enum` modul függvényei – mind *mohó kiértékelésű* – egyéb korlátos, felsorolható (enumerable) adatstruktúrákra is alkalmazhatók.
Nem korlátos adatstruktúrákra a `Stream` modul függvényeit – ezek *lusta kiértékelésűek* – lehet, ill. kell használni.

Tartalom

Listakezelés – rövid példák 1

```
iex(1)> xs = [10,20,30] # mintaillesztés és változó kötése értékhez
[10, 20, 30]
iex(2)> x = hd xs       # hd: lista feje)
10
iex(3)> rs = tl xs      # tl: lista farka
[20, 30]
iex(4)> {zs,xs} = {xs,[5,6]} # mintaillesztés és változó újrakötése
{[10, 20, 30], [5, 6]}
iex(5)> xs              # xs-hez új értéket kötöttünk
[5, 6]
iex(6)> zs              # xs változott, zs nem!
[10, 20, 30]
iex(7)> ^xs = [7,8,9]   # ^: változó 'fixálása', csak mintaillesztés, kötés nélkül [8]
** (MatchError) no match of right hand side value: ~c"\a\b\t" [9] [10]
iex(8)> hd tl xs        # összetett kifejezés is kiértékelhető
6
iex(9)> tl []           # mi az üres lista farka?
** (ArgumentError) errors were found at the given arguments:
  * 1st argument: not a nonempty list
    :erlang.tl([])
```

[8] ^ az ún. 'pin' operátor

[9] Az IEx karakterláncként írja ki a listát, ha összes eleme 7..13, 27 vagy 32..126 értékű egész szám.

[10] A `~c{...}` jelölés egy ún. *szigil*, azaz bűvös jelölés. Határolójelként a `{...}` helyett többnyire használható a `<...>`, `[...]`, `(...)`, `|...|`, `/.../`, `"..."` és `'...'` is. Részletek a Kernel dokumentációjában találhatók.

Listakezelés – rövid példák 2

```
iex(10)> for i <- 7..13, do: [i]
[~c"\a", ~c"\b", ~c"\t", ~c"\n", ~c"\v", ~c"\f", ~c"\r"]
iex(11)> for i <- 27..27, do: [i]
[~c"\e"]
iex(12)> for i <- 32..126, do: [i]
[~c" ", ~c"!", ~c"\"", ~c"#", ~c"$", ~c"%", ~c"&", ~c"'", ~c"(", ~c")", ~c"*",
 ~c"+", ~c",", ~c"-", ~c".", ~c"/", ~c"0", ~c"1", ~c"2", ~c"3", ~c"4", ~c"5",
 ~c"6", ~c"7", ~c"8", ~c"9", ~c":", ~c";", ~c"<", ~c"=", ~c">", ~c"?", ~c"@",
 ~c"A", ~c"B", ~c"C", ~c"D", ~c"E", ~c"F", ~c"G", ~c"H", ~c"I", ~c"J", ~c"K",
 ~c"L", ~c"M", ~c"N", ~c"O", ~c"P", ~c"Q", ...]
iex(13)> IO.puts (for i <- 35..126, do: [i])
#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
:ok
iex(14)> IO.inspect (for i <- 32..126, do: [i]), limit: :infinity
[~c" ", ~c"!", ~c"\"", ~c"#", ~c"$", ~c"%", ~c"&", ~c"'", ~c"(", ~c")", ~c"*",
 ~c"+", ~c",", ~c"-", ~c".", ~c"/", ~c"0", ~c"1", ~c"2", ~c"3", ~c"4", ~c"5",
 ~c"6", ~c"7", ~c"8", ~c"9", ~c":", ~c";", ~c"<", ~c"=", ~c">", ~c"?", ~c"@",
 ~c"A", ~c"B", ~c"C", ~c"D", ~c"E", ~c"F", ~c"G", ~c"H", ~c"I", ~c"J", ~c"K",
 ~c"L", ~c"M", ~c"N", ~c"O", ~c"P", ~c"Q", ~c"R", ~c"S", ~c"T", ~c"U", ~c"V",
 ~c"W", ~c"X", ~c"Y", ~c"Z", ~c"[", ~c"\", ~c"]", ~c"^", ~c"_", ~c"`", ~c"a",
 ~c"b", ~c"c", ~c"d", ~c"e", ~c"f", ~c"g", ~c"h", ~c"i", ~c"j", ~c"k", ~c"l",
 ~c"m", ~c"n", ~c"o", ~c"p", ~c"q", ~c"r", ~c"s", ~c"t", ~c"u", ~c"v", ~c"w",
 ~c"x", ~c"y", ~c"z", ~c"{", ~c"|", ~c"}", ~c"~"]
 [~c" ", ~c"!", ~c"\"", ~c"#", ~c"$", ~c"%", ~c"&", ~c"'", ~c"(", ~c")", ~c"*",
  ~c"+", ~c",", ~c"-", ~c".", ~c"/", ~c"0", ~c"1", ~c"2", ~c"3", ~c"4", ~c"5",
  ~c"6", ~c"7", ~c"8", ~c"9", ~c":", ~c";", ~c"<", ~c"=", ~c">", ~c"?", ~c"@",
  ~c"A", ~c"B", ~c"C", ~c"D", ~c"E", ~c"F", ~c"G", ~c"H", ~c"I", ~c"J", ~c"K",
  ~c"L", ~c"M", ~c"N", ~c"O", ~c"P", ~c"Q", ...]
```

Listakezelés – rövid példák 3

`fpea.ex` – Számlista összege

```
@spec sum(xs::[integer]) :: s::integer
# Az xs számlista összege s
def sum([]), do: 0 # a ", do:" jelölés többsoros változata a "do ... end"
def sum(xs)  do x = hd xs; rs = tl xs; x + sum rs end # újsor helyett ;
```

```
iex(15)> c "fpea.ex"
warning: redefining module Fpea (current version defined in memory)
  fpea.ex:1
[Fpea]
iex(16)> xs = [10, 20.5, 30.5]
[10, 20.5, 30.5]
iex(17)> Fpea.sum xs
61.0
iex(18)> Fpea.sum tl xs
51.0
iex(19)> Fpea.sum(tl(tl(tl xs)))
0
iex(20)> Fpea.sum "abc"  # "abc" !== [97, 98, 99]: "abc" sztring, nem lista
** (ArgumentError) errors were found at the given arguments:
  * 1st argument: not a nonempty list
    :erlang.hd("abc")
    fpea.ex:13: Fpea.sum/1
iex(21)> Fpea.sum 'abc'  # 'abc' === [97, 98, 99]: 'abc' karakterkódok listája
294
```

Listakezelés – rövid példák 4

`fpea.ex` – Két lista összefűzése

```
@spec append(xs::[any], ys::[any]) :: rs::[any]
# rs az xs lista ys elé fűzésével kapott lista
def append([], ys), do: ys
def append(xs, ys), do: [(hd xs) | (append (tl xs), ys)]
@spec revapp(xs::[any], ys::[any]) :: rs::[any]
# rs a megfordított xs lista ys elé fűzésével kapott lista
def revapp([], ys), do: ys
def revapp(xs, ys), do: revapp (tl xs), [(hd xs) | ys]
```

```
iex(22)> c "fpea.ex"
[Fpea]
iex(23)> xs
[10, 20.5, 30.5]
iex(24)> Fpea.append(xs, [:a,:b,:c,:d])
[10, 20.5, 30.5, :a, :b, :c, :d]
iex(25)> Fpea.revapp xs, [:a,:b,:c,:d]
[30.5, 20.5, 10, :a, :b, :c, :d]
```

Mint láttuk, az IEx a sorokat számozza (pl. `iex(25)`), hogy parancsokkal hivatkozni lehessen rájuk. A továbbiakban az egyszerűség kedvéért elhagyjuk a sorszámokat.

V. rész

Műveletek sztringeken

Tartalom

Műveletek sztringeken 1

A sztringek nem listák az Elixirben – mégcsak nem is kollekciók –, de mivel kényelmes listaszerűen kezelni őket, a `String` modulban vannak ezt lehetővé tevő függvények.
Két fogalmat kell megkülönböztetnünk: a kódpontot (code point) és a grafémát (grapheme cluster, röviden grapheme).

- A **kódpont** egyetlen Unicode karakter, egy vagy több bájt ábrázolja

```
iex> {byte_size("á"), String.length("á")}
{2, 1}
```

- A **graféma** egy vagy több kódpont, ami egyetlen karakternek *látszik*

```
iex> str = "\u0065\u0302"; {byte_size(str), String.length(str)}
{3, 1}
iex> "u\u0302"[11]
"û"
iex> String.codepoints(str)
["e", "^"] # Két egykarakteres sztring van a listában.
iex> String.graphemes(str)
["ê"] # Egyetlen egykarakteres sztring van a listában.
```

[11] Az U+0302 Unicode karakter az ún. *Combining Circumflex Accent*.

Műveletek sztringeken 2

- <> a konkatenálás jele: `"ál"<>"om"`
- `string` első / utolsó grafémája, grafémáinak száma: `first(string)`, `last(string)`, `length(string)`
- Graféma `string` `pos` pozíciójában: `at(string, pos)`
- Tartalmazza-e `string` `patts` legalább egy elemét: `contains?(string, patts)`
- `string` elejéről / végéről / mindkettőről levágja a szóköz-jellegű (whitespace) UTF-8 karaktereket: `trim_leading(string)`, `trim_trailing(string)`, `trim(string)`
- Példák

```
iex> str = " "<>" "<>"kutyafüle"<>" "; String.at(str, 8)
"ü"
iex> String.contains?(str, "ü")
true
iex> String.contains?(str, ["ü","ty","n"])
true
iex> String.contains?(str, ["n"])
false
iex> {String.trim_leading(str), String.trim(str)}
{"kutyafüle ", "kutyafüle"}
```

Műveletek sztringeken 3

- Mint a listánál, csak graféma-elemekkel: `slice(string, range)`, `slice(string, start, n)`, `duplicate(string, n)`, `reverse(string)`, `starts_with?(string, prefix)`
- Sztring két darabra vágva adott pozícióban: `split_at(string, pos)`; több darabra szabdalva UTF-8 whitespace-ek mentén:[12] `split(string)`
- Példák

```
iex> str = "indulakutyasatyukaludni"; String.reverse(str)
"indulakuytasaytukaludni"
iex> String.starts_with?(str, "indula")
true
iex> {String.slice(str, 6..10), String.slice(str, -10..-7)}
{"kutya", "tyuk"}
iex> String.duplicate("indul ", 3)
"indul indul indul "
iex> String.split_at(" "<>" "<>"kutya füle macska farka"<>" ", 12)
{"  kutya füle", " macska farka "} # párt ad eredményül
iex> String.split(" "<>" "<>"kutya füle macska farka"<>" ")
["kutya", "füle", "macska", "farka"] # listát ad eredményül
```

[12] A vezető és záró UTF-8 whitespace-eket ignorálja

VI. rész

Típusok, termek, azonosítók, változók

Tartalom

Típusok[13]

Az Elixir erősen típusos nyelv, dinamikus típusellenőrzéssel.

