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:

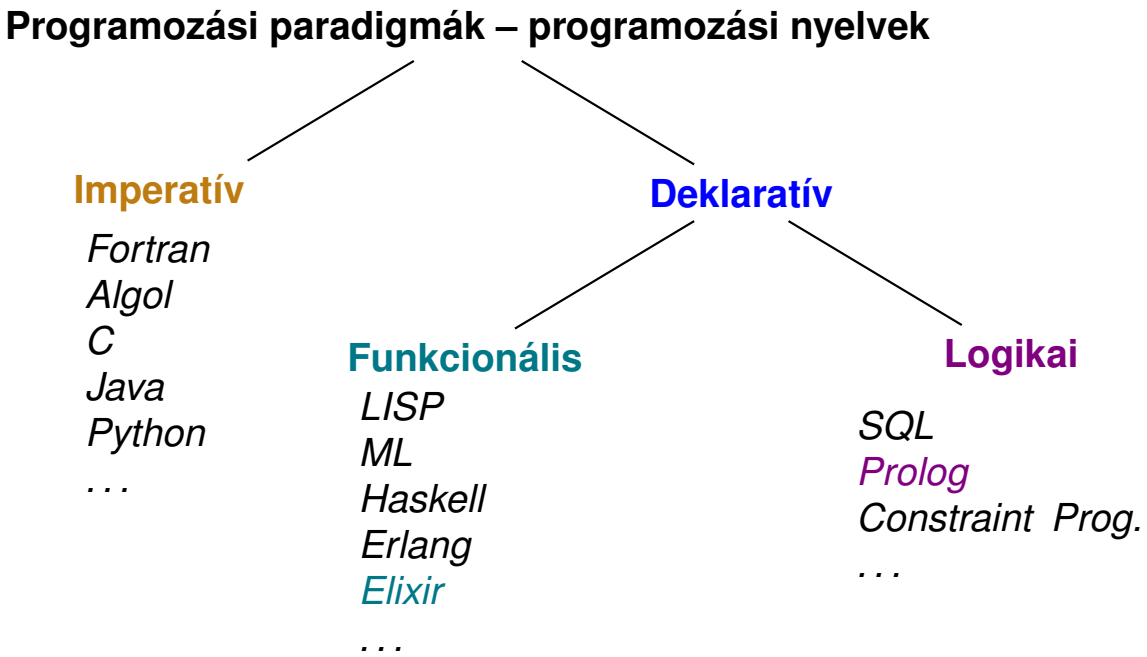
  MIT és nem HOGYAN

  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. 127. 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)>>
```

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]}
```

XI. rész

Kifejezés, őrkifejezés

Tartalom

Kifejezések

- Elég alaposan kistafíroztuk magunkat különféle könyvtári függvényekkel (ideértve az infix pozíciójú műveleteket is) ahhoz, hogy összetettebb feladatokat oldjunk meg
- Nézzük most a kifejezés különféle változatait az Elixirben
- Először is szögezzük le *ismét,* hogy az Elixirben *minden* kifejezés – az összes többi funkcionális nyelvhez hasonlóan
- A kifejezés lehet:
  - Összetett kifejezés: tetszőleges adatstruktúra, lehetnek benne elvégzendő műveletek/függvényhívások is – sok példát láttunk rá
  - Term: mint az összetett kifejezés, de *tömörnek,* tovább már nem egyszerűsíthetőnek kell lennie – erről is volt már szó
  - Szekvenciális kifejezés – ilyen példák is voltak már, de azért összefoglaljuk a dolgokat
  - Őrkifejezés – ez új téma, bár már láttunk rá példát
  - Feltételes kifejezés – ez is új, bár már láttunk rá példát

Kifejezés kiértékelése

- Funcionális nyelvek esetében mindig *kiértékelésről* (evaluation) beszélünk, sohasem *végrehajtásról*; az utóbbi használatát meghagyjuk az imperatív nyelveknek :-)
- A kifejezés *kiértékelése* alapvetően *mohó* (eager, strict) az Elixirben
- De vannak kivételek, azaz *lusta* (lazy, non-strict) kiértékelésű kifejezések, pl. kétoperandusú logikai műveletek, valamint teljes modul is: `Stream`

Vessük össze a mohó és a lusta kiértékelést az alábbi példák alapján!

```
iex> nevezo = 0
0
iex> nevezo > 0 && ( 1 / nevezo ) < 1
false
iex> (&Kernel.&&/2).( nevezo > 0, ( 1 / nevezo ) < 1) # Mi más itt?[28]
** (ArithmeticError) bad argument in arithmetic expression: 1 / 0 ...
iex> Kernel.&&( nevezo > 0, ( 1 / nevezo ) < 1) # Mi más itt?
false
iex> nevezo == 0 && ( 1 / nevezo ) < 1 # És mi más itt?
** (ArithmeticError) bad argument in arithmetic expression: 1 / 0 ...
```

[28] `&Kernel.&&/2`-ben az első `&` az ún. *capture* operátor, a `2` pedig a `Kernel.&&` függvény *aritása.*

Szekvenciális kifejezés

- Kifejezések pontosvesszővel elválasztott, opcionálisan *zárójelezett* sorozata: $\texttt{exp}_1\texttt{;}\ \texttt{exp}_2\texttt{;}\ \ldots\texttt{;}\ \texttt{exp}_n$ vagy $\texttt{(exp}_1\texttt{;}\ \texttt{exp}_2\texttt{;}\ \ldots\texttt{;}\ \texttt{exp}_n\texttt{)}$
- Kell a zárójel, ha az adott helyen egyetlen kifejezésnek kell állnia
- A szekvenciális kifejezés értéke az utolsó részkifejezés – $\texttt{exp}_n$ – értéke
- $\texttt{exp}_i$-ben ($i < n$)
  - vagy változóhoz kötünk értéket,
  - vagy mellékhatást akarunk elérni (pl. kiírunk valamit),
  - egyébként a kifejezés értéke „elvész”
- Példák:

```
iex> sumsq = fn(x,y) -> sq = fn x -> x*x end;\
                        sq.(x) + sq.(y) end; sumsq.(3,5)
34
iex>  [x=3; x+x]
** (SyntaxError) iex:1:5: syntax error before: ';'
iex> [(x=3; x+x)]
[6]
iex> "x értéke"; x
3
iex> IO.write "x értéke "; x
x értéke 3
```

Kifejezések csoportosítása

- A szekvenciális kifejezés egyfajta csoportosítást jelent
- A `do ...end` blokk a csoportosítás egy másik változata
- `do ...end` blokkot használunk pl. függvénydefinícióban, feltételes kifejezésben, for-jelölésben
- Egy `do ...end` blokk belseje egy szekvenciális kifejezés, amely állhat egyetlen kifejezésből, vagy részkifejezések pontosvesszővel vagy új sorba írással elválasztott sorozatából
- A `do ...end` blokk valójában egy olyan kulcs-érték pár alternatív írásmódja, amelyben `do:` a kulcs, az érték pedig az esetleg zárójelbe tett szekvenciális kifejezés

Példák (fájl: fpea.ex):

```
def hello_1(msg, name), do: (IO.write msg; IO.puts ", mizújs, #{name}?")
def hello_2(msg, name), do: (     # csoportosítás zárójelezéssel
    IO.write msg                  # pontosvessző helyett új sorba
    IO.puts ", mizújs, #{name}?" )
def hello_3(msg, name) do         # zárójel helyett do...end
    IO.write msg
    IO.puts ", mizújs, #{name}?" # #{tömör kif.}: ún. interpoláció
  end                            # az interpoláció sztringben használható
```

Őrkifejezés, őr 1

- Láttuk, hogy egy függvény definiálásakor a különféle esetekre önálló klózokat írunk, az eseteket pedig mintaillesztéssel ismerjük föl
- De mintaillesztéssel nem lehet megkülönböztetni minden lehetséges esetet, pl. hogy egy érték lista-e vagy atom, negatív-e vagy pozitív
- Ilyenkor `when`-nel kezdődő *őrkifejezést* (guard clause) használhatunk
- Az őrnek ***mellékhatás nélküli, hatékonyan kiértékelhető, kivételt soha nem dobó, igazságérték-eredményű*** kifejezésnek kell lennie
- Az őrkifejezésben lehet:
  - Term (tömör kifejezés)
  - Egyes modulok *guard*-ként definiált függvényei, például
    - a Kernel modul összes típust vizsgáló predikátuma (`is_TÍPUS`)
    - a Kernel modulban definiált `abs/`, `round/1`, `trunc/1`, `ceil/1`, `floor(/1`, `elem/2`, `tuple_size/1`, `hd/1`, `tl/1`, `length/1`, `in/2`, `bit_size/1`, `byte_size/1`
    - az Integer modulban definiált `is_even/`, `is_odd/1`
  - Őrökből álló olyan kifejezés, mely aritmetikai, összehasonlító, logikai (kivétel: `&&`, `||`, `!`) és konkatenáló, továbbá binárisokra / sztringekre alkalmazható műveletekkel van összerakva

Őrkifejezés, őr 2

- Őrkifejezésben tehát **nem** lehetnek
  - felhasználó által definiált függvények,
  - a modulokban *function*-ként definiált függvények,

  mert
  - mellékhatásuk lehet,
  - a kiértékelésük hatékonysága nem garantálható,
  - kivételt dobhatnak.
- Példa: faktoriális számítása (fájl: fpea.ex)
  A rekurzió soha nem áll le, ha `n` lebegőpontos vagy negatív:

```
def fac(0), do: 1
def fac(n), do: n * fac(n-1)
```

`fac_2` kivételt dob, ha `n` nem egész, vagy ha negatív:

```
def fac_2(0),                                    do: 1
def fac_2(n) when (is_integer n) and n > 0, do: n * fac_2(n-1)
```

```
iex> Fpea.fac_2 -1
** (FunctionClauseError) no function clause matching in ...
```

XII. rész

Függvény, névtelen függvény, magasabb rendű függvény

Tartalom

Függvény definiálása

- Névtelen és neves függvények definiálásával már eddig is sokszor találkoztunk – nehéz lenne úgy beszélni a funkcionális programozásról, hogy kezdettől fogva ne legyen szó függvényekről és listákról.
- Most igyekszünk mindent elmondani előbb a *névtelen*, majd a *neves* függvények definiálásáról, amit csak tudni érdemes.[29]

[29] A névtelen függvényt lambdának, lambda-függvénynek is nevezik. A λ-jelölést matematikai függvények tulajdonságainak vizsgálatához vezette be Alonzo Church az 1930-as években.

Névtelen függvény 1 (dp_fun.exs)

- A névtelen függvény, másnéven *lambda* definíciója az `fn` kulcsszóval kezdődik, egy vagy több *klózból* áll, és az `end` kulcsszóval fejeződik be:

```
fn
  klóz
  klóz ...
end
```

- A klóz *fejből* és *törzsből* áll, a kettőt -> választja el:

```
klózfej -> klóztörzs
```

- A klózfej *paraméterlistából* (mintakifejezésből) és opcionálisan *őrkifejezésből* áll:[30]

```
paraméterlista [when őrkifejezés]
```

- A klóztörzs tetszőleges szekvenciális kifejezés.
- Példa:

```
stir = # Hogy hivatkozni tudjunk a névtelen fv-re, névhez kell kötni
  fn {a, b} when a < b           -> {b, a}
     {a, b} when rem(a,b) === 0 -> d = div(a,b); {b * d, d}
     x                           -> x
  end
```

[30] A szögletes zárójelek itt az opcionálisat jelölik.

Névtelen függvény 2 (dp_fun.exs)

- A második klóz törzsében egy szekvenciális kifejezés van, amit szabad, de – ellentétben a neves függvénnyel – nem kell zárójelbe tenni:

```
...  {a, b} when rem(a,b) === 0 -> (d = div(a,b); {d * b, d})
     x                          -> x ...  # x mindenre illeszkedik
```

- A most definiált névtelen függvény túl bonyolult, rendszerint nagyon egyszerűeket és rövideket definálunk „menet közben” (on-the-fly).
- Névtelen függvényt tipikusan két esetben használunk: akkor, amikor
  - egy függvénydefinícióban lokális függvényre van szükség,
  - egy függvény paraméterként egy egyszerűbb, saját függvényt vár.
- Példák:

```
def sumsq(x, y), do: ( sq = fn x -> x*x end; sq.(x) + sq.(y) ) # Ott a . !
iex> Dp.Fun.sumsq(3, 5)
34
def sumtr(f, x, y), do: f.(x) + f.(y) # Ott a . !
iex> Dp.Fun.sumtr(fn x -> 2 * x + 1 end, 3, 5)
18
```

Névtelen függvény 3 (dp_fun.exs)

- Amikor változónevet függvényként hívunk meg, **ponttal kell elválasztani** a paraméterektől, amiket **zárójelbe kell rakni**.
- A lokális függvény paraméterei és más változói lokálisak, azokat a befogadó függvény sem látja.
- A lokális függvény használhatja az őt befogadó függvény paramétereit és más változóit.
- Példák:

```
def sumx2y2z(x, y, z), do:
  ( sq = fn x -> x*x end;
    sq.(x) + sq.(y) + z
  )
iex> Dp.Fun.sumx2y2z(3, 5, 6)
40
```

- A névtelen függvény nem rekurzív, azaz nem hívhatja önmagát.
- A kis névtelen függvények használata annyira gyakori, hogy az Elixirben *rövidítő jelölés* van a definiálásukra.

Névtelen függvény 4 (dp_fun.exs)

- Nézzük az előbbi két példát a *rövidítő jelöléssel,* továbbá a második példa egy változatát, amelyben a névtelen függvénynek három paramétere van:
- Példák:

```
def sumsq(x, y), do: (sq = fn x -> x * x end; sq.(x) + sq.(y))
def sumsq_2(x, y), do: (sq = &(&1 * &1); sq.(x) + sq.(y)) # kell a pont!
iex> Dp.Fun.sumsq_2(3, 5)
34
def sumtr(f, x, y), do: f.(x) + f.(y) # kell a pont!
iex> Dp.Fun.sumtr(fn x -> 2 * x + 1 end, 3, 5)
iex> Dp.Fun.sumtr(&(2 * &1 + 1), 3, 5)
18
iex> sumtr_2 = &(&1.(&2) + &1.(&3)) # kell a pont!
iex> sumtr_2.(&(2 * &1 + 1), 3, 5) # kell a pont!
18
```

- A névtelen függvényt változóhoz sem kell kötni, úgy is alkalmazható:

```
iex> (&(&1.(&2) + &1.(&3))).(&(2 * &1 + 1), 3, 5) # kell a pont!
18
```

Neves függvény 1 (dp_fun.exs)

- Neves függvényt csak modulban lehet definiálni `def`-fel vagy `defp`-vel.
- A `defp`-vel definiált függvény privát (azaz lokális), csak a modul más függvényeiből hívható meg, az adott modulon kívülről nem.
- Amikor egy modul függvényeiből hívunk meg olyan függvényeket, amelyek ugyanabban a modulban vannak definiálva, akkor a modul nevét nem kell – de szabad – kiírni a függvénynév elé.
- Amikor egy modulban függvénydefiníción kívül hivatkozunk egy függvényre, akkor mindig ki kell írni a modulnevet a függvénynév elé, akkor is, ha a függvény ugyanabban a modulban van definiálva.
  A modulban nem függvénydefiníciók belsejében megadott kifejezéseket az Elixir már a fordítás során futtatja.
- A neves függvény definíciója is egy vagy több *klózból* áll. *Minden* klóznak vagy `def`-fel, vagy `defp`-vel[31] kell kezdődnie:

```
def klóz
def klóz ...
```

[31] `def` és `defp` szintaxisa ugyanaz. A programszövegben egy függvény összes klózának együtt kell lenniük, a függvénydefiníciók nem keveredhetnek.

Neves függvény 2 (dp_fun.exs)

- A klóz itt is *fejből* és *törzsből* áll.
- A klózfej *névből*, *paraméterlistából* – mintakifejezésekből – és opcionálisan *őrkifejezésből* áll[32]. A paraméterlistát az egyértelműség kedvéért sokszor zárójelbe kell rakni[33]:

  ```
  név[ () | ()paraméterlista[])] [when őrkifejezés]
  ```

- A paraméterek közül az utolsó opcionális, azaz elhagyható, ha `\\` jelekkel elválasztva megadjuk az alapértelmezett értéket.

  ```
  def inspect(item, options \\ [])
  ```

- A klóztörzs a `do:`-val bevezetett tetszőleges szekvenciális kifejezés[34]:

  ```
  do: [()szekvenciális kifejezés[])]
  ```

- A klóztörzs alternatív leírási módja a `do`...`end` pár használata, ilyenkor a `do` elé *nem szabad* vesszőt rakni, a kerek zárójelek pedig elhagyhatók:

  ```
  do szekvenciális kifejezés end
  ```

- Egy függvény klózai közé más függvénydefiníciót nem szabad beékelni.

[32] A `[` és `]` ezen a dián is az opcionálisat jelölik, a `|` pedig az alternatívát.

[33] A *név* és a nyitó zárójel között nem lehet szóköz!

[34] A pontosvesszővel elválasztott vagy új sorba írt részkifejezéseket zárójelbe kell rakni.

Neves függvény 3 (dp_fun.exs)

- A függvényt a neve és paramétereinek száma (az ún. aritása) azonosítja. Ugyanazzal a névvel több különböző aritású függvény definiálható.
- A neves függvények lehetnek rekurzívak (a névtelek nem).
- Oldjuk meg a következő tanulságos feladatot!
  Válogassuk ki egy `xs` listából a *`d`-vel osztható* egészeket, rendre *szorozzuk meg őket `d`-vel, `(d+1)`-gyel stb.,* majd az így kapott elemeket adjuk vissza egy *csökkenő sorrendű* listában.
- Tudnunk kell, hogy a feltételnek megfelelő elemek közül hányadiknál járunk, mert egyesével növekvő egészekkel kell megszoroznunk.
- Ehhez egy segédfüggvényre és egy plusz paraméterre van szükségünk, hiszen globális változók nincsenek az Elixirben. A segédfüggvényt az új paraméter kezdőértékével, `d`-vel hívjuk meg:

```
def picked(xs, d), do: (picked xs, d, d)
```

- Mind a három lehetséges esetre definiálunk egy-egy klózt:
  1. a soron következő elem egész szám és `d`-vel osztható,
  2. a soron következő elem nem egész szám vagy nem osztható `d`-vel,
  3. a lista üres.

Neves függvény 4 (dp_fun.exs)

- A három klózból álló privát függvény:

```
defp picked([x|xs], d, i) when (is_integer x) and (rem x,d) === 0 do
    [ x * i | picked(xs, d, i+1) ]
end
defp picked([_x|xs], d, i), do: picked(xs, d, i)
defp picked([], _d, _i), do: []
```

- Az 1. és a 2. klóz sorrendje fontos, fordított sorrendben minden listaelemet eldobna, a `d`-vel osztható egészeket is.
- Az 1. klózban listát építünk: az eredménylista feje a soron következő elem (`x`) és a számláló (`i`) szorzata lesz, az erdeménylista farkát pedig úgy kapjuk meg, hogy a `picked` függvényt rekurzívan meghívjuk a soron következő elem utáni maradéklistára, az `i` számlálót pedig megnöveljük.
- Az 1. és a 2. klózban minden rekurzív lépésben eggyel rövidül a maradéklista. Mivel a lista korlátos, előbb-utóbb eljutunk az üres listáig, ami a rekurzió végét jelenti.
- A hatékonyságot növeli, ha az üres listára illeszkedő klóz az utolsó.
- Az eredménylista nincs rendezve! Rendezzük csökkenő sorrendben:

```
def picked(xs, d), do: Enum.sort((picked xs, d, 0), &>/2)
```
[35]

[35] `Enum.sort/2`-t a kívánt rendezési relációval kell meghívni.