Értéktípusok	**Value types**
Atom	Atom
Tetszőleges hosszú egész szám	Arbitrary-sized integer (integer)
Lebegőpontos szám	Floating-point number (float)
Függvény	Function
Tartomány	*Range*
Reguláris kifejezés	*Regular expression (regex)*
Sztring	*String*

Kollekció-típusok	**Collection types**
Ennes	Tuple
Lista	List
Bináris	Binary
Szótár	Map
Struktúra	Struct

[13] A felsorolás nem teljes. A dőlt betűs típusok más alaptípusokra épülnek.

Atom

- Kettősponttal (:) kezdődik
- Kezdődhet az angol ábécé nagybetűjével is, kettőspont nélkül, de ez konvenció szerint a modulnevekre van fenntartva
- A : után UTF-8 kódolású karaktersorozat, Elixir operátor vagy sztring állhat
- Az UTF-8 kódolású karaktersorozatban betűk, számjegyek és kétféle írásjel (_, @) lehetnek
- A karaktersorozat végén általában kérdőjel (?) vagy felkiáltójel (!) is lehet
- Saját magát jelöli, nem sztring: egy atom értéke maga a neve
- Két azonos nevű atom mindig egyenlő, akárhol is vannak definiálva
- Hasonló a Prolog névkonstanshoz (atomhoz)
- C++, Java nyelvekben a legközelebbi rokon: enum
- Példák: `:János`, `:is_bin?`, `:vált@2`, `:<>`, `:"fun/3"`, `:"éljen soká!"`, `:Éljen_soká!`, `:"Őrült Űrőr tűrjön"`, `Dp`, `Gy1`

Szám

- Egész (integer)
  - Decimális, pl. 1234
  - Hexadecimális, pl. 0xcafe
  - Oktális, pl. 0o765
  - Bináris, pl. 0b1010
  - Tagolható, pl. `123_456_789`
  - Korlátlan pontosságú, pl. `123456789012345678901234567890`
  - Karakterkód (Unicode codepoint)
    - Ha nyomtatható: `?z`
    - Ha vezérlő: `?\n`
- Lebegőpontos (float)
  - Pl. `3.14159`,
  - Vezető nullával, pl. `0.14159`
  - Exponenssel pl. `0.2e-22`
  - IEEE 754 szerinti, dupla pontosságú[14]

[14] 64 bit, kb. 16 számjegy, max. exponens kb. 10^{308}

Függvény (Function) 1 (fájl: fpea.ex)

- 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*
- Példák:

```
iex> fac = &Fpea.fac/1 # &: capture operator
&Fpea.fac/1
iex> fac.(5) # pont és zárójelpár kell, szóközökkel tagolható
120
iex> Kernel.+(3,2) # infix operátor alkalmazása prefix helyzetben
5
iex> fs = [&Kernel.+/2, &*/2, &:math.sin/1] # :math Erlang modul!
[&:erlang.+/2, &:erlang.*/2, &:math.sin/1]
iex> (hd fs).(3,2)
5
iex> (hd tl fs).(3,2)
6
iex> (hd tl tl fs).(:math.pi * 90 / 180)
1.0
```

Függvény (Function) 2 (fájl: fpea.ex)

- További példák: anonim függvény definiálása, hívása, névhez kötése

```
iex> fn ki -> "Szia, " <> ki <> "!" end # <>: konkatenálás
#Function<44.40011524/1 in :erl_eval.expr/5>
iex> fn ki -> "Szia, "<>ki<>"!" end.("Péter") # pont, zárójel!
"Szia, Péter!"
iex> szia = fn ki -> "Szia, " <> ki <> "!" end
#Function<44.40011524/1 in :erl_eval.expr/5>
iex> szia
#Function<44.40011524/1 in :erl_eval.expr/5>
iex> szia.("Bea")
"Szia, Bea!"
```

- További példa: függvénydefiníció `def`-fel, `defp`-vel

```
def sum_of_squares(a,b), do: sqr(a) + sqr(b)
defp sqr(a), do: a*a # p[rivát], azaz lokális a modulon belül
iex> Fpea.sum_of_squares 3, 4.5
29.25
```

- Függvény *típusa:* `(`*arg1 típusa*`,` *arg2 típusa*`, ...)` `::` *eredmény típusa*
  Pl. a `sum_of_squares/2` függvényé: `(number, number) :: number`

Paraméter alapértelmezett (default) értéke (fájl: fpea.ex)

- Egy függvény egy vagy több paraméterének adhatunk alapértelmezett értéket a \\ jelöléssel. Az ilyen paraméter opcionális, a többi elvárt.
- Ha egy függvényt
  - az elvártnál kevesebb paraméterrel hívunk meg, a hívás meghiúsul;
  - az elvárt számú paraméterrel hívunk meg, az összes opcionális paraméter az alapértelmezett értékét veszi fel;
  - az elvártnál több paraméterrel hívunk meg, az aktuális paraméterek értékét balról jobbra haladva veszik fel az opcionális paraméterek.
- Példák alapértelmezett értékekkel

```
def sum_of_sqrs_b5(a, b \\ 5), do: sqr(a) + sqr(b)
iex> Fpea.sum_of_sqrs_b5 3, 4.5
29.25
iex> Fpea.sum_of_sqrs_b5 3
34
def sum_of_sqrs_b5(a \\ 6, b \\ 5), do: sqr(a) + sqr(b)
iex> Fpea.sum_of_sqrs_a6b5 3
34
iex> Fpea.sum_of_sqrs_a6b5
61
```

Ennes (Tuple), tartomány (Range)

Ennes (Tuple)

- Rögzített számú, tetszőleges kifejezésből álló, fix sorrendű kollekció
- Példák:

```
iex> {0x1ff, :erlang, Armstrong, 'Joe'++[0], [], {}}
{511, :erlang, Armstrong, [74, 111, 101, 0], [], {}}
iex> {plus, per, sin} = # mintaillesztések kötésekkel
  {&Kernel.+/2, &//2, &:math.sin/1}
{&:erlang.+/2, &:erlang.//2, &:math.sin/1}
iex> {plus.(3,4), per.(3,4)} # infix volt, prefix lett
{7, 0.75}
iex> sin.(90*:math.pi/180)
1.0
```

Tartomány (Range)

- Egész számok sorozata a *[start, end]* tartományban
- Példa tartomány és lépésköz[15] definiálására, használatára:

```
iex> {18..23, 18..10}
{18..23, 18..10//-1}
iex> for i <- 18..10 // -3, do: i
[18, 15, 12]
```

[15] A lépésközt (//) az Elixir v12.1-ben vezették csak be.

Lista (List)

- Korlátlan számú, tetszőleges kifejezésből álló, *láncolt* sorozat
- Lineáris rekurzív adatstruktúra:
  - vagy üres (`[]` jellel jelöljük),
  - vagy egy elemből áll, amelyet egy lista követ: `[x|xs]`
- Első eleme, ha van, a lista *feje*
- Első eleme utáni, esetleg üres része a lista *farka*

```
iex> [:elem] # egyelemű lista
[:elem]
iex> [:elem|[]] # fejből és üres farokból létrehozott lista
[:elem]
iex> [:elem1|[:elem2]] # fejből-farokból létrehozott lista
[:elem1, :elem2]
iex> [:elem,123,3.14,'elem'] # több elemű listák
[:elem, 123, 3.14, 'elem']
iex> [:elem,123|[3.14,'elem']]
[:elem, 123, 3.14, 'elem']
iex> [:egy|[:két]] ++ [:elem,123|[3.14,'elem']] # ++: konkatenáció
[:egy, :két, :elem, 123, 3.14, 'elem']
```

Karakterlánc (single-quoted)

- Rövidítés, karakterkódok listája: `'erl'` ≡ `[?e,?r,?l]` ≡ `[101,114,108]`
- Az Elixir/Erlang shell a nyomtatható karakterkódok (7..13, 27, 32..126) listáját karakterláncként írja ki
- Ha ezektől különböző érték is van a listában, listaként írja ki
- Példák:

```
iex> [101,114,108]
~c"erl"
iex> [31,101,114,108]
[31, 101, 114, 108]
iex> [a,101,114,108] # szabad változó tilos tömör kif-ben
error: undefined variable "a"
iex> [:a,101,114,108]
[:a, 101, 114, 108]
iex> 'erl' ++ 'ang' # konkatenálható
~c"erlang"
```

- A karakterlánc NEM sztring!

Tartalom

Term

- A *term* tetszőleges adatstruktúra
- Minden termnek van *értéke* és *típusa*
- A term maga is kifejezés
- Közelítő rekurzív definíciója:
  Szám-, atom-, függvény- és más értékekből, ill. *termekből*
  konstruktorokkal felépített, *tovább nem egyszerűsíthető* kifejezés
- Példák
  - `kötött = 2021`
  - Term (tovább nem egyszerűsíthető, tömör[16])
    ```
    123456789
    {'Diák Detti', [{:khf, [:prolog, :elixir, :prolog]}]}
    [&:erlang.+/2, kötött, fn(x,y) -> x*y end]
    ```
  - Nem term (tovább egyszerűsíthető vagy nem tömör)
    ```
    5+6                    # műveletet tartalmaz
    (&:erlang.+/2).(5,6)   # függvényalkalmazást tartalmaz
    szabad                 # szabad változó
    ```

[16] Egy kifejezés akkor tömör, ha kiértékelhető, azaz nincs benne szabad változó

Azonosító (identifier)

- Kisbetűvel vagy aláhúzásjellel (_) kezdődő, betűket, számjegyeket[17] és aláhúzásjeleket tartalmazó, opcionálisan kérdő- vagy felkiáltójellel végződő karaktersorozat
- Konvenció szerint a ?-lel végződő azonosító kiértékelése igazságértéket ad eredményül, a !-lel végződő kiértékelése pedig kivételt dob, ha meghiúsul
- Konvenció szerint az azonosító részeit aláhúzásjellel tagoljuk (ún. megengedő snake_case), vö. atom szintaxisa
- Példák:

```
what_s_in_a_name   name?   exec!
_unused   rómeó_és_Júlia   year_2021
```

- Az azonosító változót vagy függvénynevet jelöl
- Valójában a függvénynév is változó, vagy még inkább: a változó is függvény, mégpedig paraméter nélküli, *konstans függvény* (vö. π)

[17] UTF-8 kódolású betű, ill. decimális számjegy; lásd `https://hexdocs.pm/elixir/unicode-syntax.html`