Neves függvény 5 (dp_fun.exs)

- Nézzünk néhány tesztesetet, futási eredményt:

```
iex> Dp.Fun.picked([3, :a, 6, 9, 0], 3)
[45, 24, 9, 0]
iex> Dp.Fun.picked('abcdefgh', 2)
[520, 408, 300, 196]
iex> Dp.Fun.picked([:b, [], '', "", {}, %{}], 4)
[]
iex> Dp.Fun.picked(Enum.to_list(2..36 // 6), 4)
[192, 100, 32]
```

- `picked` azt csinálja, amit várunk tőle. Jó tulajdonsága, hogy csak egyszer gyalogol végig a listán. A szerkezete azonban nem elég tiszta, nem eléggé érthető.
- A feladat tulajdonképpen három részfeladatból áll:
  1. a megfelelő elemek – a `d`-vel osztható egészek – kigyűjtése a listából,
  2. a kigyűjtött elemek listáján a transzformáció elvégzése elemenként,
  3. az eredménylista rendezése.
- Az eredménylista rendezése eddig is külön lépés volt, nézzük most a másik kettőt!

Neves függvény 6 (dp_fun.exs)

- A `d`-vel osztható egészek kigyűjtése a listából `picked/3`-hoz nagyon hasonló, de eggyel kisebb az aritása, és nem végez transzformációt:

```
defp filt([x|xs], d) when (is_integer x) and (rem x,d) === 0 do
  [ x | filt(xs, d) ]
end
defp filt([_x|xs], d), do: filt(xs, d)
defp filt([], _d), do: []
filt([3, :a, 8, 6, 0], 3) === [3, 6, 0]
```

- Ha egyszer `filt/2`-vel megszűrtük a listát, az új lista minden elemét lehet transzformálni, miközben nyilvántartjuk, hányadik elemnél tartunk:

```
defp mult([x|xs], i), do: [x * i | mult(xs, i+1)]
defp mult([], _i), do: []
mult([3, 6, 0], 2) === [6, 18, 0]
```

- Már csak kombinálni kell `filt/2`-t és `mult/2`-t `Enum.sort/2`-vel:

```
def picked_2(xs, d), do: Enum.sort(mult(filt(xs, d), d), &>/2)
iex> Dp.Fun.picked_2(Enum.to_list(2..36 // 6), 4)
[192, 100, 32]
```

- Csak három függvényt kombináltunk, mégis nehezen olvasható a kód, mert sok a zárójel, és mert belülről kifelé haladva kell olvasni.

Neves függvény 7: |> operátorral (dp_fun.exs)

- Segédváltozók bevezetésével átláthatóbbá, érthetőbbé tehetjük:

```
def picked_3(xs, d) do
  fs = filt(xs, d)
  ms = mult(fs, d)
  Enum.sort(ms, &>/2)
end
```

- *A funkcionális programokra jellemző, hogy sok kis függvény egyszerű transzformációt végez, melyhez adatokat vesz át más függvénytől, majd ad át más függvénynek.*
- A |> (pipe) operátorral elkerülhetjük az egymásba skatulyázást és új változók bevezetését:

```
def picked_4(xs, d) do
  xs
  |> filt(d)
  |> mult(d)
  |> Enum.sort(&>/2)
end
```

```
iex> Dp.Fun.picked_4(Enum.to_list(2..36 // 6), 4)
[192, 100, 32]
```

Neves függvény 8: |> operátorral (dp_fun.exs)

- A |> operátorral még akkor is jól olvasható a kód, ha egyetlen sorba írjuk a transzformációsorozatot:

```
def picked_4(xs, d), do: xs |> filt(d) |> mult(d) |> Enum.sort(&>/2)
```

- Az infix |> operátor a bal oldali kifejezés értékét adja át a jobb oldali függvénynek első paraméterként – így azt nem kell, *nem is szabad* kiírni
- A |> operátor jobb oldalán a függvény további paramétereit vagy a teljes függvényhívást zárójelbe kell rakni, mert a |> mohó kiértékelésű. A fenti sor így is zárójelezhető:

```
... do: xs |> (filt d) |> (mult d) |> (Enum.sort &>/2)
```

- Térjünk vissza még a `filt/2`, majd a `mult/2` függvényhez!
- Mindkettő túl speciális: az elsőbe a szűrőfeltétel, a másodikba az elvégzendő művelet van „gránitszilárdsággal” beépítve. Ha ezeket függvényparaméterként adnánk át, rugalmasabb lenne a megoldásunk.

Neves függvény 9: `filt` újra (dp_fun.exs)

- Ez itt a `filt/2` első, specifikus verziója:

```
defp filt([x|xs], d) when (is_integer x) and (rem x,d) === 0 do
  [ x | filt(xs, d) ]
end
defp filt([_x|xs], d), do: filt(xs, d)
defp filt([], _d), do: []
```

- Az őrt cseréljük le függvényhívásra – a függvényt paraméterként adjuk át:

```
defp filt_2([x|xs], f?) when f?(x), do: [ x | filt_2(xs, f?) ]
defp filt_2([_x|xs], f?), do: filt_2(xs, f?)
defp filt_2([], _f?), do: []
```

  Az `f?` névben a kérdőjellel arra utalunk, hogy a függvény igazságértéket ad eredményül – azaz predikátum.

- Van itt egy kis probléma: a `filt_2/2` fordításakor az IEx hibát jelez
  ** (CompileError) . . . anonymous call is not allowed in guards.
- Idézzük föl: a `when` őrkifejezést vezet be, őr csak könyvtári függvény lehet, azokból se mind. Az `f?` paraméterről nem tudható, mi lesz az aktuális értéke, ezért viselkedik ilyen elutasítóan az Elixir.

Neves függvény 10: `filt_2` és `oper` (dp_fun.exs)

- Őrkifejezés helyett tehát őr nélküli *feltételes kifejezést* kell használnunk, összevonva a korábbi 1. és 2. klózt:

```
defp filt_2([x|xs], f?), do:
  (if f?.(x), do: [ x | filt_2(xs, f?) ], else: filt_2(xs, f?))
defp filt_2([], _f?), do: []
```

- `mult/2` helyett írjunk egy általánosabb `oper/2`-t:

```
defp oper([x|xs], f), do: [ f.(x) | oper(xs, f) ]
defp oper([], _f), do: []
```

  Ez nem jó, mert így nem követjük, hányadik listaelemnél is tartunk, márpedig x-szel és folyton változó másik értékkel kell műveletet végezni.

- `oper/2` második paramétere legyen akkor az `{f,i}` pár:

```
defp oper([x|xs], {f, i}), do: [ f.(x, i) | oper(xs, {f, i+1}) ]
defp oper([], _fi), do: []
```

  Ez sem elég általános; később megnézzük, lehet-e általánosabb megoldást írni ilyen jellegű feladatokra.

- A következő dián összerakjuk a teljes megoldást az új részmegoldásokkal.

Neves függvény 11: `filt_2` és `oper` (dp_fun.exs)

- A teljes megoldás új verziója az új részmegoldásokkal

```
def picked_5(xs, d) do
  f? = &(is_integer &1) and (rem &1, d) === 0
  f  = &(&1 * &2)
  xs
  |> filt_2(f?)
  |> oper(f, d)
  |> Enum.sort(&>/2)
end
iex> Dp.Fun.picked_5(Enum.to_list(2..36 // 6), 4)
[192, 100, 32]
```

  Itt két névtelen két segédfüggvényt definiáltuk lokálisan, az `f?` és az `f` névhez kötve őket, de egy újabb verziójú `picked` függvény paraméterei is lehetnének. Ezzel a megoldásunk még flexibilisebb lenne.
- Az olyan függvényt, amelynek a paraméterei között függvényérték is van, *magasabb rendű* függvénynek hívjuk. Függvény eredménye is lehet függvényérték, ilyenkor is magasabb rendű függvényekről beszélünk.
- Következő témánk a magasabb rendű függvények.

Tartalom

Magasabb rendű függvények: `map/2` és `filter/2` (dp_fun.exs)

- Magasabb rendű függvény: paramétere vagy eredménye függvény.
- A magasabb rendű függvények általánosított eszközök bonyolultabb feladatok egyszerű megoldására.
- *Magasabb rendű függvények használatával a rekurzió többnyire rejtve lesz csak jelen a programjainkban.*
- Az előbb definiált `oper/2` és `filt/2` a `map/2`, ill. a `filter/2` „előfutára”.
- `map/2` egy lista elemeit transzformálja, `filter/2` szűréssel kiválogatja az adott feltételnek megfelelő elemeket a listából:

```
def map([x|xs], f), do: [ f.(x) | map(xs, f) ]
def map([], _f),    do: []

def filter([x|xs], p?), do:
       (if p?.(x), do: [ x | filter(xs, p?) ], else: filter(xs, p?))
def filter([], _p?),    do: []
```

- Működésük megértéséhez már nincs szükség magyarázatra.[36]
- „Hivatalos” verziójuk az Elixirben: `Enum.map/2` és `Enum.filter/2`, tetszőleges felsorolható kollekcióra alkalmazhatók.

[36] *Fejtörő:* miért érdemes kétszer leírni a `filter(xs, p?)` hívást `filter/2` definíciójában?

Egyszerű példák `map/2`-vel és `filter/2`-vel

```
iex> Enum.map([ "alma", "korte" ], &String.length/1)
[4, 5]
iex> Enum.map([ [10, 20], [10, 20, 30] ], &Enum.sum/1)
[30, 60]
iex> for s <- ["alma", "korte"], do: String.length(s) # map helyett
[4, 5]
```

Amikor `perms/1`-et a `'tér'` karakterlistára alkalmaztuk, számlistát kaptunk eredményül (lásd 194. dia). Ezt `map/2`-vel könnyű sztringek listájává konvertálni:[37]

```
iex> (Fpea.perms ~c"tér") |> (Enum.map &List.to_string/1)
["tér", "tré", "étr", "ért", "rté", "rét"]
```

```
iex> Enum.filter([:x, 10, L, 20, {}], &is_number/1)
[10, 20]
iex> Enum.filter([:x, {7,3}, 10, L, 20, {}], &is_tuple/1)
[{7, 3}, {}]
iex> for i <- 10..50, rem(i,7) === 0, do: i # filter helyett
[14, 21, 28, 35, 42, 49]
iex> Enum.to_list 14..50 // 7 # az előző egyszerűbben, szűrés nélkül
[14, 21, 28, 35, 42, 49]
```

[37] A |> operátor túl nagy precedenciájú, ezért kell itt zárójelbe tenni a bal oldali argumentumát.

Magasabb rendű függvények: `foldr/3` és `foldl/3`

- Gyakran van szükség listákon vagy más kollekciókon aggregáló, redukáló műveletek elvégzésére, pl. egy lista elemeinek összeadására, összeszorzására, egy lista összes elemének egy másik elé fűzésére.
- Ilyenkor a listaelemeken tipikusan valamilyen kétoperandusú műveletet kell végrehajtani, pl. 1+3+6+10+15+21
- El lehet végezni balról jobbra vagy jobbról balra haladva:
  `(((((0+1)+3)+6)+10)+15)+21 == 56`, `(((((0+21)+15)+10)+6)+3)+1 == 56`
  `(((((0-1)-3)-6)-10)-15)-21 == -56`, `(((((0-21)-15)-10)-6)-3)-1 == -56`
- A kétféle sorrendet megvalósító függvényt `foldl`-nek, ill. `foldr`-nek szokás hívni (a másodikat `reduce`-nak is); a specifikációjuk azonos:
  @spec fold(xs::[any()], acc::any(), f::((x::any(), y::any()) -> r::any())) :: rs::[any()]*
- `foldl` balról jobbra, `foldr` jobbról balra halad végig a listán.
- Az `f` függvényt, ahol a 2. paraméternek kell az `acc` akkumulátornak lennie, minden lépésben a soron következő listaelemre és az akkumulátorra alkalmazzák.
- Nem asszociatív műveletek esetén `f`-ben a műveleti sorrend fontos! Pl. `fn(x,acc)-> acc-x end` vagy `fn(x,acc)-> x-acc end` más lesz a `fold` függvények eredménye.

`foldr/3` és `foldl/3` [38] definíciója és használata (dp_fun.exs)

R

```
def sumr([x|xs], acc), do:
      (&+/2).(x, sumr(xs, acc))
def sumr([], acc),     do: acc
```

```
def foldr([x|xs], acc, fun), do:
      fun.(x, foldr(xs, acc, fun))
def foldr([], acc, _fun),    do: acc
```

L

```
def suml([x|xs], acc), do:
      suml(xs, (&+/2).(x, acc))
def suml([], acc),     do: acc
```

```
def foldl([x|xs], acc, fun), do:
      foldl(xs, fun.(x, acc), fun)
def foldl([], acc, _fun),    do: acc
```

- Példák

```
iex> Dp.Fun.foldr([1,2,3,4], 1, &*/2) # 1*(2*(3*(4*1)))
24
iex> Dp.Fun.foldl([1,2,3,4], 1, &*/2) # 4*(3*(2*(1*1)))
24
iex> Dp.Fun.foldr([1,2,3,4], 0, &-/2) # 1-(2-(3-(4-0))), fn(x,acc)-> x-acc end
-2
iex> Dp.Fun.foldl([1,2,3,4], 0, &-/2) # 4-(3-(2-(1-0))), fn(x,acc)-> x-acc end
2
iex> Dp.Fun.foldr([1,2,3,4], 0, fn(x,acc)-> acc-x end) # (((0-4)-3)-2)-1
-10
iex> Dp.Fun.foldl([1,2,3,4], 0, fn(x,acc)-> acc-x end) # (((0-1)-2)-3)-4
-10
```

[38] Az Elixirben: `List.foldr/3`, `List.foldl/3`, továbbá `Enum.reduce/2`, `Enum.reduce/3`.

További példák `foldr/3` és `foldl/3` használatára

Példák listák összefűzésére

```
iex> {is, as} = {~c"indul", ~c"aludni"}
{~c"indul", ~c"aludni"}
iex> List.foldr(is, as, &([&1|&2])) # [?i|[?n|[?d|[?u|[?l|~c"aludni"]]]]]
~c"indulaludni"
iex> List.foldl(is, as, &([&1|&2])) # [?l|[?u|[?d|[?n|[?i|~c"aludni"]]]]]
~c"ludnialudni"
iex> List.foldl(is, [], &([&1|&2])) # [?l|[?u|[?d|[?n|[?i|[]]]]]]
~c"ludni"
```

- `&([&1|&2]` a *listakonstruktorból* csinál névtelen függvényt: a második (lista-)paraméter elé fűzi az elsőt.
- Mivel `foldr` az első listaparaméter utolsó karakterével kezd, az 1. listát az *eredeti sorrendben* fűzi a 2. elé: eredménye azonos `append/2`-évél.
- `foldl` viszont az első listaparaméter első karakterével kezd, ezért az 1. listát *fordított sorrendben* fűzi a 2. elé: `revapp/2` a szokásos neve, az Elixirben `Enum.reverse/2` néven található meg.
- Ha `foldl` második paramétere az üres lista, akkor megfordítja az első listát: eredménye azonos `Enum.reverse/1`-évél.

Záró megjegyzések a magasabb rendű függvényekről

- Amikor funkcionális nyelven egy listán pl. valamilyen összegzést végzünk, az imperatív nyelvekben megszokott *ciklus* helyett *rekurziót* használunk.
- Az alábbi jól ismert példában a rekurzió *explicit*, azaz megjelenik a programszövegben, a `sum/1` meghívását a futtatórendszer *rekurzív processzként* futtatja – hacsak a fordítóprogram át nem alakítja ciklussá:

```
def sum([]), do: 0
def sum([x|xs]), do: x + sum xs
iex> Dp.Fun.sum [1,2,3,4,5]
15
```

- Ha ugyanezt a feladatot pl. `List.foldl/3`-mal oldjuk meg, a programszöveg nem tartalmaz rekurziót:

```
iex> List.foldl([1,2,3,4,5], 0, &(&1 + &2))
15
```

- Lehet, hogy a futtatott kód rekurzív, lehet, hogy nem, nem tudhatjuk.
- Mindenesetre a kódot sokkal könnyebb megérteni, sem programozóként, sem kódolvasóként nem kell a rekurzív megoldás megfejtésével bajlódnunk.

XIII. rész

Feltételes kifejezés

Tartalom

Feltételes kifejezés

- A funkcionális nyelvű programok sok kis függvényből állnak, a meghívásuk sorrendjét, a klózok kiválasztását mintaillesztéssel és esetleg őrökkel oldjuk meg
- Néha mégis szükség van olyan vezérlési szerkezetekre, amelyekhez hasonlók gyakoriak az imperatív nyelvekben: *feltételes kifejezésekre*
- A funkcionális nyelvek vezérlési szerkezetei tehát, nem meglepő módon, maguk is *kifejezések:* van értékük, *teljes jogú polgárok*[39]
- Abból, hogy **a vezérlési szerkezet kifejezés,** következik, hogy (a kivételdobástól eltekintve) bárhogyan is érjen véget a kiértékelése, **valamilyen értéket mindenképpen eredményül kell adnia**
- Az Elixirben a feltételes kifejezést az `if`, `unless`, `cond` és `case` kulcsszavak egyike vezeti be – ezekről lesz szó a következőkben
- Lehetőleg ne vagy csak ritkán, jól megfontoltan használjunk feltételes kifejezést mintaillesztés és őrök helyett: a feltételes kifejezéstől a függvény hosszabb, olvashatatlanabb lesz – szükségtelenül
- Ökölszabályként jegyezzük, meg, hogy egy 15-20 sornál hosszabb klóz valószínűleg arra utal, valamit nem elég jól kódoltunk

[39] First-class citizens

`if` és `unless` 1

- Két *paraméterük* van: egy feltétel és egy kulcs-érték lista, amelyben két kulcs van: `do:` és `else:`
- Ha a feltétel igaz, az `if` a `do:`-hoz tarozó értéket, ha hamis, akkor az `else:`-hez tartozót adja eredményül; az `unless` pont fordítva csinálja
- Az `else:` ág, azaz az `{:else, érték}` pár elhagyható, ilyenkor az eredmény `nil`, ha a feltétel `if` esetén hamis, `unless` esetén igaz
- `do: ..., else: ...` helyett `do ... else ... end` is írható
- Példák:

```
iex> if(true && 1, [{:do, "hm"}, {:else, "mh"}]) # függvényszerű
"hm"
iex> if false, do: "hm", else: "mh" # de van egyszerűbb írásmódja is
"mh"
iex> if false || 1, do: "hm" # az else: ág elhagyható [40]
"hm"
iex> if false, do: "hm" # ekkor nil az eredmény, ha a feltétel false
nil
iex> unless false, do: "hm", else: "mh" # unless fordítva csinálja
"hm"
```

[40] Ez nem szép ;-) az Elixirtől: a funkcionális nyelvekben nem szokás valamelyik ág elhagyása.

`if` és `unless` 2

- Az `if` és az `unless` nem nyelvi elem, hanem *makró*[41]
- Az `if`-ben és az `unless`-ben a feltétel igazságértékként (*truthy*, *falsy*) értelmezhető, egyébként tetszőleges kifejezés lehet
- Az `if`-ben és az `unless`-ben definiált változók *lokálisak*
- Példa:

```
iex> x = 1
1
iex> if true, do: (x = x + 1; x)
2
iex> x
1
```

- Ezért ha értéket akarunk kötni egy változóhoz, akkor ki kell használnunk, hogy `if` és `unless` kifejezések, van értékük:

```
iex> x = 1
1
iex> x = unless true, do: x + 1, else: x + 2
3
```

[41] A makrót fordításkor fejti ki az Elixir. További gyakran használt makrók: `def`, `defp`, `defmodule`, `with`.

`cond` (fájl: fpea.ex) 1

- A `cond` feltétel-érték párok sorozatát dolgozza fel
- A `cond`-ban – mint `if`-ben és az `unless`-ben – igazságérték-eredményű, egyébként tetszőleges függvények hívhatók meg a feltételben
- Mint minden kifejezés kiértékelése, a feltételes kifejezésé is balról jobbra, fölülről lefelé halad
- Az **első** teljesülő feltételhez tartozó kifejezés értéke lesz az eredmény
- Ha egyik feltétel sem teljesül, a virtuális gép kivételt dob
- `to_decval_cond` eredménye egy hexadecimális számjegy decimális értéke; kivételt dob, ha nem hexadec számjeggyel hívjuk meg

```
def to_decval_cond(hexval) do
   cond do
     ?0 <= hexval &&  hexval <= ?9 -> hexval - ?0
     ?a <= hexval and hexval <= ?f -> hexval - ?a + 10
   end
 end
iex> Fpea.to_decval_cond(?f)
15
```

`cond` (fájl: fpea.ex) 2

- A kivételdobást elkerülhetjük a `nil` érték visszaadásával:

```
        ... ?a <= hexval and hexval <= ?f -> hexval - ?a + 10
            true                          -> nil ...
iex> Fpea.to_decval_cond2(0)
nil
```

- vagy az Erlangban szokásos megoldással:

```
        ... ?0 <= hexval && hexval <= ?9 -> {:ok, hexval - ?0}
            ?a <= hexval && hexval <= ?f -> {:ok, hexval - ?a + 10}
            true                         -> :error ...
iex> Fpea.to_decval_cond3(?z)
:error
```

- A `fpea.ex` fájlban van a három változat `to_decval_cond`, `to_decval_cond2` és `to_decval_cond3` néven

case (fájl: fpea.ex)

- A case adott **értéket illeszt mintákra,** ezek kiegészíthetők őrkifejezéssel

```
def to_decval_case(hexval) do
  case hexval do
    hv when ?0 <= hv and hv <= ?9
       -> hv - ?0
    ?a -> 10
    ?b -> 11
    ?c -> 12
    ?d -> 13
    ?e -> 14
    ?f -> 15
    _  -> nil
  end
end
```

- Hangsúlyozzuk, hogy bár vannak feltételes kifejezések az Elixirben (ahogy a többi funkcionális nyelvben is), célszerű helyettük, ahol csak lehet, mintaillesztést használni, esetleg őrrel kiegészítve
- A mintaillesztést és esetleg őrt használó klózokkal elegánsabban, átláthatóbban oldható meg az ilyen jellegű feladatok nagy része

`cond` helyett `case`? (fájl: fpea.ex)

- A kiértékelendő kifejezés kiválasztására
  - a `case` kifejezést illeszt mintákra, esetleg őrkifejezéssel
  - a `cond` tetszőleges igazságérték-eredményű függvényt elfogad
- `cond` helyett tehát akkor jó a `case`, ha a választást őrökkel oldjuk meg

```
def to_decval_cond(hexval), do:
    (cond do
       ?0 <= hexval and  hexval <= ?9 -> hexval - ?0
       ?a <= hexval and hexval <= ?f -> hexval - ?a + 10
     end)
def to_decval_case2(hexval), do:
     (case hexval do
        _ when ?0 <= hexval and hexval <= ?9 -> hexval - ?0
        _ when ?a <= hexval and hexval <= ?f -> hexval - ?a + 10
        _                                    -> nil
      end)
iex> Fpea.to_decval_case2(?c)
12
```

- De minek bonyolítsuk a dolgot? ;-) Akkor használjunk `case`-t, ha mintaillesztéssel gyorsítani tudjuk az esetszétválasztást.

with 1 (fájl: fpea.ex)

Egy case-lánc, pl. hibakezelés esetén, nehezen átláthatóvá válhat.

```
def guess_case(val) do

  integer?  = fn(v when is_integer(v)) -> {v, :ok}; _ -> :non_int end
  positive? = fn(v when v > 0) -> {v, :ok}; _ -> :non_pos end
  in_range? = fn(v when 6 <=v and v <= 18) -> {v, :ok}; _ -> :out end

  case integer?.(val) do
  {v, :ok} ->
      case positive?.(v) do
        {v, :ok} ->
          case in_range?.(v) do
            {v, :ok} -> IO.puts("#{v}: bingó!")
            :out     -> IO.puts("#{v}: out of range.")
          end
        :non_pos -> IO.puts("#{v}: wrong number.")
      end
    :non_int -> IO.puts("#{val}: wrong value.")
  end
end
```

`with` 2 (fájl: fpea.ex)

Ilyen esetekben hasznos a `with` makró, ami elkülöníti az adatfeldolgozást a hibakezeléstől. Hasznos olyankor is, amikor a `|>` nem használható függvények láncolására. (Továbbiak:
`https://dev.to/martinthenth/using-elixirs-with-statement-5e36`

```
def guess_with(val) do

  integer?  = fn(v when is_integer(v)) -> {v, :ok}; _ -> :non_int end
  positive? = fn(v when v > 0) -> {v, :ok}; _ -> :non_pos end
  in_range? = fn(v when 6 <=v and v <= 18) -> {v, :ok}; _ -> :out end

  with {v, :ok} <- integer?.(val),
       {v, :ok} <- positive?.(v),
       {v, :ok} <- in_range?.(v) do
    IO.puts("#{v}: bingó!")
  else
    :non_int -> IO.puts("#{val}: wrong value.")
    :non_pos -> IO.puts("#{val}: wrong number.")
    :out     -> IO.puts("#{val}: out of range.")
  end
end
```

Emlékeztető 1: `.ex` és `.exs`

- Az `fpea.ex` fájlt az `iex fpea.ex` paranccsal tölthetjük be, vagy ha már fut az IEx, akkor a `c "fpea.ex"`-szel

```
iex> c "fpea.ex"
[Fpea]
```

- A betöltött modul(ok) nevét az IEx egy listában írja ki, most: `[Fpea]`.
- Az `.ex` fájlban egy vagy több modulnak kell lennie, az Elixir ezeket a belső alakra fordítja le, és úgy értelmezi. **Függvénydefiníció csak modulban lehet.**
- Ha `elixirc`-vel fordítunk egy fájlt, minden modulja külön `*.beam` fájlt hoz létre.
- Az `iex` az elérhető `.beam` fájlokat automatikusan betölti, ha hivatkozunk rájuk.
- A `.exs` arra utal, hogy a fájlban egy Elixir szkript van, amit az IEx nem fordít le, hanem közvetlenül interpretál.
- Egy Elixir szkriptben nem kell moduldefiníciónak lennie (de lehet benne).
- Az IEx azonnal kiértékeli a betöltött szkriptet.

Emlékeztető 2: `.ex` és `.exs`

- ```
  #def test(:cond) do # Csak modulban lehet függvénydefiníció.
    Fpea.to_decval_cond(?f)  |> IO.inspect()
    Fpea.to_decval_cond2(?6) |> IO.inspect()
    Fpea.to_decval_cond3(?z) |> IO.inspect()
  #end

  #def test(:case) do # Csak modulban lehet függvénydefiníció.
    Fpea.to_decval_case(?f)  |> IO.inspect()
    Fpea.to_decval_case2(?z) |> IO.inspect()
    #end
  iex> c "test_hexval.exs"
  15
  6
  :error
  ...
  []
  ```
- `test_hexval.exs`-ben nincs moduldefiníció (bár lehetne), ezért marad üresen a betöltés után a modulnevek listája.
  ```

XIV. rész

For-jelöléssel, egyszerűen

Tartalom

Mire jó a for-jelölés: `freq`

- A for-jelölés jó arra, hogy egy lista elemein végigmenjen.
- Egyszerre egynél több listaelemet csak „trükközve” tud feldolgozni.
- Magasabb rendű függvényekkel együtt „csodákra képes”.
- Nézzük ezt a feladatot: hányszor fordul elő egy adott elem egy listában?

```
@spec freq(val::any(), ls::[any()]) :: n::integer()
# ls-ben a val értékű elemek száma n
def freq(val, ls), do:
      (for x <- ls, x === val, do: x) |> length()
iex> Fpea.freq ?a, ~c"almafa"
3
```

- Ugyanez a `reduce:` módosítóval:

```
def freq_r(val, ls), do:
      (for x <- ls, x === val, reduce: 0, do: (acc -> 1 + acc))
iex> Fpea.freq_r ?a, ~c"almafa"
3
```

Mire jó a for-jelölés: smallers

- Ha két, esetleg több szomszédos listaelemet egyszerre kell kezelni, akkor a a szomszédos listaelemekből enneseket rakhatunk össze.
- Példa: gyűjtsük ki egy listából azokat az elemeket, melyek a közvetlenül előttük álló elemeknél kisebbek!
- Ehhez olyan párokat képezünk az Enum.zip/2 függvénnyel, amelyek a lista 0. és 1., 1. és 2., 2. és 3., s.í.t. elemeiből állnak.

```
@spec smallers(xs::[integer()]) :: ss::[integer()]
# ss azoknak a xs-beli egészeknek a listája, melyek a
# közvetlenül előttük álló listaelemnél kisebbek
def smallers([_|xs]=xxs) do
  for {p, c} <- Enum.zip(xxs ,xs), p > c, do: c
end
def smallers([]), do: []
iex> Fpea.smallers [9,8,8,8,6,7,6,5,6,7,2,1]
[8, 6, 6, 5, 2, 1]
```

Mire jó a for-jelölés: peaks

- Következő példánk a lokális csúcsokat gyűjti ki egy listából.
- Egy csúcs akkor lokális, ha nagyobb mindkét szomszédjánál.
- Itt is alkalmazhatjuk az előző trükköt, azaz képezhetünk hármasokat a lista szomszédos elemeiből, de most az Enum modul egy másik függvényét, az Enum.chunk_every/4-et fogjuk alkalmazni.

```
@spec peaks(xs::[integer()]) :: ps::[integer()]
# az xs-beli lokális csúcsok listája ps
# egy csúcs lokális, ha mindkét szomszédjánál nagyobb
def peaks(xs) do
  triads = Enum.chunk_every(xs, 3, 1, :discard)
  for [p, c, n] <- triads, p < c and n < c, do: c
end
```

```
iex> Fpea.peaks [9,8,8,8,6,7,6,5,6,5,2,1]
[7, 6]
```

- Enum.chunk_every(list, count, step, :discard)-ban a paraméterek jelentése: list a feldolgozandó lista; count a kigyűjtendő sorozatok hossza; step a következő kiszedendő sorozat kezdete az előző kezdetéhez képest; :discard használata esetén az utolsó sorozatot csak akkor adja vissza, ha az pontosan count hosszúságú.

Lokális csúcsok gyűjtése Enum.at/2-t használó for-jelöléssel

- A lokális csúcsok keresése és a hasonló feladatok megoldhatók a hagyományos tömbös szemlélettel, indexeléssel, de ez idegen a funkcionális stílustól, és nagy listák esetén a hatékonysága is kérdéses.
- Néha – például a take, drop, split, slice és hasonló függvények használatakor – szükség van a for-jelölésben az imperatív nyelvek ciklusváltozójához hasonló futóindexre. Nézzünk most erre egy példát.
- Az alábbi **ellenjavallt** példa az Enum.at/2-t használja csúcskeresésre.

```
def peaks2(xs) do
  for i <- 1..length(xs)-1, curr = Enum.at(xs, i),
    Enum.at(xs, i-1) < curr and Enum.at(xs, i+1) < curr, do: curr
end
```

Lokális csúcsok gyűjtése: hatékonyságvizsgálat benchee-vel

- Hasonlítsuk össze a kétféle megoldás futási idejét a benchee-vel.

```
randlist = for _ <- 0..5000, do: Enum.random(0..9)
Benchee.run(%{
  "peaks chunk_every/4-gyel" => fn -> randlist |> Fpea.peaks() end,
  "peaks at/2-vel"           => fn -> randlist |> Fpea.peaks2() end,
}# , profile_after: true)
```

```
Name                              IPS
peaks chunk_every/4-gyel        1.58 K
peaks at/2-vel                0.0130 K - 121.63x slower +76.55 ms
```

- Már egy 5000 elemű listára is hatalmas a különbség, több mint 120-szoros!
- 100 elemre még csak 3-szor, 20 000 elemre viszont már jóval több mint 300-szor lassabb az Enum.at/2-t használó változat. Aki nem hiszi, járjon utána! :-)
- Azaz nemcsak a funkcionális stílus, hanem a hatékonyság érdekében is érdemes kerülni az Enum.at/2-t.

Mire jó a for-jelölés: plains

- Következő példánk a maximális hosszúságú, legalább kételemű platókat gyűjti ki egy listából.
- Itt már nem működik a trükk az Enum.zip-pel, mert nem egyforma hosszú sorozatokat kell kiszednünk.
- Saját rekurzív függvény írása helyett használjuk az Enum.chunk_by/2-t.

```
@spec plains(xs::[integer()]) :: ps::[integer()]
# az xs lista azonos értékű, maximális hosszúságú,
# legalább kételemű sorozatainak, azaz platóinak listája ps
def plains(xs) do
  plains = Enum.chunk_by(xs, & &1)
  for ps = [_,_|_] <- plains, do: ps
end
```

```
iex> Fpea.plains [9,0,0,0,6,7,6,5,5,2,2,2,2,1]
[[0, 0, 0], [5, 5], [2, 2, 2, 2]]
```

- Enum.chunk_by(list, fn) azoknak a listaelemeknek a listáját adja eredményül, amelyekre az fn függvény az őket közvetlenül megelőző listalemre azonos eredményt adott.
- A példában az & &1 ún. identitásfüggvényt használjuk; írhatjuk &(&1) vagy fn(x) -> x end alakban is.

Mire **nem** jó a for-jelölés: `slope_lengths`, `slopes`

- Az előző példában a feladatot valójában az `Enum.chunk_by/2` oldotta meg, a for-jelölést csak a kettőnél rövidebb részlisták kiszűrésére használtuk – mégpedig mintaillesztéssel, hatékonyan.
- Nézzük a következő specifikációt!

  @type cmp() :: integer(), integer() :: boolean()
  @spec slope_lengths(xs::[integer()], cmp::cmp()) :: ls::[integer()]
  # az xs listából cmp alapján kigyűjtött, maximális hosszúságú, legalább kételemű
  # sorozatok – lejtők, emelkedők vagy platók – hosszának listája ls
- Ebben a feladatban bizonyos kritériumok szerinti részlisták hosszát kell meghatározni; ehhez hasonló feladatokban magukat a részlistákat kell előállítani. Ilyen feladatok megoldására a for-jelölés már nem alkalmas.
- Gyakoriak az ilyen jellegű feladatok, ezért az `Enum` modulban van megfelelő függvény, az `Enum.chunk_while/4`; a for-jelölés nem is kell hozzá.
- A megoldást meghagyjuk gyakorló feladatnak. Egy példa a valódi lejtők – `[9,0]`, `[7,6,5]`, `[5,2]`, `[2,1]` – hosszának kiszámítására.

```
iex> Slopes.slope_lengths [9,0,0,0,6,7,6,5,5,2,2,2,2,1], &</2
[2, 3, 2, 2]
```

Pitagoraszi számhármasok listája 1

- Könnyen átlátható, de nem hatékony megoldás

```
@spec pitag1(n::integer()) :: ps::[{integer(), integer(), integer()}]
# az n-nél nem nagyobb összegű pitagoraszi számhármasok listája ps
def pitag1(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}
end
```

```
iex> Fpea.pitag1 12
[{3, 4, 5}]
iex> Fpea.pitag1 36
[{3, 4, 5}, {5, 12, 13}, {6, 8, 10}, {9, 12, 15}]
iex> Fpea.pitag 1500
[{3, 4, 5}, {5, 12, 13}, ..., {30, 72, 78}, {30, 224, 226}, ...]
```

- Hogyan lehetne javítani a hatékonyságán? Leginkább úgy, hogy kevesebb *jelöltet* állítunk elő.

Pitagoraszi számhármasok listája 2

- Hatékonyabb megoldás a tartományok egyszerű szűkítésével

```
def pitag2(n), do:
  for a <- 1..n,
      b <- a..n, # b nem lehet kisebb a-nál
      c <- b..n, # c nem lehet kisebb b-nél
      a + b + c <= n,
      a * a + b * b === c * c,
    do: {a, b, c}
end
iex> Fpea.pitag2 36
[{3, 4, 5}, {5, 12, 13}, {6, 8, 10}, {9, 12, 15}]
iex> Fpea.pitag2 1500
[{3, 4, 5}, {5, 12, 13}, ..., {30, 72, 78}, {30, 224, 226}, ...]
```

- A hosszú listák sok helyet foglalnak el a veremben, az összehasonlításuk – mind az eredmény helyességét, mind a futási időt tekintve – nehezebb, ezért a pitagoraszi számhármasok helyett csak a számukat határozzuk meg, adjuk eredményül; lásd a következő diákat. A függvények közös specifikációja:

```
@spec pitag*cnt*(n::integer()) :: cnt::integer()
# az n-nél nem nagyobb összegű pitagoraszi számhármasok száma cnt
```

Pitagoraszi számhármasok száma 1

Ez a dia egy naív és három javított hatékonyságú megoldást mutat be.