Változó

- Egy változó lehet *szabad* vagy *kötött*
- A szabad változónak nincs értéke, típusa
- A kötött változó valamely konkrét term *szinonimája*
- A változóhoz köthető új érték, de ez korábbi felhasználását nem módosítja
- A ^ (pin) operátor a kötött változó értékét fixálja: nem köthető új értékhez
- Példák

```
iex> x = fn(x) -> 2*x end # a külső és a belső x nem ugyanaz!
#Function<44.40011524/1 in :erl_eval.expr/5>
iex> y = x
#Function<44.40011524/1 in :erl_eval.expr/5>
iex> ^x = y.(2)
** (MatchError) no match of right hand side value: 4
iex> x = y.(2)
4
iex> y
#Function<44.40011524/1 in :erl_eval.expr/5>
```

Változó hatásköre, komment, igazságérték

- Változó hatásköre: lexikális
  - A függvény törzsében és fejében definiált változók (utóbbiak másnéven: formális paraméterek) lokálisak a függvényre nézve
  - Modulban is lehet változót definiálni, ami csak modulszinten látható, a modulban definiált függvényekből nem
  - `with` kifejezéssel is definiálhatunk lokális változót, például

```
iex> with a = 5, b = 7, do: a*a + 2*a*b + b*b
144
iex> a = 11; with a = 5, b = 7, do: a*a + 2*a*b + b*b; a
11
```

- Komment: # jellel kezdődik, a sor végéig tart
- Igazságérték, másnéven logikai érték (boolean)
  - Három atomot tekintünk igazságértéknek: `:true`, `:false`, `:nil`
  - Mindhárom írható kettőspont nélkül is: `true`, `false`, `nil`
  - A `false` és `nil` *hamis*, minden más érték (nemcsak a `true`) *igaz*[18]

[18] Angolul szokás megkülönböztetni a *true*-t a *truthy*-tól, a *false*-t a *falsy*-tól, pl. JavaScript, Java, Elixir.

Tartalom

Típusok[19]

Az Elixir erősen típusos nyelv, dinamikus típusellenőrzéssel.

Értéktípusok	**Value types**
Atom	Atom
Tetszőleges hosszú egész szám	Arbitrary-sized integer (integer)
Lebegőpontos szám	Floating-point number (float)
Függvény	Function
Tartomány	*Range*
Reguláris kifejezés	*Regular expression (regex)*
Sztring	*String*

Kollekció-típusok	**Collection types**
Ennes	Tuple
Lista	List
Bináris	Binary
Szótár	Map
Struktúra	Struct

[19] A felsorolás nem teljes. A dőlt betűs típusok más alaptípusokra épülnek.

Bináris (Binary)

- A bináris típusba tartozó értékek bitsorozatok
- Egy bináris érték jelölése `<< kif, ... >>` alakú
- A legegyszerűbb `kif` a [0,255] tartományba eső egész szám
- A számokat bájtként tároljuk a binárisban

```
iex> b = << 1, 2, 3 >>
<<1, 2, 3>>
iex> {byte_size(b), bit_size b}
{3, 24}
```

- A tárolásra használt bitek száma megszabható

```
iex> b = << 1::size(2), 1::size(3) >> # 01 001
<<9::size(5)>> # = 9 (decimálisként)
iex> {byte_size(b), bit_size b}
{1, 5}
```

- Egész és lebegőpontos számok és más értékek is tárolhatók binárisan

```
iex> << <<1>> :: binary, <<2.5>> :: binary >>
<<1, 64, 4, 0, 0, 0, 0, 0, 0>>
```

- A bináris tárolás hasznos médiafájlok és UTF-8 karakterek tárolására, processzek közötti kommunikációban stb.

Sztring (String, double quoted)

- UTF-8 kódolású karakterek ábrázolása bájtok sorozataként (bináris típus)
- Következmények:
  - Az UTF-8 kódolás miatt a sztring rövidebb lehet az őt ábrázoló binárisnál
  - A lista- és a sztringműveletek különbözőek
- Példák:

```
iex> dxdy = "δx/δy"
"δx/δy"
iex> {String.length(dxdy), byte_size(dxdy)}
{5, 7}
iex> {String.at(dxdy,0), String.codepoints(dxdy)}
"δ", ["δ", "x", "/", "δ", "y"]
iex> [dx, dy] = String.split(dxdy, "/")
["δx", "δy"]
iex> dx <> "/" <> dy # <>: konkatenálás
"δx/δy"
```

- Sztringműveletekről, a *String* modul függvényeiről volt már szó

Ami közös a karakterláncban és a sztringben

- UTF-8 kódolású karakterekből állnak
- Lehetnek bennük ún. escape-szekvenciák:

\a	BEL (0x07)	\b	BS (0x08)	\d	DEL (0x7f)
\e	ESC (0x1b)	\f	FF (0x0c)	\n	NL (0x0a)
\r	CR (0x0d)	\s	SP (0x20)	\t	TAB (0x09)
\v	VT (0x0b)	\uhhhh	Unicode codepoint in hexadecimal		
\xhh	single byte in hexadecimal				

- Néhány karakter speciális jelentését az elé írt \ megszünteti, pl. \\
- Megengedik az ún. *interpolációt*, azaz változó helyettesítését az értékével ("...#{<expr>}...") sztringben, illetve karakterláncban:

```
iex> name = "dávid" # Sztring
"dávid"
iex> "Helló, #{String.capitalize name}!"
"Helló, Dávid!"
iex> bubo = 'Bubo' # Karakterlánc
~c"Bubo"
iex> "Helló, #{List.to_string [bubo, ? , "Réka"]}!"
"Helló, Bubo Réka!"
```

- Vannak további közös vonások, lásd pl. *Heredocs*, *Sigils*

Kulcs-érték lista (Keyword lists)

- Egy kulcs-érték párt kételemű enneként írhatunk le: `{:key, value}`, ahol a kulcs csak atom, az érték tetszőleges típusú lehet
- Gyakran van szükség ilyen listákra, ezért az Elixir többféle jelölést, rövidítést, bizonyos esetekben zárójelelhagyást is megenged
- Példák

```
iex> [{:név,"Szöszi"},{:szerelme,"jazz-zongorista"},{:város,"Prága"}]
[név: "Szöszi", szerelme: "jazz-zongorista", város: "Prága"]
iex> [név: "Szöszi", szerelme: "jazz-zongorista", város: "Prága"]
[név: "Szöszi", szerelme: "jazz-zongorista", "város: "Prága"]
iex> IO.inspect név: "Szöszi", szerelme: "jazz-zongorista", város: "Prága"
[név: "Szöszi", szerelme: "jazz-zongorista", város: "Prága"]
[név: "Szöszi", szerelme: "jazz-zongorista", város: "Prága"]
iex> inspect név: "Szöszi", szerelme: "jazz-zongorista", város: "Prága"[20]
"[név: \"Szöszi\", szerelme: \"jazz-zongorista\", város: \"Prága\"]"
iex> [:cseh_film, név: "Szöszi", város: "Prága", szerelme: "zongorista"]
[:cseh_film, {név: "Szöszi", város: "Prága", szerelme: "zongorista"}]
iex> {:cseh_film, név: "Szöszi", szerelme: "zongorista", város: "Prága"}
{:cseh_film, [név: "Szöszi", szerelme: "zongorista", város: "Prága"]}
```

- A kulcs-érték párokat leginkább függvényopciók megadására használjuk, pl. `limit: :infinity, charlists: :as_lists`

[20] inspect == Kernel.inspect !== IO.inspect

Szótár (Map) 1

- A szótár kulcs-érték párok rendezett kollekciója
- Jelölés (map literal): `%{ key1 => value1, key2 => value2, ...}`
- Ha a kulcs atom, alternatív jelölés: `%{atom1: value1, atom2: value2}`
- A kulcsok és az értékek típusa tetszőleges; lehet kifejezés is
- Egy szótáron belül a kulcsok különböző típusúak lehetnek
- Példák:

```
iex> states = %{"UA"=>"Ukraine", "SK"=>"Slovakia", "AT"=>"Austria"}
%{"AT" => "Austria", "SK" => "Slovakia", "UA" => "Ukraine"}
iex> msgs = %{{:error,:enoent} => :fatal, {:error,:busy} => :retry}
%{{:error, :busy} => :retry, {:error, :enoent} => :fatal}
iex> colors = %{:red=>0xff0000, :green=>0x00ff00, :blue=>0x0000ff}
%{green: 65280, red: 16711680, blue: 255}
iex> colors = %{red: 0xff0000, green: 0x00ff00, blue: 0x0000ff}
%{green: 65280, red: 16711680, blue: 255}
iex> mix = %{(&+/2).(3,2) => "három+kettő", fütty: "dal"<>"olka"}
%{5 => "három+kettő", :fütty => "dalolka"}
```

Szótár (Map) 2

- A szótár típust elsősorban asszociatív tömbként szokás használni
- Szótárból értéket a kulccsal lehet kinyerni szögletes zárójeles jelöléssel
- Ha a kulcs atom, a rövidebb pontos jelölés is használható
- Példák:

```
iex> states["UA"]
"Ukraine"
iex> states["HU"]
nil
iex> msgs[{:error, :busy}]
:retry
iex> colors[:green]
65280
iex> colors.red
16711680
iex> mix[(&Kernel.*/2).(1,5)]
"három+kettő"
iex> mix.fütty
"dalolka"
```

- További részletek a *Map* modul dokumentációjában

Reguláris kifejezés (Regex) 1

- A reguláris kifejezést ritkán tekintik önálló típusnak; az Elixirben az
- Jelölés: `~r{regexp}`[21] vagy `~r{regexp}options`
- A reguláris kifejezés szintaxisa a PCRE[22] szerinti.
- Példák:

```
iex> Regex.run ~r{[cdr]}, "madárcsicsergés"
["d"]
iex> Regex.scan ~r{[cdr]}, "madárcsicsergés"
[["d"], ["r"], ["c"], ["c"], ["r"]]
iex> Regex.split ~r{[cdr]}, "madárcsicsergés"
["ma", "á", "", "si", "se", "gés"]
iex> Regex.replace ~r{[cdr]}, "madárcsicsergés", "."
"ma.á..si.se.gés"
```

- További részletek a *Regex* modul dokumentációjában

[21] A `~r{...}` jelölés is egy *szigil*, azaz bűvös jelölés. A szigilekről részletek a Kernel dokumentációjában találhatók.