```
def pitag1cnt(n), do:
  ls = 1..n
  for a <- ls,
      b <- ls,
      a <= b,
      c <- ls,
      a + b + c <= n,
      a * a + b * b === c * c,
      reduce: 0,
    do: (acc -> 1 + acc)
end
def pitag2cnt1(n), do:
    for a <- 1..n,
      b <- a..n, # b >= a
      c <- b..n, # c >= b
      a + b + c <= n,
      a * a + b * b === c * c,
      reduce: 0,
    do: (acc -> 1 + acc)
end
```

```
def pitag2cnt2(n), do:
    for a <- 1..n,
      a2 = a * a, # Javít-e?
      b <- a..n,
      b2 = b * b, # Javít-e?
      c <- b..n,
      a + b + c <= n,
      a2 + b2 === c * c,
      reduce: 0,
    do: (acc -> 1 + acc)
end
def pitag2cnt3(n), do:
    for a <- 1..n,
      b <- a..n,
      (ab = a+b) < n, # Javít-e?
      c <- b..n,
      ab + c <= n, # Javít-e?
      a * a + b * b === c * c,
      reduce: 0,
    do: (acc -> 1 + acc)
end
```

Pitagoraszi számhármasok száma 2

Megkérdeztem a *Copilotot*, hogyan csökkenthetném a jelöltek számát. A tartományok további szűkítési lehetőségét mutatta meg. (Egy zárójelhiba azért volt a kódban.) További két verzióval próbáltam javítani a futási időn.

```
def pitag3cnt1(n), do:
  for a <- 1..(div(n, 3)),
      b <- (a + 1)..(div(n - a, 2)),
      c = :math.sqrt(a * a + b * b),
      c == trunc(c),
      a + b + c <= n
    do {a, b, trunc(c)}
  end
  |> length()
end
def pitag3cnt2(n), do:
  for a <- 1..(div(n, 3)),
    b <- (a + 1)..(div(n - a, 2)),
    c = :math.sqrt(a * a + b * b),
    c == trunc(c),
    a + b + c <= n,
    reduce: 0,
  do: (acc -> 1 + acc)
end
```

```
def pitag3cnt3(n), do:
  for a <- 1..(div(n, 3)),
    b <- (a + 1)..(div(n - a, 2)),
    c = :math.sqrt(a * a + b * b),
    c == trunc(c),
    a + b + c <= n
  do :any_atom
  end
  |> length()
  end
```

Példák

```
iex> Fpea.pitag1cnt 1000
325
iex> Fpea.pitag2cnt3 1000
325
iex> Fpea.pitag3cnt1 1000
325
```

Pitagoraszi számhármasok száma 3: mérések *Benchee*-vel

```
Benchee.run(
%{"pitag1cnt:  naiv"                    => fn -> 200 |> Fpea.pitag1cnt() end,
  "pitag2cnt1: b>=a,c>=b"               => fn -> 200 |> Fpea.pitag2cnt1() end,
  "pitag2cnt2: 2cnt1 + a2,b2"           => fn -> 200 |> Fpea.pitag2cnt2() end,
  "pitag2cnt3: 2cnt1 + a+b<n"           => fn -> 200 |> Fpea.pitag2cnt3() end,
  "pitag3cnt1: szamolt c + {}-lista" => fn -> 200 |> Fpea.pitag3cnt1() end,
  "pitag3cnt2: 3cnt1 + reduce"          => fn -> 200 |> Fpea.pitag3cnt2() end,
  "pitag3cnt3: 3cnt1 + atomlista"       => fn -> 200 |> Fpea.pitag3cnt3() end
 }# , profile_after: true)
```

```
Name                                    IPS
pitag3cnt2: 3cnt1 + reduce              5743.38
pitag3cnt3: 3cnt1 + atomlista           5597.51 - 1.03x slower +0.00454 ms
pitag3cnt1: szamolt c + {}-lista        5546.79 - 1.04x slower +0.00617 ms
pitag2cnt3: 2cnt1 + a+b<n                 77.52 - 74.09x slower +12.73 ms
pitag2cnt1: b>=a,c>=b                     64.59 - 88.93x slower +15.31 ms
pitag2cnt2: 2cnt1 + a2,b2                 61.56 - 93.30x slower +16.07 ms
pitag1cnt:  naiv                          24.30 - 236.39x slower +40.98 ms
```

Gyorsrendezés (Quicksort)

```
@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"]
```

Permutáció

```
@spec perms(xs::[any()]) :: pss::[[any()]]
# Az xs lista elemeinek összes permutációját tartalmazó lista pss
def perms([]), do: [[]]
def perms(xs), do: (for y <- xs, ys <- perms(xs--[y]), do: [y|ys])
```

Példa:

```
iex> Fpea.perms [:a, :b]
[ [:a, :b], [:b, :a] ]
iex> Fpea.perms 'tér'
[
  [116, 233, 114],
  [116, 114, 233],
  [233, 116, 114],
  [233, 114, 116],
  [114, 116, 233],
  [114, 233, 116]
]
iex> 'tér'
[116, 233, 114]
```

Mivel az *é* nem 7 bites ASCII karakter, az IEx számlistaként írja ki az eredményt.

Tartalom

„Send more money” 1

- Valahol Angliában táviratot kézbesít a postás: `SEND + MORE = MONEY`.
- A címzett választáviratot küld a fiának: „Küldök, de mennyit?”. És a viszontválasz: „Megírtam! Minden betű más számjegy. Oldd meg!”
- A fiú nyilván nem szerénykedik, ötszámjegyű összeget kér, ezért az `M` nem lehet 0.
- Az egyszerű *generate-and-test* módszer nem elég jó, mert 10^8 nagyságrendű a variációk száma. Ezért már az első változatban figyelembe vesszük, hogy a betűk különböző számjegyeket jelentenek.
- Először a segédfüggvényt írjuk meg az összeg ellenőrzéséhez az

$$1000s + 100e + 10n + d + 1000m + 100o + 10r + e =$$

$$10000m + 1000o + 100n + 10e + y$$

egyenlet felhasználásával.

„Send more money” 2: `Sendmory.check_sum/1`

```
@type d() :: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
@type octet() :: d(),d(),d(),d(),d(),d(),d(),d()

@spec check_sum(c::octet()) :: b::boolean()
# b igaz, ha a c jelölt teljesíti az összeadási feltételt
defp check_sum(s,e,n,d,m,o,r,y) do
  send  = num([s,e,n,d])
  more  = num([m,o,r,e])
  money = num([m,o,n,e,y])
  send+more === money
end
```

A `num/1` függvény a számjegyek listáját egész számmá alakítja:

```
@spec num(ns::[d()]) :: n::integer()
# Az ns számjegylista decimális számként n.
def num(ns) do for n <- ns, reduce: 0, do: (acc -> 10*acc + n) end
```

„Send more money” 3: `Sendmory.smm1/0`

```
@spec smm1() :: [octet()]
# Ellenőrzések generálás közben is.
def smm1() do
  ds = 0..9
  for s <- ds,
    e <- ds, e !== s,
    n <- ds, not (n in [s,e]),
    d <- ds, not (d in [s,e,n]),
    m <- 1..9, not (m in [s,e,n,d]),
    o <- ds, not (o in [s,e,n,d,m]),
    r <- ds, not (r in [s,e,n,d,m,o]),
    y <- ds, not (y in [s,e,n,d,m,o,r]),
    check_sum(s,e,n,d,m,o,r,y),
    do: s,e,n,d,m,o,r,y
end
```

Name	ips	average	deviation	median	99th %
smm1	2.09	478.31 ms	±0.72%	477.12 ms	483.46 ms

Mindössze kétszer fut le egy másodperc alatt! Nagyon gyenge! Hol lehetne javítani rajta?

„Send more money” 4: `Sendmory.smm1/0`

#	CALLS	%	TIME	uS/CALL
Total	77326776	100.0	7263355	0.09
:lists.reverse/1	1	0.00	0	0.00
anonymous fn/0 in :elixir_compiler_2.__FILE__/1	1	0.00	0	0.00
anonymous fn/2 in Sendmory.smm1/0	10	0.00	1	0.10
Sendmory.smm1/0	1	0.00	1	1.00
:erlang.apply/2	1	0.00	3	3.00
anonymous fn/3 in Sendmory.smm1/0	100	0.00	10	0.10
anonymous fn/4 in Sendmory.smm1/0	900	0.00	140	0.16
anonymous fn/5 in Sendmory.smm1/0	7200	0.01	917	0.13
anonymous fn/6 in Sendmory.smm1/0	45360	0.08	5723	0.13
anonymous fn/7 in Sendmory.smm1/0	272160	0.50	36193	0.13
anonymous fn/8 in Sendmory.smm1/0	1360800	2.59	188308	0.14
Sendmory.num/1	4898880	4.30	312577	0.06
Enum.reduce/3	5612357	4.86	352752	0.06
Sendmory.check_sum/1	1632960	5.52	400715	0.25
Enum.reduce_range/5	3567385	8.03	583426	0.16
:lists.member/2	7129620	8.48	616031	0.09
anonymous fn/9 in Sendmory.smm1/0	5443200	11.17	811333	0.15
anonymous fn/2 in Sendmory.num/1	21228480	19.10	1387139	0.07
Enum."-reduce/3-listŝfoldl/2-0-"/3	26127360	35.36	2568086	0.10

Nem könnyű kihámozni, hogy a sok (19) függvény közül melyikhez nyúljunk, ráadásul sok köztük a névtelen. De a `Sendmory.num/1` gyanús.

„Send more money” 5: `Sendmory.check_sum2/1`, `smm1a/0`

A `Sendmory.num/1` gyanús, mert sokat hívja, közvetetten, a legtöbb időt felhasználó `Enum."-reduce/3-listsfoldl/2-0-"/3`-t.
A `Sendmory.check_sum/1` is az, mert ő hívja a `Sendmory.num/1`-ot.

```
defp check_sum(s,e,n,d,m,o,r,y) do
  send  = num([s,e,n,d])
  more  = num([m,o,r,e])
  money = num([m,o,n,e,y])
  send+more === money
end
```

Használjuk ki, hogy a számjegyek száma ismert (4 és 5)!

```
defp check_sum2(s,e,n,d,m,o,r,y) do
  send  = 1000*s + 100*e + 10*n + d
  more  = 1000*m + 100*o + 10*r + e
  money = 10000*m + 1000*o + 100*n + 10*e + y
  send+more === money
end
```

`Sendmory.smm1a/0`-ban csak annyi a változás, hogy `check_sum2/1`-t hívja.

```
Name             ips        average  deviation         median         99th %
smm1a           3.26      306.41 ms     ±1.00%      305.36 ms      313.92 ms
smm1            2.07      483.01 ms     ±0.85%      482.71 ms      488.42 ms
```

„Send more money” 6: `Sendmory.smm2/0`

Kb. 40%-os a javulás. Hol lehetne tovább javítani rajta? A profil egy részlete:

```
#                                          CALLS     %   TIME uS/CALL
Sendmory.check_sum2/1                    1632960  6.45 168839    0.10
anonymous fn/8 in Sendmory.smm1a/0       1360800  6.98 182793    0.13
Enum.reduce_range/5                      3567385 23.37 612292    0.17
:lists.member/2                          7129620 25.56 669607    0.09
anonymous fn/9 in Sendmory.smm1a/0       5443200 32.66 855524    0.16
```

A `check_sum2/1` sokszor és sokat fut, de amíg a varációk számát nem csökkentjük (most 1 632 960), addig ezen nem tudunk változtatni.

A `:lists.member/2`, azaz a `Kernel.in/2` operátor, a tagsági vizsgálat még többet fut, próbáljuk meg listák kivonásával, a `Kernel.--/2`-vel kiváltani.

```
def smm2() do
    ds = Range.to_list(0..9)
    for s <- ds,
      e <- ds -- [s],
      n <- ds -- [s,e],
      d <- ds -- [s,e,n],
      m <- ds -- [0,s,e,n,d],
      o <- ds -- [s,e,n,d,m],
      r <- ds -- [s,e,n,d,m,o],
      y <- ds -- [s,e,n,d,m,o,r],
      ...
```

„Send more money” 7: `Sendmory.smm2/0`, `smm3/0`

Name	ips	average	deviation	median	99th %
smm2	5.92	168.99 ms	±1.68%	169.47 ms	170.75 ms
smm1a	3.26	306.44 ms	±0.55%	305.61 ms	308.81 ms

Majdnem felére csökkent a futási idő, de még mindig nagyon sok. A variációk számát kellene csökkenteni, például úgy, hogy a listák építése közben eldobjuk a rossz variációkat.

Egy (tovább finomítható) lehetőség, hogy *hátulról* kezdjük a számjegyek összeadását, és megvizsgáljuk, hogy pl. `d+e` – ha kétjegyű, akkor a második számjegye – egyenlő-e `y`-nal (vö. sen**d** + mor**e** = mone**y**).

```
for d <- ds,
    e <- ds -- [d],
    y <- ds -- [d,e],
    rem(d+e, 10) === y,
    n <- ds -- [d,e,y],
    r <- ds -- [d,e,y,n],
    rem(num([n,d]) + num([r,e]), 100) === num([e,y]),
    o <- ds -- [d,e,y,n,r],
    rem(num([e,n,d]) + num([o,r,e]), 1000) === num([n,e,y]),
    s <- ds -- [d,e,y,n,r,o],
    m <- ds -- [0,d,e,y,n,r,o,s],
  check_sum2(s,e,n,d,m,o,r,y), ...
```

„Send more money” 8: `Sendmory.smm3/0`

```
Name           ips        average  deviation         median         99th %
smm3        1.42 K      703.67 us     ±3.43%      695.03 us      789.63 us
smm2      0.00613 K   163209.60 us     ±2.18%   163440.91 us   167366.79 us
```

Óriási a nyereség, kb. 230-szoros a gyorsulás! Nem olyan meglepő, ha azt is látjuk, hogy variációk száma 1866-ra csökkent az 1 632 960-ról:

```
#                                                 CALLS     %  TIME uS/CALL
Sendmory.check_sum2/1                              1866  1.56   233    0.12
...
Sendmory.num/1                                    13422  7.45  1115    0.08
Enum.reduce/3                                     15304  7.72  1155    0.08
anonymous fn/2 in Sendmory.num/1                  31194 18.33  2743    0.09
Enum."-reduce/3-listsf̂oldl/2-0-"/3                54894 47.97  7180    0.13
```

`smm3/0`-ban a `num/1`-et hívjuk, amiről már láttuk, hogy lassú. Hagyjuk el!

```
... n <- ds -- [d,e,y],
    r <- ds -- [d,e,y,n],
    rem(num([n,d]) + num([r,e]), 100) === num([e,y]),
    o <- ds -- [d,e,y,n,r],
    rem(num([e,n,d]) + num([o,r,e]), 1000) === num([n,e,y]), ...
```

Hat helyen hagytuk el `num/1`-et. Nézzük, hozott-e ez változást a futási időkben `smm3a/0`-ban.

„Send more money” 9: `Sendmory.smm3a/0`

```
Name             ips        average  deviation         median         99th %
smm3a        2.40 K      416.94 us     ±9.63%      405.19 us      601.81 us
smm3         1.55 K      645.86 us     ±6.75%      631.41 us      851.17 us
```

Most is kb. 40%-os a javulás. Hol lehetne még javítani a futási időkön?

```
#                                                 CALLS     % TIME uS/CALL
...
anonymous fn/4 in Sendmory.smm3a/0                   90  0.42   15    0.17
anonymous fn/5 in Sendmory.smm3a/0                  720  2.19   79    0.11
anonymous fn/6 in Sendmory.smm3a/0                  504  2.47   89    0.18
anonymous fn/9 in Sendmory.smm3a/0                  732  3.77  136    0.19
Enum.reduce/3                                      1882  4.22  152    0.08
anonymous fn/8 in Sendmory.smm3a/0                 1450  5.25  189    0.13
Sendmory.check_sum2/1                              1866  6.91  249    0.13
anonymous fn/9 in Sendmory.smm3a/0                 1866  9.77  352    0.19
anonymous fn/7 in Sendmory.smm3a/0                 3024 10.07  363    0.12
:erlang.--/2                                       1881 13.10  472    0.25
Enum."-reduce/3-listsf̂oldl/2-0-"/3                10278 41.66 1501    0.15
```

További varációk kizárásával, például úgy, hogy a két szám összeadásakor esetleg keletkező átvitelt is felhasználjuk.
Esetleg a for-comprehenzió kiváltásával (most már csak a komprehenzió hívja az `Enum."-reduce/3-listsf̂oldl/2-0-"/3`-t).

XV. rész

Lineáris és elágazó rekurzió

Tartalom

Rekurzív adatstruktúrák

A rekurzív adatstruktúrák (adattípusok) alapvetően kétfélék.

- Lineáris rekurzív adatstruktúra (lista)
  A funkcionális nyelvekben: beépített típus
  Megvalósítás: láncolt listával
  - Elixirben a beépített lista típusa: *@type list() :: [] | [any() | list()]*
  - *Ennessel is megvalósíthatjuk,* nevezzük veremnek:
    @type lifo() :: :empty | {lifo(), any()} # last-in-first-out
- Elágazó rekurzív adatstruktúra (fa)
  Általában nem beépített típus a funkcionális nyelvekben, de többnyire vannak fák használatát segítő modulok (pl. Erlangban a gb_tree)
  - Bináris fa: *@type btree() :: :lf | {btree(), any(), btree()}*
  - Három ágú fa: *@type ttree() :: :lf | {ttree(), ttree(), ttree(), any()}*
  - Sok ágú fa (pl. egy listában vannak a csomóponthoz tartozó fák):
    @type ltree() :: :lf | {any(), [ltree()]}

  Az ennesben az érték és az ágak sorrendje tetszőleges. Az érték lehetne csak a levélben vagy levélben is.

Verem megvalósítása ennessel 1 (`dp_lifo.ex`)

- Típusa: *@type lifo() :: :empty | {lifo(), any()} # last-in-first-out*
- Definíciók:
  - verem teteje = a verem tetején lévő elem
  - verem alja = a verem teteje alatti veremrész
- Műveletek:
  - üres verem létrehozása (`empty/0`)
  - a verem üres voltának vizsgálata (`is_empty/1`)

```
@spec empty() :: s::lifo()
# s az üres verem
def empty(), do: :empty

@spec is_empty(s::lifo()) :: b:: boolean()
# b igaz, ha s üres
def is_empty(:empty),   do: true
def is_empty({_s,_x}),  do: false
```

Verem megvalósítása ennessel 2 (`dp_lifo.ex`)

- Típusa: *@type lifo() :: :empty | {lifo(), any()} # last-in-first-out*
- Műveletek:
  - egy elem berakása a verembe (`push/2`)
  - a verem alja (`pop/1`)
  - a verem teteje (`top/1`)

```
@spec push(s::lifo(), x::any()) :: s_new::lifo()
# s_new az x-szel megfejelt s verem
def push(:empty, x),       do: {:empty, x}
def push({_s,_x} = s, x), do: {s, x} # {_s,_x} = s: réteges minta

@spec pop(s::lifo()) :: s_new::(lifo() | nil)
# s_new az s verem alja, vagy nil, ha s üres
def pop(:empty),  do: nil
def pop({s, _x}), do: s

@spec top(s::lifo()) :: x::(any() | nil)
# x az s verem teteje, vagy nil, ha s üres
def top(:empty),  do: nil
def top({_s, x}), do: x
```

Kis példák a verem használatára 1 (dp_lifo.ex)

Elemek berakása egy verembe

```
iex> s1 = Dp.Lifo.push(Dp.Lifo.empty(), 1)
{:empty, 1}
iex> s2 = Dp.Lifo.push(s1, 2)
{{:empty, 1}, 2}
iex> s3 = Dp.Lifo.push(s2, 3)
{{{:empty, 1}, 2}, 3}
```

Ugyanez tömörebben és érthetőbben:

```
iex> push = &Dp.Lifo.push/2; Dp.Lifo.empty() |> push.(1) |> push.(2) |> push.(3)
{{{:empty, 1}, 2}, 3}
```

Kis példák a verem használatára 2 (`dp_lifo.ex`)

Fordítsunk meg egy karakterlistát!

Emlékeztetőül a `push` specifikációja:
@spec push(s::lifo(), x::any()) :: s_new::lifo().

1. lépés: a verembe betesszük az elemeket

```
iex> ~c"szöveg"
[115, 122, 246, 118, 101, 103]
iex> lifo = ~c"szöveg" \
  |> List.foldl(Dp.Lifo.empty(), &(Dp.Lifo.push(&2,&1)))
{{{{{{:empty, 115}, 122}, 246}, 118}, 101}, 103}
```

2. lépés: a verem elemeit sorban kivesszük és listába fűzzük

```
@spec to_list(s::lifo()) :: ls::[any()]
# ls az s verem elemeit tartalmazó lista LIFO sorrendben
def to_list(:empty), do: []
def to_list({s, x}), do: [x | to_list(s)]
```

```
iex> Dp.Lifo.to_list(lifo)
[103, 101, 118, 246, 122, 115]
iex> Dp.Lifo.to_list(lifo) |> List.to_string
"gevözs"
```

Rekurzív adatstruktúrák 2

A rekurzív adatstruktúrák (adattípusok) alapvetően kétfélék

- Lineáris rekurzív adatstruktúra (lista)
  A funkcionális nyelvekben: beépített típus
  Megvalósítás: láncolt listával
  - Elixirben a beépített lista típusa: *@type list() :: [] | [any() | list()]*
  - Ennessel is megvalósíthatjuk, nevezzük veremnek (last-in-first-out): *@type lifo() :: :empty | {lifo(), any()} # last-in-first-out*
- Elágazó rekurzív adatstruktúra (fa)
  Általában nem beépített típus a funkcionális nyelvekben, de többnyire vannak fák használatát segítő modulok (pl. Erlangban a gb_tree)
  - Bináris fa: *@type btree() :: :lf | {btree(), any(), btree()}*
  - Három ágú fa: *@type ttree() :: :lf | {ttree(), ttree(), ttree(), any()}*
  - Sok ágú fa (pl. egy listában vannak a csomóponthoz tartozó fák): *@type ltree() :: :lf | {any(), [ltree()]}*

  Az ennesekben az érték és az ágak sorrendje tetszőleges. Az érték lehetne csak a levélben vagy lehetne levélben is.

Bináris fából lista preorder bejárással (`dp_tree.ex`)

- Típusdefiníció (egy a sok lehetséges közül)
  @type btree() :: :lf | {btree(), any(), btree()}
- Jelölés (példák)

```
bt1  = {:lf,:a,:lf}
bt2a = { {:lf,:c,:lf}, :b, :lf }
bt2b = {:lf, :e, {:lf,:f,:lf} }
bt2c = { {:lf,:h,:lf}, :g, {:lf,:i,:lf} }
bt2  = {bt2a, :a, {bt2b, :d, bt2c } }
```

- Bejárás (preorder)
  @spec bt_to_list(btr :: btree()) :: xs :: [any()]
  # A btr fa preorder bejárásának eredménye az xs lista

```
def bt_to_list(:lf),          do: []
def bt_to_list({bt, v, jt}), do:
    [v | bt_to_list(bt)] ++ bt_to_list(jt)
```

- Futtatás

```
iex> Dp.Tree.bt_to_list(bt1)
[:a]
iex> Dp.Tree.bt_to_list(bt2)
[:a, :b, :c, :d, :e, :f, :g, :h, :i]
```

Három ágú fából lista preorder bejárással (`dp_tree.ex`)

- Típusdefiníció (egy a sok lehetséges közül)
  @type ttree() :: :lf | {ttree(), ttree(), ttree(), any()}
- Jelölés (példák)

```
tt1  = {:lf,:lf,:lf,:a}
tt2a = { {:lf,:lf,:lf,:c}, :lf, :lf, :b }
tt2b = { {:lf,:lf,:lf,:d}, {:lf,:lf,:lf,:f}, :lf, :e }
tt2c = { {:lf,:lf,:lf,:h}, :lf, {:lf,:lf,:lf,:i}, :g }
tt2  = { tt2a, tt2b, tt2c, :a }
```

- Bejárás (preorder)
  @spec tt_to_list(ttr :: ttree()) :: xs :: [any()]
  # A ttr fa preorder bejárásának eredménye az xs lista

```
def tt_to_list(:lf),              do: []
def tt_to_list(bt, mt, jt, v), do:
    [v | tt_to_list(bt)] ++ tt_to_list(mt) ++ tt_to_list(jt)
```

- Futtatás

```
iex> Dp.Tree.tt_to_list(tt1)
[:a]
iex> Dp.Tree.tt_to_list(tt2)
[:a,:b,:c,:d,:e,:f,:g,:h,:i]
```

Sok ágú fából lista preorder bejárással (`dp_tree.ex`)

- Típusdefiníció (egy a sok lehetséges közül)
  @type ltree() :: :lf | {any(), [ltree()]}
- Jelölés (példák)

```
lt1  = {:lt, []}
lt2a = {:b, [{:c, []}, {:d, []}]}
lt2b = {:e, [{:f, []}]}
lt2c = {:g, [{:h, []}, {:i, []}]}
lt2  = {:a, [lt2a, lt2b, lt2c]}
```

- Bejárás (preorder)
  @spec lt_to_list(ltr :: ltree()) :: xs :: [any()]
  # Az ltr fa preorder bejárásának eredménye az xs lista

```
def lt_to_list(:lf),   do: []
def lt_to_list(v, ts), do:
  [ v | Enum.concat(Enum.map(ts, &lt_to_list/1)) ]
```

- Futtatás

```
iex> {(Dp.Tree.lt_to_list :lf), Dp.Tree.lt_to_list lt1}
{[], [:a]}
iex> Dp.Tree.lt_to_list(lt2)
[:a, :b, :c, :d, :e, :f, :g, :h, :i]
```

Egyszerű műveletek bináris fákon 1 (dp_tree.ex)

- Legyen most ez a bináris fánk típusa:

  @type btree2() :: :lf | {any(), btree2(), btree2()}
- Fa létrehozása

  @spec empty() :: btr::(:lf)
  # btr az egyetlen levélből álló üres fa

```
def empty(), do: :lf
```

  @spec node(v: any(), lt::btree2(), rt::btree2()) :: btr::btree2()
  # btr a v értékből, az lt és az rt fákból összerakott fa

```
node(v, lt, rt), do: {v, lt, rt}
```

- Fa leveleinek és szintjeinek száma

  @spec leaves_cnt(btr::btree2()) :: n::integer()
  # n a btr fa leveleinek száma

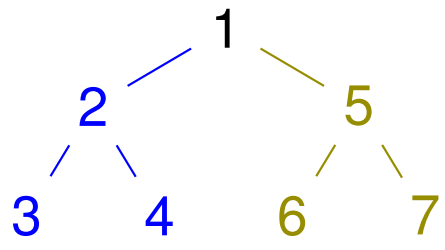
```
def leaves_cnt(:lf),       do: 1
def leaves_cnt({_,lt,rt}), do: leaves_cnt(lt)+leaves_cnt(rt)
```

  @spec depth(btr::btree2()) :: d::integer()
  # d a btr fa mélysége, azaz szintjenek száma

```
def depth(:lf),       do: 0
def depth({_,lt,rt}), do: 1 + max(depth(lt), depth(rt))
```

Egyszerű műveletek bináris fákon 2 (dp_tree.ex)

```
e=empty(), t=node(1, node(2, node(3,e,e),
                             node(4,e,e)),
                     node(5, node(6,e,e),
                             node(7,e,e)))
```

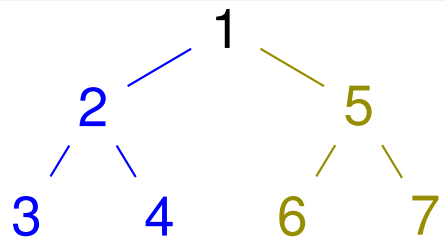



```
iex> e = &Dp.Tree.empty/0; n = &Dp.Tree.node/3; \
     t = n.(1, n.(2, n.(3, e.(), e.()), n.(4, e.(), e.())), \
               n.(5, n.(6, e.(), e.()), n.(7, e.(), e.())))
{1, {2, {3,:lf,:lf}, {4,:lf,:lf}}, {5, {6,:lf,:lf}, {7,:lf,:lf}}}

iex> Dp.Tree.leaves_cnt t
8
iex> Dp.Tree.depth t
3
```

Műveletek bináris fákon: fából lista (dp_tree.ex)

```
e=empty(), t=node(1, node(2, node(3,e,e),
                             node(4,e,e)),
                     node(5, node(6,e,e),
                             node(7,e,e)))
```




```
@spec to_list_preord(btr::btree2()) :: xs::[any()]
# A btr fa preorder bejárásának eredménye az xs lista
def to_list_preord(:lf),       do: []
def to_list_preord({v,lt,rt}), do:
      [v] ++ to_list_preord(lt) ++ to_list_preord(rt)
```

```
@spec to_list_inord(btr::btree2()) :: xs::[any()]
# A btr fa inorder bejárásának eredménye az xs lista
def to_list_inord(:lf),       do: []
def to_list_inord({v,lt,rt}), do:
      to_list_inord(lt) ++ ([v] ++ to_list_inord(rt))
```

```
iex> Dp.Tree.to_list_preord(t)
[1, 2, 3, 4, 5, 6, 7]
iex> Dp.Tree.to_list_inord(t)
[3, 2, 4, 1, 6, 5, 7]
```

Műveletek bináris fákon: listából fa (dp_tree.ex)

```
t = {1, {2, {3,:lf,:lf}, {4,:lf,:lf}}, {5, {6,:lf,:lf}, {7,:lf,:lf}}}
```

```
@spec from_list_preord(xs::[any()]) :: btr::btree2()
# btr az xs listából preorder sorrendben felépített fa
def from_list_preord([]),    do: empty()
def from_list_preord([x|xs]) do
  {l1s, l2s} = Enum.split(xs, div(length(xs), 2)))
  node(x, from_list_preord(l1s), from_list_preord(l2s))
end
```

```
@spec from_list_inord(xs::[any()]) :: btr::btree2()
# btr az xs listából inorder sorrendben felépített fa
def from_list_inord([]),  do: empty()
def from_list_inord(xs)   do
  {l1s, [x|l2s]} = Enum.split(xs, div(length(xs), 2))
  node(x, from_list_inord(l1s), from_list_inord(l2s))
end
```

```
iex> Dp.Tree.from_list_preord([1, 2, 3, 4, 5, 6, 7])
{1,{2,{3,:lf,:lf},{4,:lf,:lf}},{5,{6,:lf,:lf},{7,:lf,:lf}}}
iex> Dp.Tree.from_list_inord([3, 2, 4, 1, 6, 5, 7])
{1,{2,{3,:lf,:lf},{4,:lf,:lf}},{5,{6,:lf,:lf},{7,:lf,:lf}}}
```

XVI. 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
```

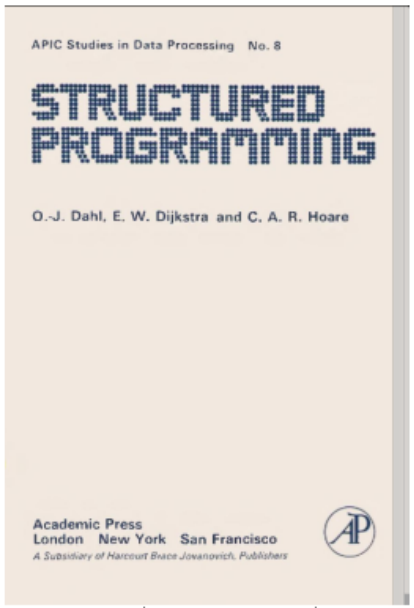
NASA, 1960-as évek eleje, Lásd:
`http://spiff.rit.edu/classes/phys317/lectures/boom/mercury.txt`

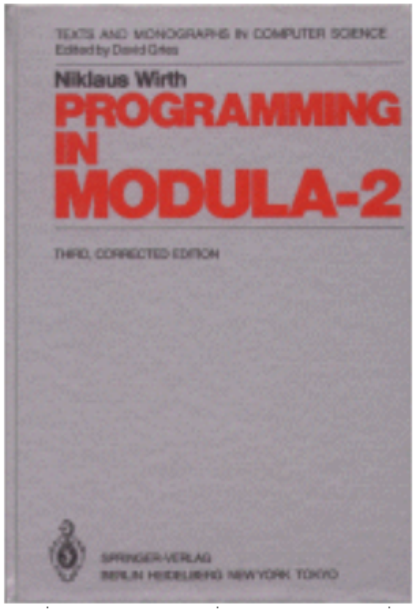
Programozás az 1970-es évek második felétől

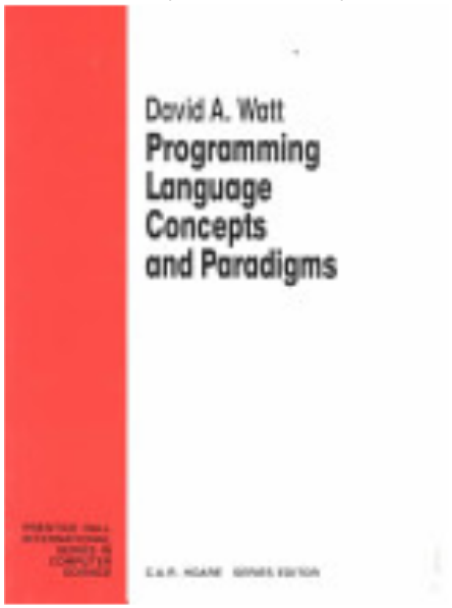
- Simula (1960+), ALGOL 68 (1968)
- C (1970+), később (1985) C++ programozási nyelv
- Absztrakt adattípusok (1974)
- CDL (Compiler Description Language, C.H.A. Koster, 1971)
- Structured Programming (O.-J. Dahl, E.W. Dijkstra, T. Hoare, 1972)
- Modular Programming, Modula (N. Wirth, 1972)
- ELAN (Educational Language, C.H.A. Koster, 1974)
- Objektum-orientált programozás (elterjedése 1970-es évek végétől)
- Ada (1983)
- Eiffel (B. Meyer, 1986), Sather (UC Berkeley, 1990)
- Specifikációs nyelvek (pl. „Z”, 1980)
- SML (1983), OCAML (1996) funkcionális programozási nyelvek

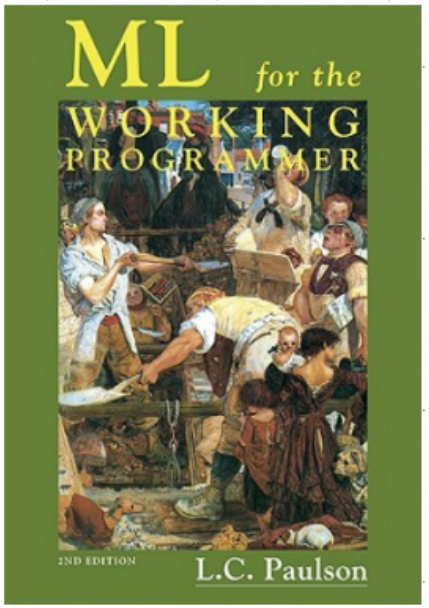
$\Rightarrow$Törekvés megbízható, biztonságos, karbantartható szoftverre

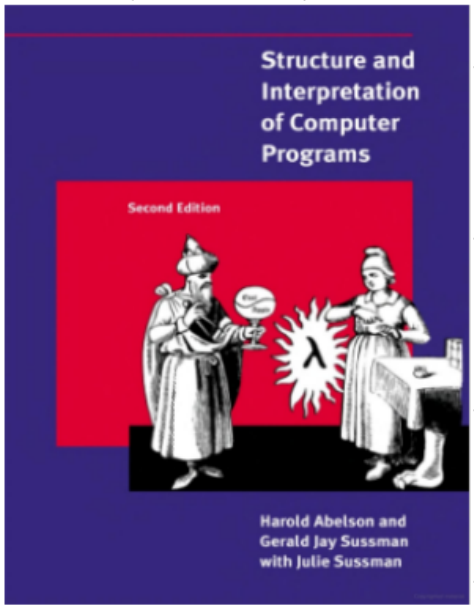
Néhány (nekem) fontos könyv a programozásról











Funkcionális programozás, nyelvek

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

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

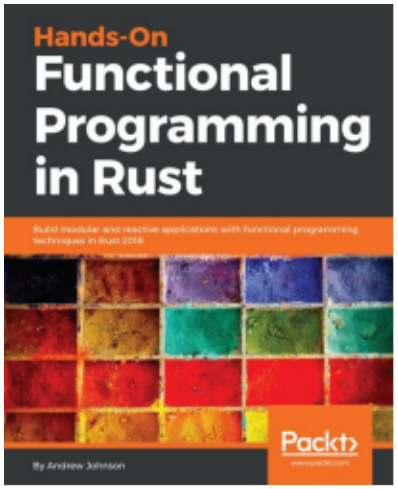
Mi jutott a Google „eszébe” 2021-ben, amikor a *books* és *functional programming* szavakra kerestem rá? [42]

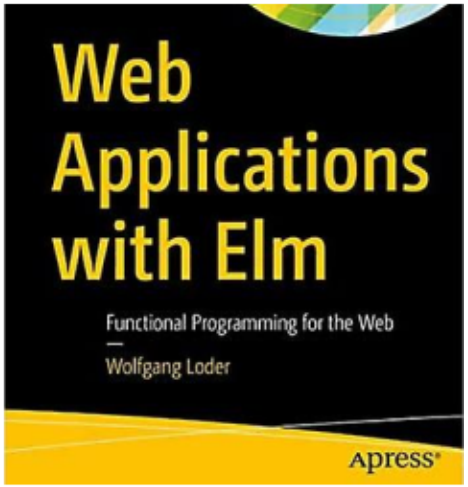
[42] A találati sorrend nagyjából a következő diákon látható sorrend volt 2021-ben, a többszörös vagy hasonló találatok közül csak egyet-kettőt hagytam meg.