[22] Perl Compatible Regular Expressions, `http:www.pcre.org`

Reguláris kifejezés (Regex) 2

- A regexp után egy vagy több egykarakteres opció állhat

Jel	Jelentés
f	Többsoros sztring első sorában kezdődjön az illesztés
i	Az illesztés ne különböztesse meg a kis- és nagybetűket
m	Többsoros sztring esetén a ^ és a $ az egyes sorok elejét és végét jelentse (a \A és \z jelentése változatlanul a sztring eleje és vége)
s	A . illeszkedjen az újsor-karakterekre is
U	Az egyébként mohó * és + módosítók legyenek lusták, azaz a minta a lehető leghosszabb karaktersorozat helyett a lehető legrövidebbre illeszkedjen
u	Engedje meg Unicode-specifikus minták, pl. \p használatát
x	Engedje meg a bővített mód használatát: ignorálja a szóköz-jellegű (ún. whitespace) karaktereket és a kommenteket (a # jeltől a sor végéig)

- Példák:

```
iex> Regex.run ~r{cs.*s}, "Madarak Csicsergése"
["csergés"]
iex> Regex.run ~r{cs.*s}i, "Madarak Csicsergése"
["Csicsergés"]
iex> Regex.run ~r{cs.*s}iU, "Madarak Csicsergése"
["Csics"]
```

VII. rész

Problémamegoldási technikák

Tartalom

Dinamikus programozás

Dinamikus programozás

Optimalizálási feladatok megoldására használható módszer. Richard Bellman fejlesztette ki 1950 környékén részben a hadsereg megrendelésére. Az elnevezést a jobb eladhatóság miatt adta neki.

- Az eredetihez hasonló részfeladatokat tűzünk ki, ami lehet akár általánosabb is az eredeti feladatnál.
- A részfeladatokat általában kisebb inputra oldjuk először meg.
- A részfeladatok eredményét eltároljuk.
- Sorba rendezem a feladatokat úgy, hogy a későbbi feladatok megoldásánál fel tudjam használni a korábbiak eredményét.
- Az első néhány részfeladat legyen könnyen megoldható önmagában is.
- A későbbi feladatok eredményét a korábbiak eredményét felhasználva kapjuk. A részfeladatokat úgy határozzuk meg, hogy ez könnyen menjen.
- Az összes részfeladat megoldásából már könnyen megkapható az eredeti feladat megoldása.

Dinamikus programozás: Fibonacci. Benchmarking és debugging

A Fibonacci-számok kiszámítására hétféle algoritmust mutat be a második előadás segédanyaga:

- Elágazó rekurzióval
- Memoizálással (dinamikus programozás felülről lefelé haladva), Elixir Map-pel
- Táblázattal (dinamikus programozás alulról felfelé haladva), Elixir Map-pel
- Táblázattal (dinamikus programozás alulról felfelé haladva), Erlang :array-jel
- Táblázattal (dinamikus programozás alulról felfelé haladva), Elixir Arrays-szel
- Táblázattal (dinamikus programozás alulról felfelé haladva), Elixir List-tel
- Az (n-2)-edik (prev) és az (n-1)-edik (curr) Fibonacci-szám nyilvántartásával

A második előadás segédanyaga a honlapról letölthető:

- dp25a-fp2ea-fibonacci-benchee-kino.zip
- dp25a-fp2ea-fibonacci-benchee-kino.livemd
- dp25a-fp2ea-fibonacci-benchee-kino.pdf

A segédanyag a `Benchee` és a `KinoExplorer` modulok használatára is mutat példákat.

Tartalom

Csúszóablakos technika: maximális összegű folytonos részlisták

Egy lista különféle szempontok szerint kiválasztott folytonos részlistáit csúszóablakos módszerrel állíthatjuk elő, egymásba ágyazott ismétlésekkel. Mivel a funkcionális nyelvekben nincs ciklus, az ismétlést rekurzióval valósítjuk meg, a részeredményeket egy, esetleg több akkumulátorban gyűjtjük. Mivel a gyűjtéshez plusz paraméter(ek)re van szükség, általában segédfüggvényeket is definiálunk.
A lista maximális összegű folytonos részlistáinak előállítására többféle megoldást is bemutat a harmadik előadás segédanyaga. Az egyes megoldások futási idejét is megmérjük a `benchee`-vel.
Egy példa:

- Bemenet (egészlista): `[1, 2, 3, 4, -10, 4, 3, 2, 1]`
- Eredmény (egy pár, első eleme a részlisták összege, második eleme ezen maximális összegű részlisták listája):
  `{10, [[1, 2, 3, 4], [1, 2, 3, 4, -10, 4, 3, 2, 1], [4, 3, 2, 1]]}`

A harmadik előadás segédanyaga a honlapról letölthető:

- dp25a-fp3ea-reszlistak-elosztas-kihagy_bevesz_rek.livemd
- dp25a-fp3ea-reszlistak-elosztas-kihagy_bevesz_rek.pdf

Tartalom

Kihagy-bevesz rekurzió: elemek kombinációja, kétfelé válogatása

A harmadik előadás segédanyagában két példa is van a kihagy-bevesz *(inclusion-exclusion)* rekurzióra. Az első, `komb/1` egy lista elemeinek összes kombinációját adja eredményül, a második, `eloszt/1` pedig megmutatja, hogy listaelemeket hányféleképpen lehet úgy kétfelé válogatni, hogy a részösszegek különbségének abszolút értéke a lehető legkisebb legyen.
Példák a függvényhívásokra és az eredményekre:

1. `komb([1,2,3]) == [[], [1], [2], [2,1], [3], [3,1], [3,2], [3,2,1]]`
2. `eloszt([28, 7, 11, 8, 9, 7, 27]) == %{[9, 11, 28] => 48}`
3. `eloszt([4,1,2,5,3]) == %{[4, 2, 1] => 7, [4, 3] => 7, [5, 2] => 7}`

Az `eloszt/1` eredménye egy szótár (map), melynek kulcsai az összegfeltételt kielégítő részlisták, elemei pedig az összegfeltételt kielégítő kisebbik összeg.
A második példában tehát a fennmaradó `[7,7,8,27]` elemekből áll az eredményben nem szereplő másik rész, melynek összege `49`.
A harmadik előadás segédanyaga, mint már írtuk, a honlapról letölthető:

- dp25a-fp3ea-reszlistak-elosztas-kihagy_bevesz_rek.livemd
- dp25a-fp3ea-reszlistak-elosztas-kihagy_bevesz_rek.pdf

Dinamikus programozás: olvasnivalók, feladatok gyakorlásra

- Az *Algoritmuselmélet (VISZAA08)* tantárgy dinamikus programozásról szóló és más prezentációi: `https://www.cs.bme.hu/algel/`
- Horváth Gyula (ELTE) *összefoglalója és feladatgyűjteménye* a dinamikus programozásról: `https://people.inf.elte.hu/szlavi/VersenyFeladatok/DinaProg/dinprog.pdf`
- A Közép-európai Informatikai Diákolimpia (Central-European Olympiad in Informatics, CEOI) *versenyfeladatai:* `http://ceoi.inf.elte.hu/tasks-archive/`
- A Nemzetközi Informatikai Diákolimpia (International Olympiad in Informatics, IOI) *versenyfeladatai:* `https://ioi.contest.codeforces.com/` (regisztráció szükséges)
- *DSA Tutorial – Learn Data Structures and Algorithms:* `https://www.geeksforgeeks.org/dsa/dsa-tutorial-learn-data-structures-and-algorithms/` (sok példával és gyakorló feladattal), többek között a *dinamikus programozásról:* `https://www.geeksforgeeks.org/competitive-programming/dynamic-programming/`

VIII. rész

Hasznos segédeszköz: dialyzer

Tartalom

`dialyxir` telepítése

```
# Run "mix help deps" to learn about dependencies.
defp deps do
 [
  {:dialyxir, "~> 1.4", only: [:dev, :test], runtime: false},
  # {: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 modulokat![23]

```
~/tmp/fp$ mix do deps.get, deps.compile
Resolving Hex dependencies...
Resolution completed in 0.051s
New:
  dialyxir 1.4.6
  erlex 0.2.7
...
Compiling ...
```

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

Szignatúravizsgálat: `mix dialyzer`

- A projekt forrásfájljainak a `lib` mappában kell lenniük: rakjunk be ide egy Elixir programot, pl. az egyik kisházit, rontsunk el egy-két specifikációt, és dializáljuk.
- A dializálás a `lib` mappában lévő **összes** `.ex` fájlt vizsgálja.
- Az első futtatás soká tart, mert a *dialyzer* rengeteg ún. PLT-fájlt telepít a modulokhoz tartozó típusszignatúrákkal (PLT = Persistent Lookup Table).
- A dializálás a `.beam` fájlokat elemzi, ezért ha változott valamelyik forrásfájl, az elemzés előtt fordítás készül belőle.

```
@spec sum(xs::[integer()]) :: s::[integer()]
# Az xs számlista összege s
def sum([x|xs]), do: x + sum(xs)
def sum([]), do: 0
```