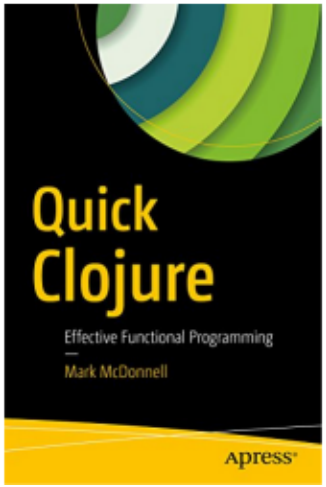
Könyvek a funkcionális programozásról 1

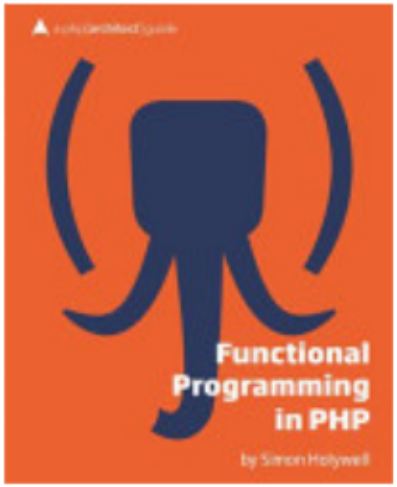
JavaScript, Scala, **Haskell**, TypeScript, Python, Kotlin, **F#**, C++, C#

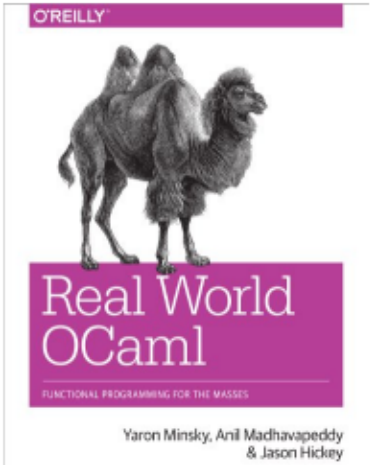
Könyvek a funkcionális programozásról 2

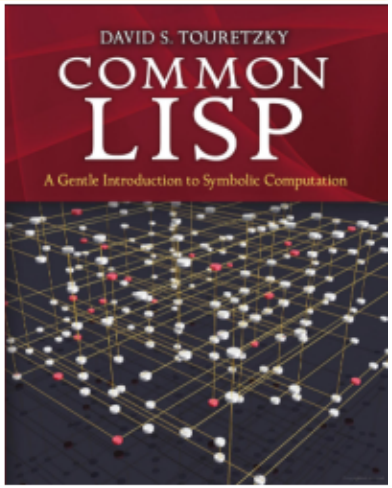






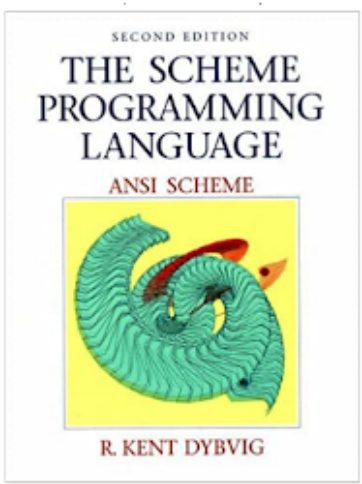


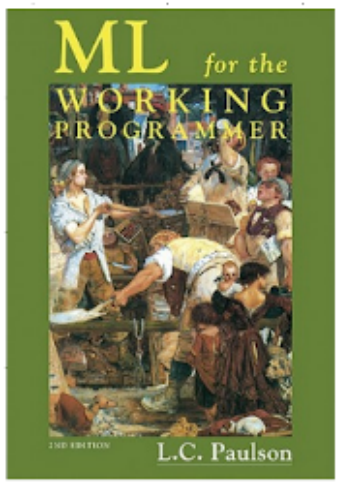


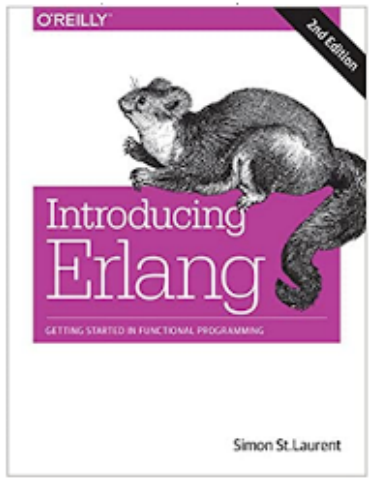


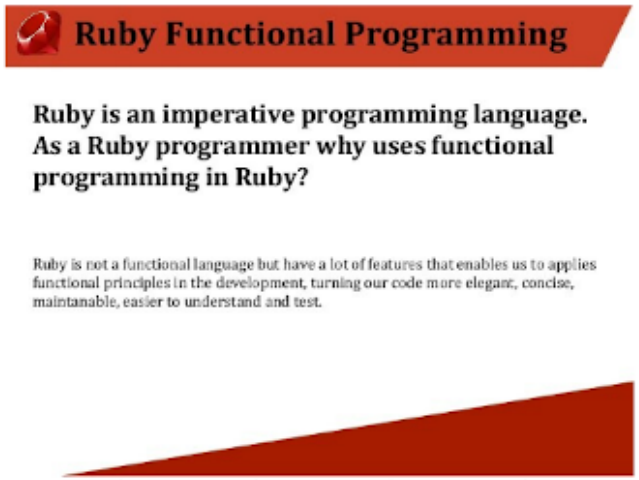
Rust, Swift, **Elm**, **Clojure**, Java, PHP, **OCaml**, **Lisp**

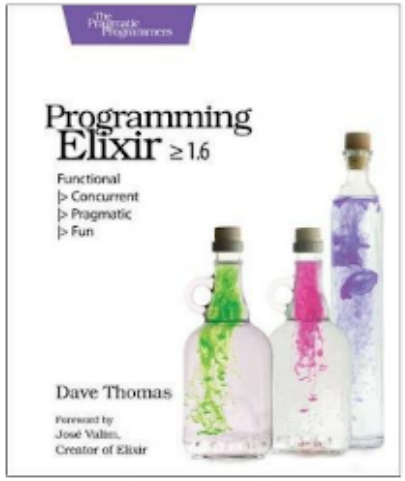
Könyvek a funkcionális programozásról 3

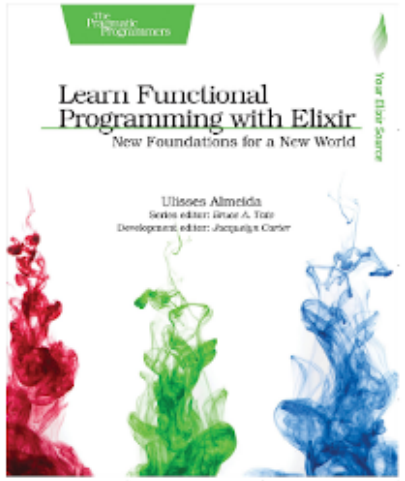












Scheme, **SML**, **Erlang**, Ruby, **Elixir**

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 [$\mathbb{D}$], Elixir [$\mathbb{D}$], **Gleam** [$\mathbb{S}$]: Erlang VM-en [BEAM] futnak
- 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; például a BEAM-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`

XVII. rész

Programozás Prologban

A kurzus Logikai Programozás (LP) része

- A Prolog LP nyelv alapjai
  - Szintaxis
  - Deklaratív szemantika
  - Procedurális szemantika (végrehajtási mechanizmus)
  - A legfontosabb beépített eljárások
  - Prolog programozási módszerek
- Haladó deklaratív programozás Prologban
  - Fejlettebb nyelvi és rendszerelemek
  - Kitekintés: új irányzatok a logikai programozásban

A logikai programozás alapgondolata

- Logikai programozás (LP):
  - Programozás a matematikai logika segítségével
    - egy logikai program nem más mint logikai állítások halmaza
    - egy logikai program futása egy következtetési folyamat
  - De: a logikai következtetés óriási keresési tér bejárását jelenti...
    - Szorítsuk meg a logika nyelvét!
    - Válasszunk egyszerű, ember által is követhető következtetési algoritmust!
  - Az LP máig legelterjedtebb megvalósítása a már több mint 50 éves Prolog = Programozás logikában (Programming in logic)
    - az elsőrendű logika (First Order Logic, FOL) egy erősen megszorított részhalmaza az ún. definit- vagy más néven Horn-klózok
    - végrehajtása (következtetés): mintaillesztéses eljáráshíváson alapuló visszalépéses keresés.
  - A Prolog nyelv mottója: „WHAT rather than HOW”

Tartalom

A logikai és a funkcionális programozás összehasonlítása

Mi a közös a funkcionális (FP) és logikai (LP) nyelvekben?

1. A matematikai változó fogalma: egyetlen még ismeretlen értéket jelöl, nem mutábilis (nem változtatható)
2. Ciklusok helyett: rekurzió, vagy más eszközök (pl. listanézet)

Miben más az LP (ill. a Prolog) mint az FP megközelítés?

3. Az FP a lambda-kalkuluson alapul, az LP alapja az elsőrendű logika (FOL)
   - Az LP-ben függvények helyett predikátumokat (relációkat, vö. adatbáziskezelés) kell definiálnunk, predikátum $\equiv$ eljárás
   - Egy LP program futása 0, 1, vagy több eredményt adhat – nemdeterminisztikus keresés, vö. SQL lekérdezések
4. A Prolog szintaxisa kiterjeszthető – saját új operátorokat definiálhatunk
5. LP-ben a változók egy adatstruktúra (pl. lista) belsejében is előfordulhatnak – ezek a logikai változók pointerként működnek, pl. `[X,X,X]` egy olyan 3-elemű listát jelöl, amely csupa azonos (de egyelőre még ismeretlen) elemből áll
6. Az LP különösen jó szimbolikus feladatokra (pl. szimbolikus deriválás)
7. Sok FP nyelv van, de praktikusan a Prolog az egyetlen LP nyelv

Ismétlés: listák összefűzése

- A Prolog lista szintaxisa megegyezik az Elixir szintaxissal, de Prologban a változók kötelezően nagybetűvel vagy aláhúzásjellel (_) kezdődnek
- Idézzük fel a két listát összefűző `app` 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]
```

- Ennek egy 3-argumentumú Prolog predikátum felel meg, (`app/3`), itt a 3. argumentum lesz az összefűzött lista (az Elixir függvény 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
```

- A `C` segédváltozó az `A⊕B` részeredményt tárolja.
- Példa az `app/3` predikátum használatára:

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

- A fenti `app/3` eljárás `append` néven beépített predikátumként is elérhető.

Be- és kimenő argumentumok

- Az `app/3` predikátum az `app/2` Elixir függvény átírásával állt elő
- A Prolog predikátum azonban használható más módon is, pl:

```
| ?- app(L1, L2, [1,2]).
L1 = [], L2 = [1,2] ? ;
L1 = [1],  L2 = [2] ? ;
L1 = [1,2], L2 = [] ? ; no
```

- Az ún. I/O mód jelölésrendszer a különböző módú hívások leírására:
  - `+`: bemenő (input) arg., a hívás pillanatában behelyettesített
  - `-`: kimenő (output) arg., a hívás pillanatában behelyettesítetlen
  - `?`: be- és kimenő arg., tetszőleges Prolog kifejezés lehet
- Példák az `app(L1, L2, L3)` hívás különböző módú hívásaira:

  `(+,+,+)`: konkatenálás ellenőrzése, pl. `app([1], [2], [1,2])` ⇒ `yes`

  `(+,+,-)`: konkatenálás, pl. `app([1], [2], L3)` ⇒ `L3 = [1,2]`

  `(+,-,+)`: két lista „különbsége”, pl. `app([1], L2, [1,2])` ⇒ `L2 = [2]`

  `(+,-,-)`: nyílt végű lista előállítása, pl. `app([1], L2, L3)` ⇒ `L3 = [1|L2]`

  `(-,-,+)`: lista szétszedése, pl. `app(L1, L2, [1,2])` ⇒ lásd fent

  `(-,?,-)`: ∞ keresési tér: pl. `app(L1, [1], L3)` ⇒ `L1 = [], L3 = [1]? ;`

  `L1 = [A], L3 = [A,1]? ; L1 = [A,B], L3 = [A,B,1]? ;...`
- Az eredményekben logikai változók is lehetnek, lásd fenn, pl. `L2`, `A`, `B` stb.

Tartalom

A Prolog alapelemei: a családi kapcsolatok példája

- Adottak személyekre vonatkozó, adatbázis-szerű állítások, pl.

„gyerek–szülő” tábla

gyerek	szülő
Imre	István
Imre	Gizella
István	Géza
István	Sarolt
Gizella	C. Henrik
Gizella	B. Gizella

„férfiak” tábla

férfi
Imre
István
Géza
C. Henrik

- Rövidítések feloldása: C. Henrik $\Rightarrow$ Civakodó Henrik,
  B. Gizella $\Rightarrow$ Burgundi Gizella
- Definiáljuk az unoka–nagyszülő kapcsolatot, azaz hozzunk létre egy származtatott (virtuális) „unoka–nagyszülő” táblát!

A nagyszülő feladat — Prolog megoldás

- Egy Prolog program állításokból, ún. klózokból (clause) áll
- A legegyszerűbb klóz a tényállítás (fact), formája:

  `relációnév(`$\mathtt{Arg}_1$`, ..., `$\mathtt{Arg}_n$`).` (ez egy `klózfej`)
- A `relációnév` egy névkonstans (atom): kisbetűvel kezdődő azonosító vagy aposztrófok közé zárt tetsz. karaktersorozat (első közelítésben)
- Az argumentumok lehetnek névkonstansok, változók, stb.
- A változókat nagybetűvel kezdődő azonosítókkal – pl. `Gy`, `Sz` – jelöljük
- Az Imre herceg őseit leíró adatbázis-táblák Prolog alakja:

```
% sz(Gy, Sz): Gy szülője Sz.
  sz('Imre',    'Gizella').    % (sz1)
  sz('Imre',    'István').     % (sz2)
  sz('István',  'Sarolt').     % (sz3)
  sz('István',  'Géza').       % (sz4)
  sz('Gizella', 'B. Gizella'). % (sz5)
  sz('Gizella', 'C. Henrik').  % (sz6)
```

```
% ffi(Személy): Személy férfi.
  ffi('Imre').       % (f1)
  ffi('István').     % (f2)
  ffi('Géza').       % (f3)
  ffi('C. Henrik'). % (f4)
```

- A predikátumok jelentését egy `% fejkomment`-tel írjuk le, `/* ez is komment */`
- Azonos nevű és argumentumszámú klózok sorozata egy predikátumot alkot, pl. a fenti klózok az `sz/2` ill. `ffi/1` predikátumokat

A nagyszülő feladat — Prolog megoldás (folyt.)

- A klózok másik fajtája az ún. szabály (rule), formája:

```
klózfej :- cél_1, ..., cél_k.        % klózfej ← cél_1 ∧ ... ∧ cél_k
        % ^--klóztörzs--^
```

- A `cél` (goal), más néven hívás (call) szintaxisa (azonos a `klózfej`-évél):

```
relációnév(Arg_1, ..., Arg_n)
```

- A „nagyszülője” kapcsolatot definiáló szabály:

```
% Gyerek nagyszülője Nagyszulo.
nsz(Gyerek, Nagyszulo) :-        % Gyerek nagyszülője Nagyszulo ha ∃ Szulo
      sz(Gyerek, Szulo),               % Gyerek szülője Szulo és
      sz(Szulo, Nagyszulo).            % Szulo szülője Nagyszulo        (nsz)
```

- Egy program futtatásához egy célsorozatot (lekérdezést) kell megadni:

```
%    Ki Imre nagyapja? (pontosabban  Ki Imre férfi nagyszülője?)
| ?- nsz('Imre', NA), ffi(NA).       NA = 'C. Henrik' ? ;
                                     NA = 'Géza' ? ; no
%    Ki Géza unokája?
| ?- nsz(U, 'Géza').                 U = 'Imre' ? ; no
%    Ki Imre nagyszülője?
| ?- nsz('Imre', NSz).               NSz = 'B. Gizella' ? ;
                                     NSz = 'C. Henrik' ? ;
                                     NSz = 'Sarolt' ? ; NSz = 'Géza' ? ; no
```

Deklaratív szemantika – klózok logikai alakja

- A szabály jelentése egy implikáció: a törzsbeli célok konjunkciójából következik a fej.
  - Példa: `nsz(Gy,NSz) :- sz(Gy,Sz), sz(Sz,NSz).`
  - Logikai alak: $\forall \texttt{Gy}, \texttt{NSz}, \texttt{Sz}(\texttt{nsz}(\texttt{Gy}, \texttt{NSz}) \leftarrow \texttt{sz}(\texttt{Gy}, \texttt{Sz}) \wedge \texttt{sz}(\texttt{Sz}, \texttt{NSz}))$
  - Ekvivalens alak: $\forall \texttt{Gy}, \texttt{NSz}(\texttt{nsz}(\texttt{Gy}, \texttt{NSz}) \leftarrow \exists \texttt{Sz}(\texttt{sz}(\texttt{Gy}, \texttt{Sz}) \wedge \texttt{sz}(\texttt{Sz}, \texttt{NSz})))$
- A tényállítás feltétel nélküli állítás, pl.
  - Példa: `sz('Imre', 'István').`
  - Logikai alakja változatlan
  - Ebben is lehetnek változók, ezeket is univerzálisan kell kvantálni
- A célsorozat jelentése: keressük azokat a változó-behelyettesítéseket amelyek esetén a célok konjunkciója igaz
- Egy célsorozatra kapott válasz helyes, ha az adott behelyettesítésekkel a célsorozat következménye a program logikai alakjának – WHAT
- A Prolog garantálja a helyességet, de a teljességet nem: nem biztos, hogy minden megoldást megkapunk (kaphatunk hibajelzést, végtelen ciklust, végtelen keresési teret stb.) – HOW

Procedurális szemantika: az ún. redukciós modell

Redukciós lépés: egy CS_i célsorozat visszavezetése CS_{i+1}-re úgy, hogy

$$CS_i \leftarrow Program \wedge CS_{i+1}$$

Pl. az `(1)` célsorozat redukciója az `(nsz)` programklózzal `(2)`-t eredményezi:

```
:- nsz('Imre', NA),              ffi(NA).    (kiinduló célsorozat)   (1)
:- sz('Imre', Sz1), sz(Sz1, NA), ffi(NA).    (redukált célsorozat)   (2)
```

1. A klózt **lemásoljuk**, a változókat szisztematikusan újakra cserélve
   `nsz(Gy1, NSz1) :- sz(Gy1, Sz1), sz(Sz1, NSz1).` (nsz′)
2. `(1)`-et szétbontjuk, első cél: `nsz('Imre', NA)`, maradék célsor.: `ffi(NA)`.
3. Az első célt **egyesítjük** a klózfejjel, azaz változók behelyettesítésével a klózfejjel azonos alakra hozzuk (**kétirányú** mintaillesztés): behelyettesítés: `Gy1 = 'Imre', NSz1 = NA`, közös alak: `nsz('Imre', NA)`
4. Ha az egyesítés sikertelen, akkor a redukciós lépés meghiúsul, egyébként behelyettesítjük a klóztörzset: `sz('Imre', Sz1), sz(Sz1, NA)` és a maradék célsorozatot is (ebben most nincs változás): `ffi(NA)`
5. Új célsorozat = klóztörzs és utána a maradék célsorozat (lásd fenn, `(2)`)

Az `(1)`→`(2)` redukciós lépés értelmezhető az `(nsz)` „makró” kifejtéseként...

További redukciós lépések

A (2) célsorozat redukciója az (sz1) programklózzal:

```
:- sz('Imre', Sz1), sz(Sz1, NA), ffi(NA).                    (2)
```

1. Az (sz1) klóz nem tartalmaz változót, így nem szükséges lemásolni:

   ```
   sz('Imre', 'Gizella') /* :- (üres klóztörzs) */.          (sz1)
   ```

2. (2) első célja: `sz('Imre', Sz1)`, maradék célsor.: `sz(Sz1, NA), ffi(NA).`
3. Az első célt **egyesítjük** a klózfejjel
   behelyettesítés: `Sz1 = 'Gizella'`, közös alak: `sz('Imre', 'Gizella')`
4. A behelyettesített maradék: `sz('Gizella', NA), ffi(NA).`
5. Az új célsorozat: az (sz1) klóz (üres) törzse + a maradék célsorozat:

```
:- sz('Gizella', NA), ffi(NA).                               (3)
```

(Tényállítással redukálva 1-gyel csökken a célsorozat hossza!)

(3)-at redukálva (sz6)-tal (`sz('Gizella', 'C. Henrik').`) a `NA = 'C. Henrik'` behelyettesítést kapjuk, az új célsorozat:

```
:- ffi('C. Henrik').                                         (4)
```

(4)-et redukálva (f4)-gyel (`ffi('C. Henrik').`) üres célsorozatot (□) kapunk.
Ezzel megállapítottuk, hogy az `NA = 'C. Henrik'` egy megoldás (1)-re.

A nagyszülő példa végrehajtása – egy teljes levezetés

```
% sz(Gy, Sz): Gy szülője Sz.
  sz('Imre',     'Gizella').    % (sz1)
  sz('Imre',     'István').     % (sz2)
  sz('István',   'Sarolt').     % (sz3)
  sz('István',   'Géza').       % (sz4)
  sz('Gizella',  'B. Gizella'). % (sz5)
  sz('Gizella',  'C. Henrik').  % (sz6)
```

```
% ffi(Személy): Személy férfi.
  ffi('Imre').        % (f1)
  ffi('István').      % (f2)
  ffi('Géza').        % (f3)
  ffi('C. Henrik').   % (f4)
```

```
% Gy nagyszülője NSz.
nsz(Gy, NSz) :-        sz(Gy, Sz), sz(Sz, NSz).          % (nsz)
```

```
(1)  :- nsz('Imre', NA), ffi(NA).                    (nsz): (1) ← (2)

(2)  :- sz('Imre', Sz1), sz(Sz1, NA), ffi(NA).       (sz1): (2) ← (3)
                                          Sz1 = 'Gizella'
(3)  :- sz('Gizella', NA), ffi(NA).                  (sz6): (3) ← (4)
                                          NA  = 'C. Henrik'
(4)  :- ffi('C. Henrik').                            (f4):  (4) ← (5)

(5)  □                                               (5) azonosan igaz
```

Bebizonyítottuk, hogy (1) teljesül az `NA = 'C. Henrik'` behelyettesítés esetén

A nagyszülő példa – a válasz követése az `answer` cél segítségével

```
% sz(Gy, Sz): Gy szülője Sz.
  sz('Imre',     'Gizella').    % (sz1)
  sz('Imre',     'István').     % (sz2)
  sz('István',   'Sarolt').     % (sz3)
  sz('István',   'Géza').       % (sz4)
  sz('Gizella',  'B. Gizella'). % (sz5)
  sz('Gizella',  'C. Henrik').  % (sz6)
```

```
% ffi(Személy): Személy férfi.
  ffi('Imre').        % (f1)
  ffi('István').      % (f2)
  ffi('Géza').        % (f3)
  ffi('C. Henrik').   % (f4)
```

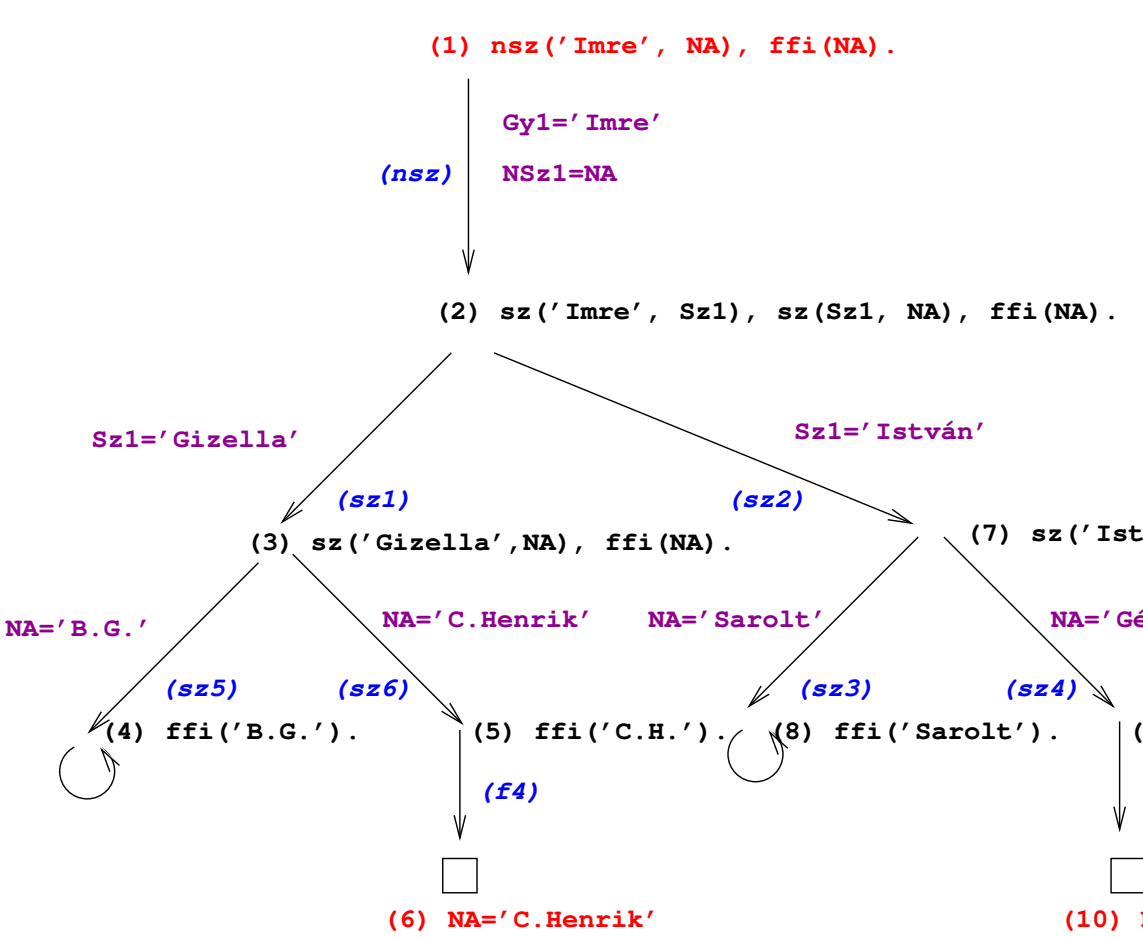
```
% Gy nagyszülője NSz.
nsz(Gy, NSz) :-        sz(Gy, Sz), sz(Sz, NSz).                    % (nsz)
```

```
(1)  :- nsz('Imre', NA), ffi(NA),                 answer(NA).      (nsz): (1) ← (2)
(2)  :- sz('Imre', Sz1), sz(Sz1, NA), ffi(NA), answer(NA).      (sz1): (2) ← (3)
(3)  :- sz('Gizella', NA), ffi(NA),               answer(NA).      (sz6): (3) ← (4)
(4)  :- ffi('C. Henrik'),                  answer('C. Henrik').    (f4):  (4) ← (5)
(5)  :-                                    answer('C. Henrik').    A lekérdezés sikeres
```

A futás végén az `answer` „virtuális” cél tartalmazza a választ.

A nagyszülő példa végrehajtása – keresési tér

- A Prolog minden lehetséges redukciót szisztematikusan végigpróbál,
- balról jobbra haladó mélységi keresés formájában.




```
sz('Imre',    'Gizella').    % (sz1)
sz('Imre',    'István').     % (sz2)
sz('István',  'Sarolt').     % (sz3)
sz('István',  'Géza').       % (sz4)
sz('Gizella', 'B. Gizella'). % (sz5)
sz('Gizella', 'C. Henrik').  % (sz6)
```

```
ffi('Imre').       % (f1)
ffi('István').     % (f2)
ffi('Géza').       % (f3)
ffi('C. Henrik').  % (f4)
```

```
nsz(Gy, NSz) :-
    sz(Gy, Sz),
    sz(Sz, NSz).   % (nsz)
```

A Prolog végrehajtási algoritmus – első közelítés

Egy célsorozat végrehajtása

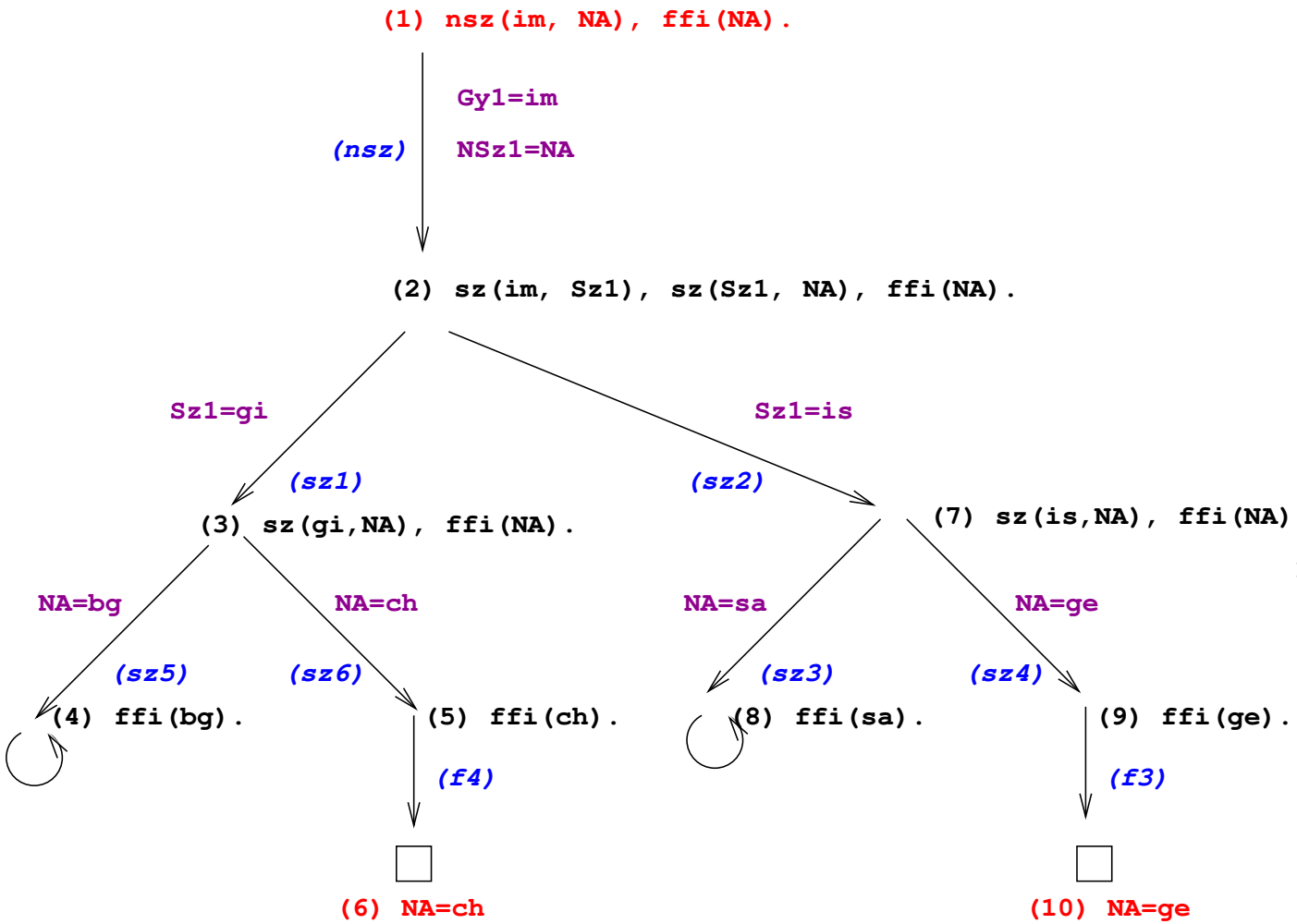
1. Ha az első cél beépített eljárást (BIP) hív, végrehajtjuk a BIP-et.
2. Ha az első cél felhasználói eljárásra vonatkozik, akkor megkeressük az eljárás első (visszalépés után: következő) olyan klózát, amelynek feje egyesíthető a hívással, és végrehajtjuk a redukciót.
3. Ha a redukció sikeres (találunk egyesíthető fejű klózt), folytatjuk a végrehajtást 1.-től az új célsorozattal.
4. Ha a redukció meghiúsul, akkor visszalépés következik:
   - visszatérünk a legutolsó, felhasználói eljárással történt (sikeres) redukciós lépéshez,
   - annak *bemeneti* célsorozatát megpróbáljuk *újabb* klózzal redukálni – ugrás a 2. lépésre
     (Ennek meghiúsulása értelemszerűen újabb visszalépést okoz.)

A végrehajtás nem „intelligens”

- Pl. `:- nsz(Gy, 'Géza').` hatékonyabb lenne ha a klóz törzsét jobbról balra hajtanánk végre
- DE: így a végrehajtás a program írója számára átlátható; a Prolog nem tételbizonyító, hanem programozási nyelv (WHAT rather than HOW)

A nagyszülő példa „tömörített” változata

- Imre herceget és felmenőit kétbetűs atomokkal jelöljük:
  Imre ⇒ `im`, Gizella ⇒ `gi`, István ⇒ `is`, Sarolt ⇒ `sa`, Géza ⇒ `ge`,
  Burgundi Gizella ⇒ `bg`, Civakodó Henrik ⇒ `ch`.




```
sz(im, gi).      % (sz1)
sz(im, is).      % (sz2)
sz(is, sa).      % (sz3)
sz(is, ge).      % (sz4)
sz(gi, bg).      % (sz5)
sz(gi, ch).      % (sz6)

ffi(im).         % (f1)
ffi(is).         % (f2)
ffi(ge).         % (f3)
ffi(ch).         % (f4)
```

```
nsz(U, NSz) :-
    sz(U, Sz),
    sz(Sz, NSz). % (nsz)
```

A cél-redukciós modell alapfogalmai

- A végrehajtás bemenete:
  - egy Prolog program (klózok sorozata), pl. a „nagyszülő” program, és
  - egy célsorozat, pl. `:- nsz(im, Sz).`
    a megoldás meghatározása érdekében ezt egy utolsó,
    `answer(Megoldás)` fiktív céllal bővítjük ki, pl.

```
:- nsz(im, NSz), answer(NSz).        % Kik Imre nagyszülei?
:- sz(Gy, Sz),   answer(Gy-Sz).      % Mik a gyerek-szülő párok?
```

- Az `answer(...)` cél segítségével követhetjük a megoldás felépülését
- Ha a célsorozat már csak az `answer` célt tartalmazza, akkor eljutottunk egy megoldáshoz (ezt a szerepet korábban az üres célsorozat játszotta)
- Az `answer` csak egy elméleti eszköz, nem beépített elj., de definálhatjuk, így: `answer(M) :- write(M), nl, fail.`
- A végrehajtásnak többféle kimenetele lehetséges:
  - Hiba (kivétel, exception), pl. `:- Y = alma, X is Y+1.`
    (Ezzel most nem foglalkozunk részletesebben.)
  - Meghiúsulás (nincs megoldás), pl. `:- sz(ge, Sz), answer(Sz).`
  - Siker (1 vagy több megoldás), pl. `:- sz(im, Sz), answer(Sz).`
- A végrehajtási modell gyakorlása ⇒ `https://ait.plwin.dev/P1-1`

A redukciós végrehajtás alapfogalmai (folyt.)

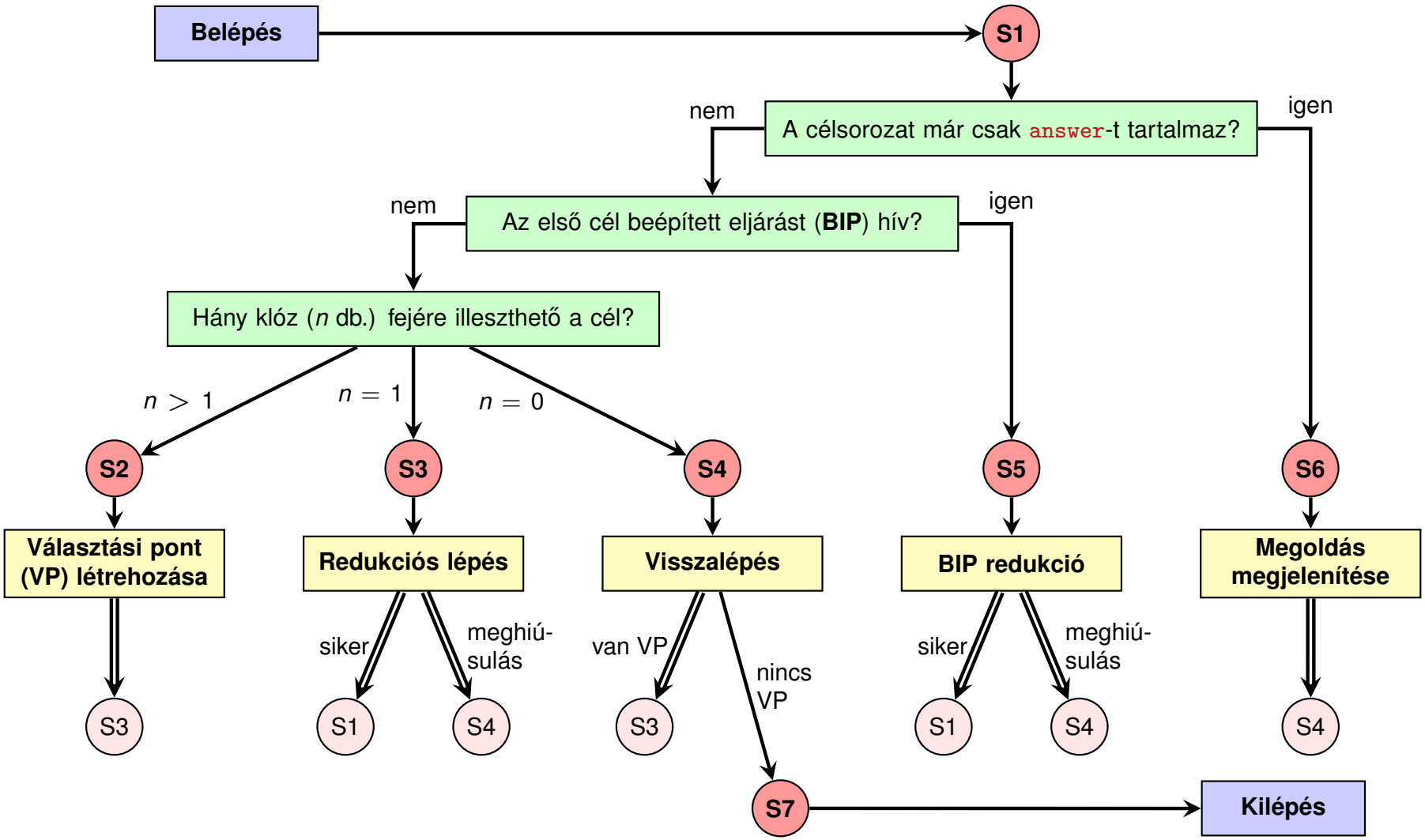
- A végrehajtás által használt (imperatív!) adatstruktúrák:
  - a jelenlegi célsorozatot tartalmazó változó (Goal)
  - a választási pontokat (VP) tartalmazó verem (Choice point stack)
- Például a `nsz(im, NA), ffi(NA), answer(NA)` célsorozat végrehajtásakor az alábbi VP verem jön létre:

Choice point stack

ChPoint name	Clause list	Goal	
CHP2	[sz5,sz6]	(3)	`sz(gi,NA),ffi(NA),answer(NA).`
CHP1	[sz1,sz2]	(2)	`sz(im,Sz),sz(Sz,NA),ffi(NA),answer(NA).`

- A VP verem akkor mélyül, ha 2 vagy több klózzal lehet redukálni
  - a redukció előtt a veremre elmentjük a célsorozatot és a redukcióban használható klózok listáját, majd folytatjuk a végrehajtást
  - ennek meghiúsulása esetén
    - a verem tetején levő klózlistából elhagyjuk az első elemet,
    - a klózlistában most első klózzal folytatjuk a redukciót,
    - ezt megelőzően, ha egyelemű a klózlista, megszűntetjük a VP-t
  - ha meghiúsuláskor üres a VP-verem $\Rightarrow$ kimerítettük a keresési teret

A redukciós modell folyamatábrája



A kettős nyilak jelentése: ugrás a halványlila körben megadott lépésre (folytatás a megfelelő sötétlila körnél).