```
~/tmp/fp$ mix dialyzer
lib/sum.ex:2:invalid_contract
The @spec for the function does not match the success typing ...
Function: Sum.sum/1
Success typing: ([number()]) -> number()
But the spec is: (xs::[integer()]) -> s::[integer()]
```

IX. rész

For-jelölés (for-comprehension)

Tartalom

For-jelölés 1

Gyűjtemények (kollekciók) kezelésére használjuk a for-jelölést, vagy az angol elnevezést teljesen átvéve: a for-komprehenziót). Most vegyünk sorra mindent, amit a for-jelölésről tudni kell (a szögletes zárójelek jelentése itt: *opcionális).*

A for-jelölés: `for` q_1[, q_2, ..., q_n][, `into:` coll], `do:` exp, ahol

- a q_i
  1. `pattern <- list` alakú *generátor*, vagy
  2. *predikátum* (igazságérték-eredményű függvény, feltétel);
- legalább egy q_i-nak *generátornak* kell lennie;
- a `pattern` mintának illeszkednie kell a `list` lista kiválasztandó elemeire, és ki kell elégítenie az adott `pattern <- list` generátortól jobbra álló összes q_i predikátumot;
- az `exp` tetszőleges, a `pattern` mintától függő vagy nem függő kifejezés;
- a generátorban a minta előállítására lista helyett más felsorolható kollekciót, leggyakrabban tartomány-típusú értéket is megadhatunk.

(Folytatás a következő dián.)

For-jelölés 2

A for-jelölés (folyt.): `for` q_1[, q_2, ..., q_n][, `into:` coll], `do:` exp, ahol

- az opcionális `into:` opció után álló `coll`-lal megadhatjuk, hogy *milyen típusú* felsorolható kollekciót hozzon létre a for-jelölés, ha elhagyjuk, alapértelmezés szerint lista jön létre;
- `coll`-ként üres kollekciót kell megadni: `""` (sztring), `%{}` (szótár), `[]` (lista), <<>> (bináris);
- a for-jelölésben definiált változók lokálisak;
- a for-jelölés *értéke* az összes olyan $\texttt{exp}_j$ kifejezés kollekciója, amelyre a **mintaillesztés sikerült** és a **predikátumok teljesültek;**
- a for-jelölés eddig bemutatott változata tehát egy vagy több kollekció elemein műveletek elvégzésére és/vagy bizonyos elemek szűrésére használható (vö. `Enum.map/2`, `Enum.filter/2`).
- Összefoglaló a for-jelölésről: https://www.mitchellhanberg.com/the-comprehensive-guide-to-elixirs-for-comprehension/
- A komprehenzió ma már sokféle programozási nyelvben megtalálható, lásd: https://en.wikipedia.org/wiki/List_comprehension

For-jelölés 3

A for-jelölés `uniq:` opcióval:

`for` q_1[, q_2, ..., q_n][, `into:` coll], `uniq: true|false, do:` exp, ahol

- a `uniq: true` opció hatására az előállított gyűjteménybe csak egymástól különböző értékek kerülnek be;
- csak lista, sztring és bináris esetén van értelme használni, hiszen a szótárban a kulcsok sohasem ismétlődhetnek.

A for-jelölés `reduce:` opcióval:

`for` q_1[, q_2, ..., q_n], `reduce:` acc0 `do` acc -> fun(pat, acc) `end`, ahol

- a `reduce:` opcióval a for-jelölés nem az `Enum.map/2`-et, hanem az `Enum.reduce/3`-t váltja ki,
- az acc0 az eredményt gyűjtő akkumulátor kezdőértéke,
- a `do` ... `end` között egy névtelen függvényt kell megadni (az `fn` és a hozzá tartozó `end` nélkül!), amelynek egyetlen argumentuma az acc akkumulátor (a neve bármi lehet), a törzsében pedig egy olyan kétargumentumú függvényt vagy operátort kell használni, amelynek két argumentuma közül az egyik a generátorban használt pat minta, a másik pedig ugyancsak az acc akkumulátor (az argumentumok sorrendjének hatása lehet az eredményre!).

For-jelölés: kis példák 1

```
iex> for x <- 1..6 // 2, do: x      # { x | x ∈ {1,3,5} }
[1, 3, 5]
iex> for x <- [1,2,3], do: 2*x+1    # { 2·x+1 | x ∈ {1,2,3} }
[3, 5, 7]
iex> for x <- 1..9, rem(x, 2) === 0, x > 2, do: 2*x
[8, 12, 16]
iex> for {k,v} <- [egy: 1, két: 2, há: 3], into: %{}, do: {k,v}
%{egy: 1, há: 3, két: 2}
iex> for {k,v} <- %{egy: 1, két: 2, há: 3}, into: [], do: {k,v}
[egy: 1, há: 3, két: 2]
iex> for c <- [?c, ?s, ?ó, ?k, ?a], into: "", do: <<c>>
<<99, 115, 243, 107, 97>>
iex> for c <- [?c, ?s, ?o, ?k, ?a], into: "", do: <<c>>
"csoka"
iex> for x <- 0..-2 // -2, y <- 1..x, do: {x,y}
[{0, 1}, {0, 0}, {-2, 1}, {-2, 0}, {-2, -1}, {-2, -2}]
iex> for x <- 0..-2 // -2, y <- 1..x, xy = {x,y}, do: xy
[{0, 1}, {0, 0}, {-2, 1}, {-2, 0}, {-2, -1}, {-2, -2}]
iex> for x <- 2..4, rem(x,3) !== 0, y <- 1..3, x > y, do: {x,y}
[{2, 1}, {4, 1}, {4, 2}, {4, 3}]
```

For-jelölés: kis példák 2

```
iex> for i <- 1..3, j <- 2..1//-1, do: {i,j} # keresztszorzatok listája
[{1, 2}, {1, 1}, {2, 2}, {2, 1}, {3, 2}, {3, 1}]
iex> for i <- 1..3, do: (for j <- 2..1//-1, do: {i,j}) # listák listája
[[{1, 2}, {1, 1}], [{2, 2}, {2, 1}], [{3, 2}, {3, 1}]]
iex> for i <- 1..3, j <- 2..1//-1, into: %{}, do: {i,j} # ismétlődő kulcs felülír!
%{1 => 1, 2 => 1, 3 => 1}
iex> for i <- 1..3, j <- 2..1//-1, into: %{}, do: {i+j*4,j} # így a kulcsok egyediek
%{5 => 1, 6 => 1, 7 => 1, 9 => 2, 10 => 2, 11 => 2}
iex> for _i <- 1..5, into: "", do: "1" # többszörözve
"11111"
iex> for _i <- 1..5, into: "", uniq: true, do: "1" # azonosak csak egyszer
"1"
iex> for _i <- 1..5, into: <<>>, do: <<1::size 1>> # 5 bit (0b11111), 1 byte
<<31::size(5)>>
iex> for _i <- 1..5, into: <<>>, uniq: true, do: <<1::size 1>> # 1 bit (0b1), 1 byte
<<1::size(1)>>
iex> for _i <- 1..5, into: <<>>, do: <<1::size 2>> # 10 bit (0b01010101, 0b01), 2 byte
<85, 1::size(2)>>
iex> for _i <- 1..5, into: <<>>, uniq: true, do: <<1::size 2>> # 2 bit (0b01), 1 byte
<1::size(2)>>
```

For-jelölés: nagyobb példák (fájl: fpea.ex)

- Pitagoraszi számhármasok

```
@spec pitag(n::integer) :: ps::[{integer, integer, integer}]
# az n-nél nem nagyobb összegű pitagoraszi számhármasok listája ps
def pitag(n), do:
    (ls = 1..n
     for a <- ls, b <- ls, a < b,
         c <- ls, a + b + c <= n,
         a * a + b * b === c * c,
           do: {a, b, c}
    )
iex> Fpea.pitag 12
[{3, 4, 5}]
iex> Fpea.pitag 36
[{3, 4, 5}, {5, 12, 13}, {6, 8, 10}, {9, 12, 15}]
```

- Hányszor fordul elő egy elem egy listában?

```
@spec freq(val::any, ls::[any]) :: n::integer
# ls-ben a val értékű elemek száma n
def freq(elem, ls), do:
      length(for x <- ls, b = (x === elem), do: b)  # b = ...: réteges minta!
iex> Fpea.freq ?a, 'almafa'
3
```

Gyorsrendezés (Quicksort) for-jelöléssel (fájl: fpea.ex)

```
@spec qsort(us::[any]) :: ss::[any]
# Az us lista elemeinek monoton növekedő listája ss
def qsort([]), do: []
def qsort([pivot|tail]), do:
      qsort(for x <- tail, x < pivot, do: x) ++
        [ pivot | qsort(for x <- tail, x >= pivot, do: x) ]
```

Példák:

```
iex> Fpea.qsort [34, 1, 55, 78, 43.2, :math.pi(), :math.exp(1), 31.7]
[1, 2.718281828459045, 3.141592653589793, 31.7, 34, 43.2, 55, 78]
iex> Fpea.qsort [:ab, :acb, :aca, :bca, :bbca, :bac, :abc, :a, :b, :c]
[:a, :ab, :abc, :aca, :acb, :b, :bac, :bbca, :bca, :c]
iex> Fpea.qsort 'the quick brown fox jumps over the lazy dog'
~c"        abcdeeefghhijklmnoooopqrrsttuuvwxyz"
iex> Fpea.qsort ["ba",'ba',:ba,9.3,6,&:math.exp/1,[4,5],[],{2,3}]
[6, 9.3, :ba, &:math.exp/1, 2, 3, [], [4, 5], ~c"ba", "ba"]
```

X. rész

Műveletek, BIFek, típusvizsgálat/konverzió, mintaillesztés

Tartalom

Aritmetikai és bitműveletek

- Aritmetikai műveletek (Kernel modul)
  - Előjel: `+`, `-` (precedencia: 1)
  - Multiplikatív műveletek: `*`, `/`, `div`, `rem` (precedencia: 2)
  - Additív műveletek: `+`, `-` (precedencia: 3)
- Bitműveletek (Bitwise modul)
  - `bnot` vagy `~~~`, `band` vagy `&&&` (precedencia: 2)
  - `bor` vagy `|||`, `bxor`, `bsl` vagy `<<<`, `bsr` vagy `>>>` (precedencia: 3)

Megjegyzések

- `+`, `-`, `*` és `/` egész és lebegőpontos operandusokra is alkalmazhatók
- `+`, `-` és `*` eredménye egész, ha mindkét operandusuk egész, egyébként lebegőpontos
- `/` eredménye mindig lebegőpontos
- `div` és `rem` prefix helyzetűek, ereményük egész
- `div`, `rem` és a bitműveletek operandusai csak egészek lehetnek
- `~~~`, `&&&`, `|||`, `<<<`, `>>>` infix, a többi bitművelet prefix helyzetű
- A bitműveleteket engedélyezni kell: `use Bitwise` vagy

```
use Bitwise[, (only_operators | skip_operators): true]
```

Összehasonlító műveletek (relációk)

- Egy reláció (összehasonlítás) eredménye a `true` vagy `false` atom
- Termek összehasonlítási sorrendje (vö. típusok):
  `number < atom < reference < function < port < pid < tuple < list < binary`
- Kisebb, kisebb-egyenlő, nagyobb-egyenlő, nagyobb: <, <=, >=, >
- *Érték szerinti* egyenlőség (`integer` és `float` lehet egyenlő): `==`, `!=`
- *Szigorú egyenlőség* (`integer` és `float` nem lehet egyenlő): `===`, `!==`
- Példák: `5.0 == 5 == true`, `5.0 === 5 == false`

Elrettentő példák:

```
10.1 - 9.9 == 0.2 ⇝ false
(10.1 - 9.9) * 10 ⇝
1.99999999999993
0.000000000000001 + 1 == 1 ⇝ false
0.0000000000000001 + 1 == 1 ⇝ true
```

Lebegőpontos értékek összehasonlítása helyett vizsgáljuk a különbségüket a <= vagy >= relációval (ε-nál kisebb-e a különbségük?)

Logikai műveletek

- Prefix helyzetű operátor: `not` és `!`; `not` operandusa csak `boolean`, `!`-é tetszőleges típusú kifejezés lehet
- Infix helyzetű operátorok: `and` és `&&`, `or` és `||`[24]
  - `and` és `or` első operandusa csak `boolean`, `&&` és `||` első operandusa tetszőleges típusú kifejezés lehet[25]
  - Második operandusuk tetszőleges típusú kifejezés lehet
  - Eredményük típusa a két operandus típusának uniója
  - Lusta kiértékelésű, ún. *short-circuit* műveletek: ha az első operandus kiértékelése eldönti eredményt, a másodikra nem kerül sor
- Példák:

```
iex> !:atom && div(3,0) === 2
false
iex> :atom && div(3,0) === 2
** ...bad argument in ...: div(3, 0)
iex> :atom and rem(3,2) === 1
** ...expected a boolean on left-side of "and"
```

```
iex> true and rem(3,2)
1
iex> false and rem(3,2)
false
iex> nil && rem(3,2)
nil
```

[24] `not`, `and` és `or` használható őrkifejezésben, `!`, `&&` és `||` nem.

[25] A `false` és `nil` értékű kifejezéseket kivéve *minden más* érték `true`-nak számít.

Beépített függvények (Built-In Functions, BIFs)

- A BEAM-be beépített, rendszerint C-ben írt függvények
- Többségük az *erts* Erlang-könyvtár `erlang` moduljának része
- Elixir-specifikációjuk az Elixir Kernel moduljában található
- A csak az Erlang `erlang` moduljában definiált BIF-ek az `:erlang` modulnévvel hívhatók
- Az alaptípusokon alkalmazható leggyakoribb BIF-ek:
  - Számok: `abs(num)`, `trunc(num)`, `ceil(num)`, `floor(num)`, `round(num)`, `:erlang.float(num)`[26]
  - Sztring, bináris: `bit_size(string)`, `byte_size(string)`
  - Szótár: `map_size(map)`
  - Ennes: `tuple_size(tuple)`, `elem(tuple, index)`, `put_elem(tuple, index, value)`[27]
  - Lista: `length(list)`, `hd(list)`, `tl(list)`
- Az operátorok is BIF-ek a `Kernel`-ben, pl. `Kernel.*(3,4)`

[26] `:erlang.float` helyett 1-gyel oszthatjuk az egész számot, pl. `5/1`

[27] Megjegyzés: $0 \leq$ `index` $\leq$ `tuple_size(tuple)-1`

Egyéb alapfügvények (típusvizsgálat és típuskonverzió)

- Típusvizsgálat (BIF-ek a Kernelben)
  - `is_integer(term)`, `is_float(term)`, `is_number(term)`,
  - `is_atom(term)`, `is_boolean(term)`, `is_nil(term)`,
  - `is_binary(term)`, `is_bitstring(term)`,
  - `is_tuple(term)`, `is_list(term)`„ `is_map(term)`
  - `is_function(term)`, `is_function(term, arity)`
- Típuskonverzió (az egyes típusokhoz tartozó modulokban)
  - Atom: `to_charlist(atom)`, `to_string(atom)`,
  - Float: `to_charlist(float)`, `to_string(float)`,
  - Integer: `to_charlist(integer)`, `to_string(integer)`,
  - List: `to_atom(list)`, `to_charlist(list)`, `to_float(list)`, `to_integer(list)`, `to_integer(list, base)`, `to_string(list)`, `to_tuple(list)`
  - String: `to_atom(list)`, `to_charlist(string)`, `to_float(string)`, `to_integer(string)`, `to_integer(string, base)`,
  - Tuple: `to_list(tuple)`,
  - Map: `to_list(map)`,

Tartalom

Mintakifejezés, minta, mintaillesztés (pattern matching)

- Mintakifejezés, röviden minta: termhez hasonló olyan kifejezés, amelyben nincs függvénykifejezés, de lehet benne szabad változó
- Egy szabad változó mindenre illeszkedik, és lehet rá hivatkozni
- Az aláhúzásjel (_) és a vele kezdődő változónév mindenre illeszkedik; az előbbire nem lehet, az utóbbira nem szokás hivatkozni
- Egy mintában ugyanaz a változó többször is előfordulhat, ha mindenütt azonos értékre kell illeszkednie
- A mintaillesztés műveleti jele az =, bal oldalán a mintával, jobb oldalán egy tömör kifejezéssel (a mintaillesztés egyirányú)
- A mintaillesztés a minta nem fixált (vö. ^ operátor) változóit értékhez köti
- A kötés **nem** értékadás!
- Függvényhíváskor az aktuális paramétereket *illesztjük* a formális paraméterekre
- Ha egy változót értékhez kötünk, de nem használjuk, az Elixir figyelmeztet rá, kivéve akkor, ha a változónév _-sal kezdődik
- Figyelem: a Prologban a funkcionális nyelvekkel ellentétben *kétirányú* mintaillesztés van, *egyesítés* a neve

Példák mintaillesztésre 1

```
iex> [x, &+/2] = [5, &+/2]
** (CompileError) ... & is not allowed in matches
iex> [x, f] = [5, &+/2]
[5, &:erlang.+/2]
iex> [x, f] = [5, f]
[5, &:erlang.+/2]
iex> a = fn(x) -> x+1 end
#Function<44.40011524/1 in :erl_eval.expr/5>
iex> {a, b} = {fn(x) -> x+1 end, 23}
#Function<44.40011524/1 in :erl_eval.expr/5>, 23
iex> fn(x) -> x+1 end = a
** (CompileError) ... fn is not allowed in matches
iex> 3 = szabad
** (CompileError) iex:505: undefined function szabad/0
iex> [z | zs] = [0,1,2,3]
[0, 1, 2, 3]
iex> [z1 | [z2 | [z3 | [z4 | zs]]]] = [0,1,2,3]
[0, 1, 2, 3]
iex> [z1, z2, z3 | [3]] = [0,1,2,3]
[0, 1, 2, 3]
```

Példák mintaillesztésre 2

```
iex> [z1, z2, z3 | 3] = [0,1,2,3]
** (MatchError) no match of right hand side value: [0, 1, 2, 3]
iex> [z1, z2 | [3]] = [0,1,2,3]
** (MatchError) no match of right hand side value: [0, 1, 2, 3]
iex> {{a, b}, {a, b}} = {{:a, :b}, {:a, :b}}
{{:a, :b}, {:a, :b}}
iex> {{a, b, a}} = {{:a, :b, :b}}
** (MatchError) no match of right hand side value: {{:a, :b, :b}}
iex> {a, b, _b, _} = {:a, :b, :b, :a}
{:a, :b, :b, :a}
iex> {a, b}
{:a, :b}
iex> _b
warning: the underscored variable "_b" is used after being set...
please rename the variable to remove the underscore
:b
iex> x = %{b: "barna", z: "zöld"}
%{b: "barna", z: "zöld"}
iex> %{k1: v1, k2: v2} = x
** (MatchError) no match of right hand side value: %{b: "barna", z: "zöld"}
```

Példák mintaillesztésre 3

```
iex> %{k1 => v1, k2 => v2} = x
** (CompileError) iex:3: cannot use variable k1 as map key inside a pattern...
iex> %{b: v1, z: v2} = x
%{b: "barna", z: "zöld"}
iex> {v1, v2}
"barna", "zöld"
iex> %{z: ^v1, b: v2} = x
** (MatchError) no match of right hand side value: %{b: "barna", z: "zöld"}
iex> %{z: v1, b: v2} = x
%{b: "barna", z: "zöld"}
iex> {v1, v2}
"zöld", "barna"
iex> %{b: v} = x # részleges mintaillesztés
%{b: "barna", z: "zöld"}
iex> v
"barna"
iex> ([y|ys] = yss) = [1, 2, 3] # réteges minta (layered pattern)
[1,2,3]
iex> {y, ys, yys}
{[1], [2, 3], [1,2,3]}
```

Függvénydefiniálás mintaillesztéssel (fájl: fpea.ex)

- Már láttunk rá példákat korábban
- A funkcionális, ill. általában a deklaratív nyelvekben az *if*, *switch*, *case* stb. vezérlési szerkezetek helyett, ha lehet, mintaillesztést használunk
- *Klózoknak* nevezzük egy azonos nevű – vagy névtelen – és argumentumszámú függvény különféle esetekre illeszkedő verzióit
- Rekurzív függvényt csak `def` vagy `defp` definícióval lehet létrehozni
- Példák (Erlang-stílusú hibajelzéssel)

```
def head([x|_xs]), do: {:ok, x}
def head([]),      do: {:error, nil}
iex> {(Fpea.head []), Fpea.head [3,4,5]}
{{:error, nil}, {:ok, 3}}
iex> tail = fn [_x|xs] -> {:ok, xs}; [] -> {:error, nil} end
#Function<44.40011524/1 in :erl_eval.expr/5>
iex> {tail.([]), tail.([3,4,5])}
{{:error, nil}, {:ok, [4, 5]}}
iex> cnt_a = fn [_x|xs] -> (cnt_a xs) + 1; [] -> 0 end
** (CompileError) iex:73: undefined function cnt_a/1 ...
def cnt_n([_x|xs]), do: (cnt_n xs) + 1; def cnt_n([]), do: 0
iex> Fpea.cnt_n(~c"alma")
4
```

XI. rész

Programozás, funkcionális programozás

Tartalom

Programozás az 1970-es évek első feléig

- Sokféle, de egyszerű CPU: assembly nyelvek
- Ún. autokódok („emberközeli gépi kód”): MOST (Odra), FOCAL (PDP)
- FORTRAN, COBOL
- *LISP,* BASIC (interpretált)
- ALGOL
- Pascal (oktatási célra)

Tipikus jellemzők

- Monolit programozás (nincsenek modulok)
- GOTO használata
- Szubrutinok (nem rekurzív), eljárások (rekurzív is lehet)
- Típusok nincsenek vagy megkerülhetőek
- Túlzottan megengedő, sokszor rosszul definiált szintaxis
- Globális, átírható (frissíthető) változók, nincs deklarációkényszer
- Bottom-up (alulról felfelé haladó) kódolás
- Párhuzamos és eloszott programozást nem támogatja

$\Rightarrow$ Nem megbízható, nem biztonságos, nehezen karbantartható stb. szoftver

Egy hírhedt hiba a NASA-nál a 1960-as évek elejéről

```
DO 10 I=1.10
...
10 CONTINUE
```

amit így értelmezett a FORTRAN fordító (mert a szóközöket a FORTRAN megengedte, de figyelmen kívül hagyta a változónevekben):

```
DO10I=1.10
...
10 CONTINUE
```

vagyis létrehozta a `DO10I` változót, ami az `1.1` értéket vette fel, miközben a programozó egy `DO`-ciklust akart írni (pont helyett vesszővel):

```
DO 10 I=1,10
...
10 CONTINUE
```

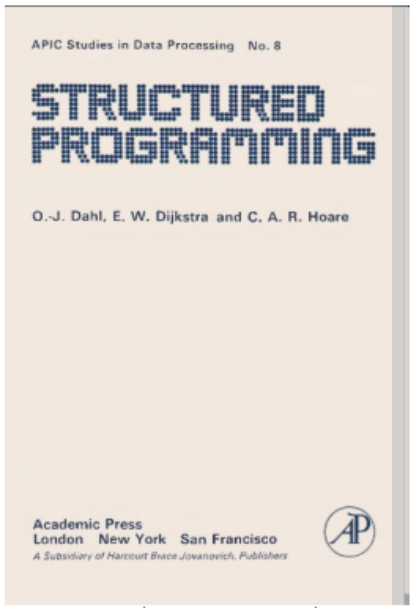
NASA, 1960-as évek eleje, Lásd:
`http://spiff.rit.edu/classes/phys317/lectures/boom/mercury.txt`

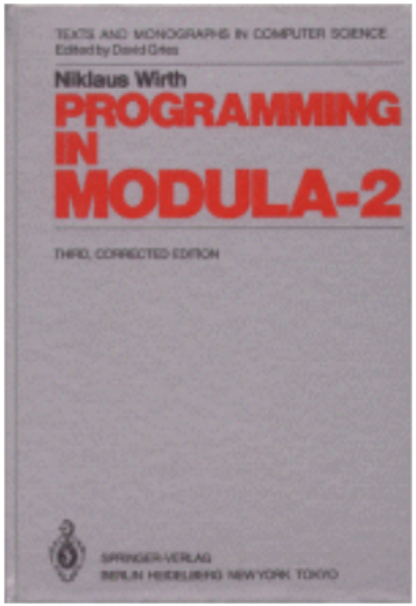
Programozás az 1970-es évek második felétől

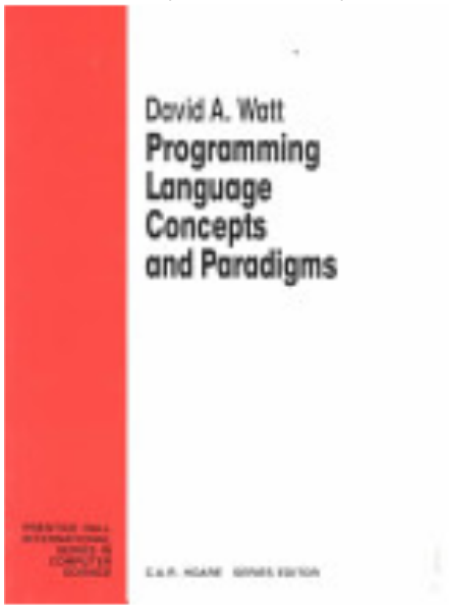
- „C”, később C++ programozási nyelv
- Absztrakt adattípusok
- Simula, ALGOL 68
- Structured Programming (Dahl, Dikstra, Hoare), 1972
- Modular Programming, Modula (Wirth, 1972)
- CDL (Compiler Description Language, Koster, 1971)
- ELAN (Educational Language, Koster, 1974)
- Objektum-orientált programozás
- Ada
- Eiffel (B. Meyer), Sather (UC Berkeley)
- Specifikációs nyelvek (pl. „Z”)
- SML, OCAML funkcionális programozási nyelvek

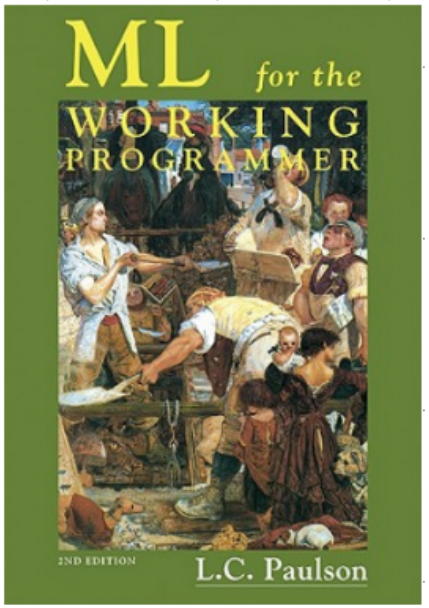
$\Rightarrow$Törekvés megbízható, biztonságos, karbantartható szoftverre

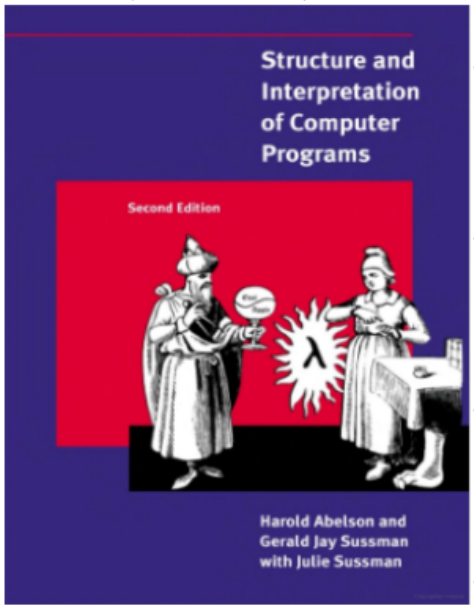
Néhány (nekem) fontos könyv a programozásról











Funkcionális programozás, nyelvek

Hallott már korábban (a DP felvételét megelőzően) a funkcionális programozásról?

Ha igen, milyen programozási nyelv(ek) jut(nak) az eszébe?

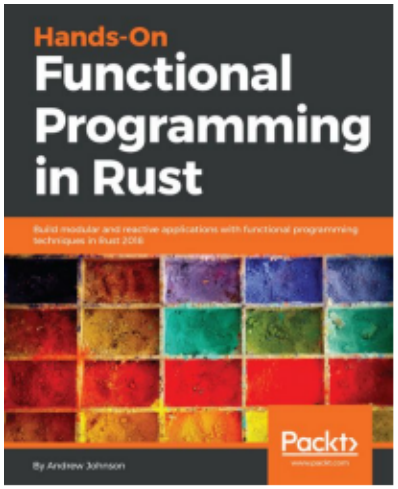
Mi jutott a Google „eszébe” 2021-ben, amikor a *books* és *functional programming* szavakra kerestem rá? [28]

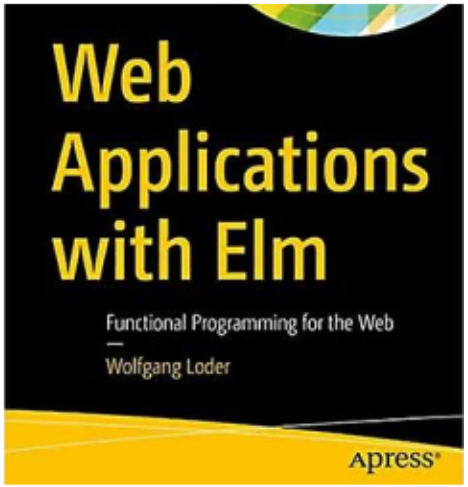
[28] A találati sorrend nagyjából a következő diákon látható sorrend volt 2021-ben, a többszörös vagy hasonló találatok közül csak egyet-kettőt hagytam meg.

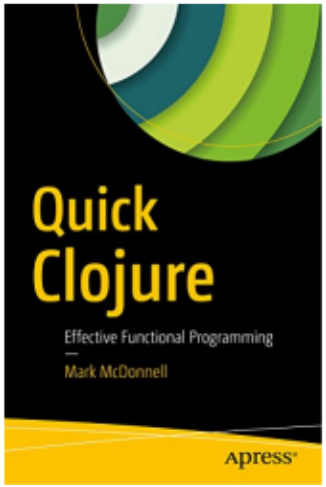
Könyvek a funkcionális programozásról 1

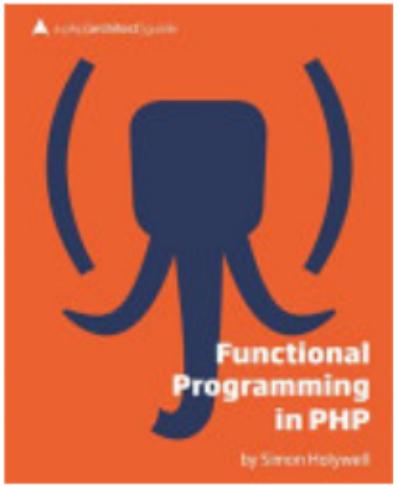
JavaScript, Scala, **Haskell**, TypeScript, Python, Kotlin, **F#**, C++, C#

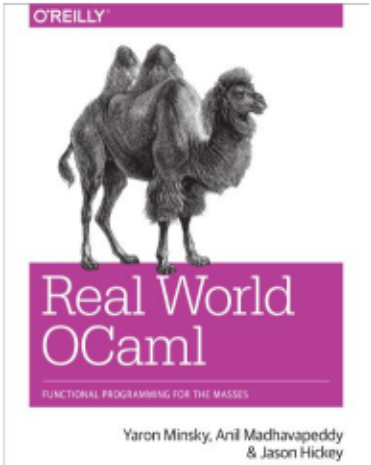
Könyvek a funkcionális programozásról 2

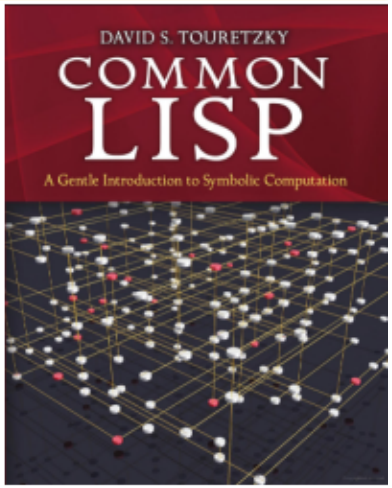






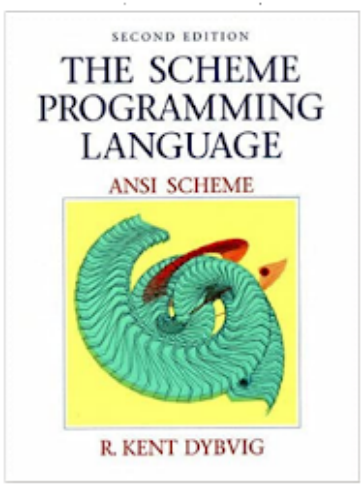


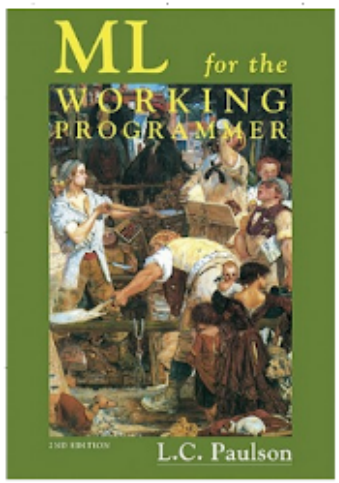


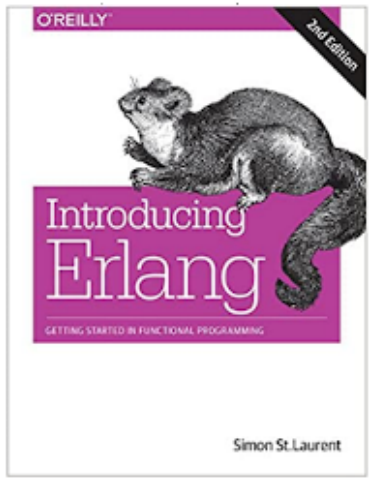


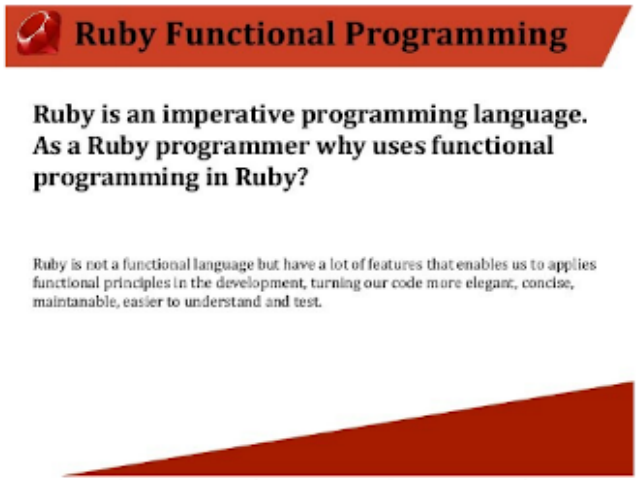
Rust, Swift, **Elm**, **Clojure**, Java, PHP, **OCaml**, **Lisp**

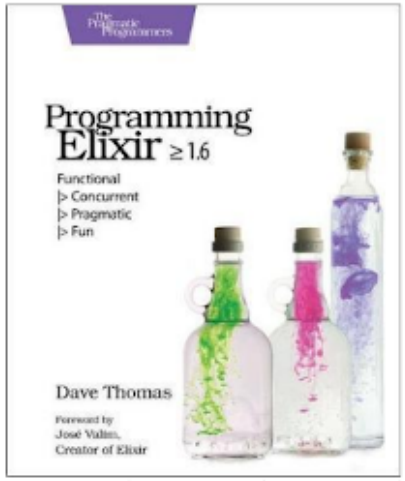
Könyvek a funkcionális programozásról 3

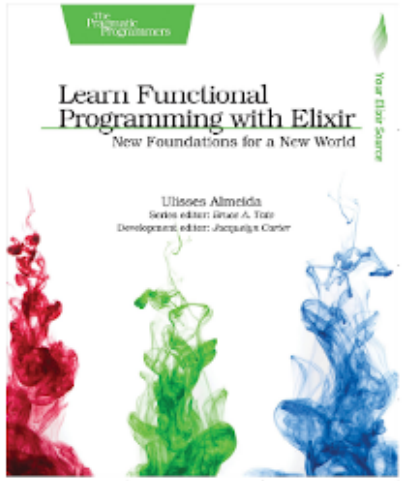












Scheme, **SML**, **Erlang**, Ruby, **Elixir**

Funkcionális programozási nyelvek, nyelvcsaládok

- Lisp (Common Lisp) – az ős, Scheme, Clojure (JVM-en fut) [$\mathbb{D}$]
- SML, Caml, Caml Light, OCaml, Alice, F# (.NET) [$\mathbb{S}$]
- Clean, Haskell (pure FP languages) [$\mathbb{S}$]
- Erlang, Elixir (Erlang VM-en [BEAM] fut) [$\mathbb{D}$]
- Elm (JavaScriptre fordít) [$\mathbb{S}$]
- Funkcionális is: Kotlin (JVM-re, JavaScriptre, LLVM közvetítésével gépi kódra fordít), Python, Julia, Scala, Rust, Swift, ... [$\mathbb{D}$]
- Funkcionális, logikai és imperatív: Flix (ML-család, JVM-re fordít) [$\mathbb{S}$]

$\mathbb{D}$: dinamikusan típusos, $\mathbb{S}$: statikusan típusos

Funkcionális programozás (FP): mi az?

- Programozás *függvények alkalmazásával*.
- Ritkábban *applikatív programozásnak* is nevezik (vö. function application).
- A függvény: leképezés – az argumentumából állítja elő az eredményt.
- A tiszta (matematikai) függvénynek nincs *mellékhatása.*
- Az FP fő jellemzői:
  - függvények (csak **bemenő** paraméterek + **visszatérési** érték),
  - nem frissíthető változók, kötések,
  - rekurzió (algoritmusok, adatok – **listák,** fák),
  - magasabb rendű függvények.
- A mellékhatás kizárása (vagy kordában tartása) miatt az FP-nyelvek különösen alkalmasak a párhuzamos programozásra (többmagos processzorok, elosztott rendszerek).

Funkcionális programozási szemlélet

- Minek köszönhető a funkcionális programozási *szemlélet* terjedése?
  - A mellékhatások eliminálása – vagy inkább csak kordában tartása, minimalizálása,
  - a sok kis függvényből álló programszerkezet

  biztonságossá teszi a programozást.
- Ugyanezen okokból elosztott rendszerek programozására is alkalmasabbak az ilyen nyelvek az imperatív, objektum-orientált nyelveknél.
- Nem használnak közös memóriát – nincs rá szükségük –, a processzek üzeneteket küldenek egymásnak.
- Az egyes CPU-k teljesítménye nem nő drasztikusan, de nő a magok száma a számítógépeinkben – ezek között el kell osztani a munkát.

Az Erlang nyelv

- 1985: megszületik „Ellemtelben” (Ericsson–Televerket labor)
  - A név eredete: A. K. Erlang dán matematikus, ill. Ericsson language
  - 1985-86: első interpreter Prologban! **(Joe Armstrong)**
- 1991: első megvalósítás, első projektek
- 1997: első OTP (Open Telecom Platform) +
  BEAM virtuális gép (B’s – Bogdan’s, Björn’s – Erlang Abstract Machine)
- 1998-tól: nyílt forráskódú, szabadon használható
  `http://www.erlang.org/`
- Funkcionális alapú (functionally based)
- Konkurens (párhuzamos) programozást segítő (concurrency-oriented)
- Hibatűrő (fault-tolerant) – hatékony hibakezelés
- Skálázható (scalable)
- Gyakorlatban használt
  `http://en.wikipedia.org/wiki/Erlang_(programming_language)#Distribution`,
  `https://www.erlang-solutions.com/`

„Programming is fun!”

Az Elixir nyelv

- 2012: megszületik Brazíliában *(José Valim)*
- Ruby, Erlang és Closure alapokon
- Funkcionális és konkurens
- Elosztott és hibatűrő alkalmazások fejlesztésére készült
- A BEAM virtuális gépre fordít (bytecode)
- Fő jellemzői:
  - A nyelvben minden kifejezés
  - Rekurzió és magasabb rendű függvények (ciklusok helyett)
  - Mintaillesztés
  - Nincs semmi megosztva, a processzek üzenetekkel kommunikálnak
  - Teljes körű Unicode támogatás, UTF-8 sztringek, karakterláncok
  - Erlang függvények hívhatók Elixirből, Elixir függvények Erlangból
  - Metaprogramozás, polimorfizmus támogatása
  - Dokumentálás támogatása (Markdown formatting language)
  - Beépített eszközkészlet támogatja a fejlesztést

José Valim az Elixir nyelv születéséről 2014-ben 1

A couple of decades ago, memory was a very limited resource. It made sense back then for our software to take hold of some piece of memory and mutate it as necessary. However, allocating this memory and cleaning up after we no longer needed it was a very error-prone task. Some memory was never freed; sometimes memory was allocated over another structure, leading to faults. At the time, garbage collection was a known technique, but we needed faster CPUs in order to use it in our daily software and free ourselves from manual memory management. That has happened–most of our languages are now garbage-collected.

Today, a similar phenomenon is happening. Our CPUs are not getting any faster. Instead, our computers get more and more cores. This means new software needs to use as many cores as it can if it is to maximize its use of the machine. This conflicts directly with how we currently write software.

José Valim az Elixir nyelv születéséről 2014-ben 2

In fact, mutating our memory state actually slows down our software when many cores are involved. If you have four cores trying to access and manipulate the same piece of memory, they can trip over each other. This potentially corrupts memory unless some kind of synchronization is applied.

In the Erlang VM, all code runs in tiny concurrent processes, each with its own state. Processes talk to each other via messages. And since all communication happens by message-passing, exchanging messages between different machines on the same network is handled transparently by the VM, making it a perfect environment for building distributed software!

However, I felt there was still a gap in the Erlang ecosystem. I missed first-class support for some of the features I find necessary in my daily work–things such as metaprogramming, polymorphism, and first-class tooling. From this need, Elixir was born.

José Valim az Elixir nyelv születéséről 2014-ben 3

Elixir is a pragmatic approach to functional programming. It values its functional foundations and it focuses on developer productivity. Concurrency is the backbone of Elixir software. As garbage collection once freed developers from the shackles of memory management, Elixir is here to free you from antiquated concurrency mechanisms and bring you joy when writing concurrent code.

A functional programming language lets us think in terms of functions that transform data. This transformation never mutates data. Instead, each application of a function potentially creates a new, fresh version of the data. This greatly reduces the need for data-synchronization mechanisms.

All this is powered by the Erlang VM, a 20-year-old virtual machine built from scratch to support robust, concurrent, and distributed software. Elixir and the Erlang VM are going to change how you write software and make you ready to tackle the upcoming years in programming.

Forrás: Előszó Dave Thomas: *Programming Elixir* $\geq$ *1.6* című könyvéhez

Elixir-szakirodalom

Könyvek

- Dave Thomas: *Programming Elixir ≥ 1.6.*, 2018
  `https://pragprog.com/titles/elixir16/programming-elixir-1-6`
- Ulisses Almeida: *Learn Functional Programming With Elixir*, 2018
  `https://pragprog.com/titles/cdc-elixir/learn-functional-programming-with-elixir`
- Saša Jurić: *Elixir in Action*, 2024
  `https://www.manning.com/books/elixir-in-action-third-edition`

További olvasnivalók

- Írások az Elixir nyelv érdekességeiről: `https://dp.iit.bme.hu/readings.html`
- Elixir Getting Started: `https://elixir-lang.org/getting-started/introduction.html`
- Elixir Tutorial: `https://www.tutorialspoint.com/elixir/index.htm`
- Elixir Documentation: `https://elixir-lang.org/docs.html`
- Elixir School: `http://elixirschool.com/`
- Lásd még: `https://elixir-lang.org/learning.html`

Gyakorlóhely (Exercism): `https://exercism.org/tracks/elixir`
Ajánlott videók: `https://dp.iit.bme.hu/videos`