Megjegyzések a folyamatábrához

- Hétféle végrehajtási lépésünk van: **S1**–**S7**, ahol **S1** a kiindulási pont (de közbülső is), **S7** a végállapot.
- Az **S1** lépés alapvető feladata az elágaztatás az **S2**–**S6** lépések egyikére
  - ha `Goal` már csak az `answer(...)` elemet tartalmazza ⇒ **S6**;
  - ha az első cél beépített eljárást hív ⇒ **S5**;
  - egyébként az első cél felhasználói eljárást hív. Ekkor megvizsgáljuk (általában csak közelítően), hogy az eljárás mely klózainak fejére illeszthető az első cél, és ezek száma (n) szerint ⇒ **S2**, **S3** vagy **S4**.
- **S2** létrehoz egy VP-t, majd az első klózzal redukál (⇒ **S3**).
- **S3** meghiúsulhat, ha **S1**-ben n csak közelítés volt, ilyenkor ⇒ **S4**.
- **S4** a VP-ban eltárolt következő klózzal való redukcióra lép (⇒ **S3**), ha van ilyen; egyébként befejezi a végrehajtást (⇒ **S7**).
- **S5** az **S3** lépéssel analóg módon vagy ⇒ **S1**, vagy ⇒ **S4**.
- **S6**-ban a megoldás megjelenítése után visszalépéssel folytatjuk (⇒ **S4**, további megoldások keresése).

A Prolog adatfogalma, a Prolog kifejezés (term)

- konstans (az `atomic` beépített eljárásnak felel meg)
  - számkonstans (`number`) – egész/lebegőpontos, pl. `1`, `-2.3`, `3.0e10`
  - névkonstans (`atom`), pl. `'István'`, `szuloje`, `*`, `++`, `tree_sum`
  - egy C konstans funktora $C/0$
- összetett- vagy struktúra-kifejezés (`compound`)
  - ún. kanonikus alak: $\langle$ struktúranév $\rangle(\langle \text{arg}_1 \rangle, \ldots, \langle \text{arg}_n \rangle)$
    - a $\langle$ struktúranév $\rangle$ egy névkonstans, az $\langle \text{arg}_i \rangle$ argumentumok tetszőleges Prolog kifejezések
    - a kifejezés funktora: $\langle$ struktúranév $\rangle$/n
    - példák: `person(ian,smith,2003)`, `<(X,Y)`, `is(X, +(Y,1))`
  - szintaktikus „édesítőszerek”, pl. operátorok és listák:
    `X is Y+1` $\equiv$ `is(X, +(Y,1))`, ill. `[1,2,3|X]` $\equiv$ `.(1,.(2,.(3,X)))`
- változó (`var`), pl. `Valtozo`, `X`, `_var`, `_` (don’t care) (nincs funktora)
  - a változó alaphelyzetben behelyettesítetlen, értékkel nem bír, egyesítés során egy tetszőleges Prolog kifejezést (akár egy másik változót) vehet fel értékül – dinamikus típusfogalom
  - a változó „first class citizen”, előfordulhat egy struktúra argumentumaként – logikai változó minta $\equiv$ adat

Néhány alapvető beépített eljárás (Built-In-Procedure, BIP)

- Kifejezések egyesítése
  - `X = Y`: az `X` és `Y` szimbolikus kifejezések egyesítése ≡ azonos alakra hozása változók esetleges behelyettesítésével, a lehető legáltalánosabb módon
  - `X \= Y`: az `X` és `Y` kifejezések nem egyesíthetőek (nem hozhatók azonos alakra)

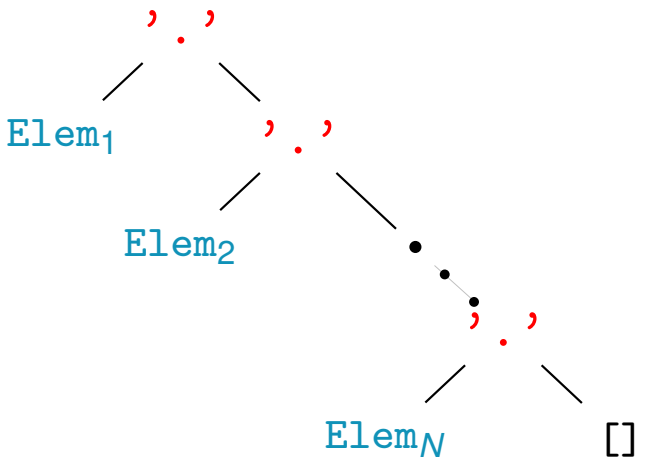
```
| ?- U+V = 1+(2*3).   ⟹   U = 1, V = 2*3
| ?- U-V = (8-4)-2.   ⟹   U = 8-4, V = 2
| ?- U+1 = 4+V.       ⟹   U = 4, V = 1 ?      % Kétirányú a mintaillesztés!
| ?- U+1 \= V+4.      ⟹   yes % szimbolikus kifejezések, a + nem kommutatív!
```

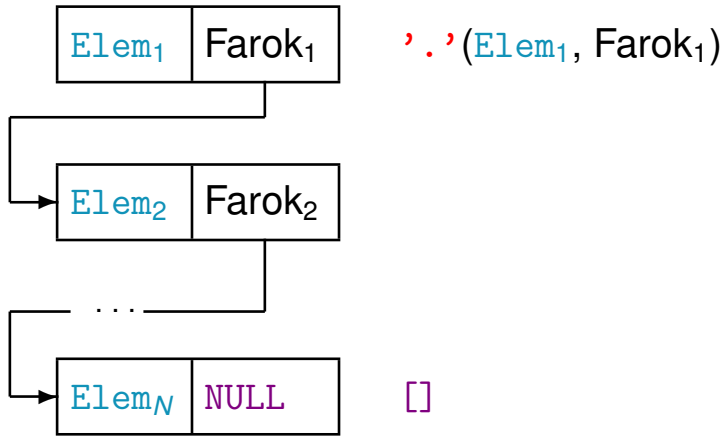
- Típusvizsgálat (az elemi, nem bontható típusok aláhúzással jelölve)
  - `var(X)`: `X` változó, ill. ennek komplementere: `nonvar(X)`: `X` nem változó
  - `atomic(X)`: `X` konstans
    - `number(X)`: `X` szám (`float(X)`: `X` lebegőp., `integer(X)`: `X` egész)
    - `atom(X)`: `X` névkonstans
  - `compound(X)`: `X` összetett kifejezés

```
| ?- X = 1, atomic(X), number(X), integer(X).  ⟹   yes
| ?- atomic(X), X = 1.                         ⟹   no   (What rather than How)
```

A Prolog lista-fogalma

- A Prolog lista
  - Az üres lista a `[]` névkonstans.
  - A nem-üres lista a `'.'(Fej,Farok)` struktúra:
    - `Fej` a lista feje (első eleme), míg
    - `Farok` a lista farka, azaz a fennmaradó elemekből álló lista.
  - A listákat egyszerűsítve is leírhatjuk („szintaktikus édesítés”).
  - Megvalósításuk optimalizált, időben és helyben is hatékonyabb.
- A listák fastruktúra alakja és megvalósítása





- Az SWI Prolog nem szabványos, a lista-konstruktor nem `'.'`, hanem `'[|]'` :-(((

Listák jelölése – szintaktikus „édesítőszerek”

- Az alapvető édesítés:
  `.(Fej,Farok)` helyett a `[Fej|Farok]` kifejezést írjuk
- Kiterjesztés N darab „fej”-elemre, a skatulyázás kiküszöbölése:
  $[\texttt{Elem}_1|[\ldots|[\texttt{Elem}_N|\texttt{Farok}]\ldots]] \Longrightarrow [\texttt{Elem}_1,\ldots,\texttt{Elem}_N|\texttt{Farok}]$
- Ha a farok `[]`, a „`|[]`” jelsorozat elhagyható:
  $[\texttt{Elem}_1,\ldots,\texttt{Elem}_N|[]] \Longrightarrow [\texttt{Elem}_1,\ldots,\texttt{Elem}_N]$
- Egy `Kif` Prolog kifejezés nyílt végű lista, ha `Kif` változó,
  vagy `Kif = [_|Farok]` ahol `Farok` nyílt végű lista
  (azaz ha előbb-utóbb egy változó található a farokpozíción)

```
| ?- [1,2] = [X|Y].          ⟹   X = 1, Y = [2] ?
| ?- [1,2] = [X,Y].          ⟹   X = 1, Y = 2 ?
| ?- [1,2,3] = [X|Y].        ⟹   X = 1, Y = [2,3] ?
| ?- [1,2,3] = [X,Y].        ⟹   no
| ?- [1,2,3,4] = [X,Y|Z].    ⟹   X = 1, Y = 2, Z = [3,4] ?
| ?- L = [1|_], L = [_,2|_]. ⟹   L = [1,2|_A] ? % nyílt végű!
| ?- L = .(1,[2,3|[]]).      ⟹   L = [1,2,3] ?
| ?- L = [1,2|.(3,[])].      ⟹   L = [1,2,3] ?
```

Egy egyszerű listakezelő eljárás

```
% unalmas(Lista, X): Lista minden eleme = X
unalmas([], _X).
unalmas([H|T], X) :-
    H = X,
    unalmas(T, X).
```

```
% Ekvivalens eljárás
unalmas([], _).
unalmas([X|T], X) :-

    unalmas(T, X).
```

```
| ?- unalmas([1,2,3], _).            ⟹   no
| ?- unalmas([2,2,2], 1).            ⟹   no
| ?- unalmas([2,2,2], 2).            ⟹   yes
| ?- unalmas([2,2,2], X).            ⟹   X = 2 ? ; no
| ?- L=[_,_,_], unalmas(L, 3).       ⟹   L = [3,3,3] ? ; no
| ?- L=[_,_,_], unalmas(L, X).       ⟹   L = [X,X,X] ? ; no
| ?- length(L, 10), unalmas(L, X).  ⟹   L = [X,X,X,X,X,X,X,X,X,X] ? ; no
| ?- length(L, 10), unalmas(L, X), X = 5.
      ⟹   L = [5,5,5,5,5,5,5,5,5,5], X = 5 ? ; no
```

Lista-komprehenzió: megoldások listába gyűjtése

- A `findall(Gyűjtő, Cél, Lista)` eljárás
  - a `Cél` kifejezést eljáráshívásként értelmezi;
  - a `Cél` eljárást meghívja, és minden sikeres lefutása után a `Gyűjtő` adatstruktúra másolatát elmenti;
  - a `Cél` hívás keresési terének kimerítésekor a `Gyűjtő` változó összes elmentett másolatát (a keletkezésük sorrendjében) egy listába gyűjti és ezt egyesíti a `Lista` kimenő paraméterrel.
- Példák az eljárás használatára:

```
| ?- findall(NSz, nsz('Imre', NSz), NSzk).
        ⟹    NSzk = ['B. Gizella','C. Henrik','Sarolt','Géza'] ? ; no
| ?- findall(Sz, sz('István',Sz), Szulok).
        ⟹    Szulok = ['Sarolt','Géza'] ? ; no
| ?- findall(Gy, sz(Gy, 'István'), Gyerekek).
        ⟹    Gyerekek = ['Imre'] ? ; no
| ?- findall(Gy, sz(Gy, 'Imre'), Gyerekek).
        ⟹    Gyerekek = [] ? ; no
```

- Ha nincs megoldás, akkor (értelemszerűen) egy üres listát kapunk eredményül.

Aritmetikai beépített eljárások

- Egy aritmetikai kifejezés[43] (`AKif`) a BIP végrehajtásakor kötelezően:
  - tömör (ground) – behelyettesítetlen változót nem tartalmaz;
  - csak számokból és megengedett aritmetikai függvényekből áll
- A legfontosabb (2-arg.-ú) függvények: `+`, `-`, `*`, `/` (lebegőp. eredményt ad), `//` (egész-osztás, 0 felé kerekít), `rem` (maradék, `//` szerint)
- `X is AKif`: Az `AKif` aritmetikai kif. értékét egyesíti `X`-szel, pl.

```
| ?- X = 2, Y is X+1.                  ⟹  X = 2, Y = 3 ?; no
| ?- Y is X+1, X = 2.                  ⟹  ! Instantiation error
| ?- 3 is 2+1.                         ⟹  yes
| ?- 1+3 is 6-2.                       ⟹  no % a bal oldalt nem értékeli ki!
| ?- X = 1, Y is (X-27) rem (X+2).     ⟹  X = 1, Y = -2 ?; no
| ?- Kif = X/(X-1), X = 6, Y is Kif.   ⟹  Kif = 6/(6-1), X = 6, Y = 1.2 ?; no
```

- További aritmetikai BIP-ek: `AKif1 < AKif2`, `AKif1 > AKif2`, `AKif1 =< AKif2` (vigyázat: nem <=), `AKif1 >= AKif2`, `AKif1 =:= AKif2` (aritmetikailag egyenlő), `AKif1 =\= AKif2` (aritmetikailag nem-egyenlő) – ezek mindkét oldalt kiértékelik, és elvégzik a kért összehasonlítást:

```
| ?- 1+3 =:= 6-2.                      ⟹  yes
| ?- 1+1 =\= 6/3.                      ⟹  no
```

[43] pl. `https://sicstus.sics.se/sicstus/docs/latest/html/sicstus/ref_002dari_002daex.html`

Példa: faktoriális számítása Prologban

- Funkc. nyelven a faktoriális egy 1-argumentumú függvény: `F = fakt(N)`
- Prologban ennek egy kétargumentumú reláció felel meg: `fakt(N, F)`
- Konvenció: az utolsó argumentum(ok) a kimenő pararaméter(ek)
- Idézzük föl a faktoriális függvény Elixir megvalósítását:

```
def fakt0(0)              do 1               end
def fakt0(n) when n > 0   do n * fakt0(n-1)  end
```

- Írjuk át úgy, hogy a `fakt0` hívás különüljön el az egyéb számításoktól:

```
def fakt1(0)              do                                          1 end
def fakt1(n) when n > 0 do n1 = n-1; f1 = fakt1(n1); f = f1*n; f end
```

- Az „`F = fakt1(N)`” $\Longrightarrow$ `fakt(N, F)` transzformáció adja a Prolog kódot:

```
% fakt(N, F): F = N!.
fakt(0, 1).                    % 0! = 1.
fakt(N, F) :-                  % N! = F ha létezik olyan N1 és F1, hogy
    N > 0, N1 is N-1,          %    N > 0 és N1 = N-1 és
    fakt(N1, F1),              %    F1 = N1! és
    F is F1*N.                 %    F = F1*N.
```

```
| ?- fakt(5, F).  ⟹  F = 120 ? ; no
| ?- fakt(4, 24). ⟹  yes
| ?- fakt(0, F).  ⟹  F = 1 ? ; no
```

```
| ?- fakt(0, 2).  ⟹  no
| ?- fakt(N, 1).  ⟹  N = 0 ? ;
                      ! Inst.  err...
```

Aritmetika plusz lista-komprehenzió

Egy korábbi Elixir példa: gyűjtsük össze azokat a pitagoraszi számhármasokat, amelyek összege egy adott `N` számnál kisebb-egyenlő

```
:- use_module(library(between)). % SWI Prologban elhagyandó

% pitag(N, Hármasok):
% Hármasok = { p(A,B,C) | 1 =< A < B < C, A+B+C =< N, A*A + B*B = C*C }
pitag(N, Pk) :-
    findall(

      p(A,B,C),
      ( between(1, N, A),
        between(1, N, B), A < B,
        between(1, N, C), A+B+C =< N,
        A*A + B*B =:= C*C
      ),
      Pk).
```

```
% 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}
% )
```

```
| ?- pitag(12, Pk).      ⟹    Pk = [p(3,4,5)] ? ; no
| ?- pitag(36, Pk).
     ⟹    Pk = [p(3,4,5),p(5,12,13),p(6,8,10),p(9,12,15)] ? ; no
```

Programfejlesztési beépített predikátumok

- `consult(File)`: A `File` állományban levő programot beolvassa és értelmezendő alakban eltárolja. (`File = user` $\Rightarrow$ terminálról olvas.)
- `compile(File)` vagy `[File]`: mint `consult`, csak kompilált alakban tárol (gyorsabb kód, de egyes BIP-ek nem nyomkövethetők)
- `trace`, `notrace`: A (teljes) nyomkövetést be- ill. kikapcsolja.
- `listing` vagy `listing(Predikátum)`: Az értelmezendő alakban eltárolt összes ill. adott nevű predikátumokat kilistázza.
- `halt`: A Prolog rendszer befejezi működését.

```
> sicstus
SICStus 4.4.1 (x86_64-linux-glibc2.12) ...
| ?- consult(fakt).
% consulted /home/user/fakt.pl in module user, 10 msec 91776 bytes
yes
| ?- fakt(4, F).
F = 24 ? ;
no
| ?- listing(fakt).
(...)
yes
| ?- halt.
>
```

Adatstruktúrák Prologban – a bináris fák példája

- A bináris fa adatstruktúra
  - vagy egy csomópont (`node`), amelynek két részfája van (`left,right`)
  - vagy egy levél (`leaf`), amely egy egészt tartalmaz

Binárisfa-struktúra C-ben

```
enum treetype {Node, Leaf};
struct tree {
 enum treetype type;
 union {
   struct { struct tree *left;
            struct tree *right;
          } nd;
   struct { int value;
          } lf;
 } u;
};
```

A Prolog dinamikusan típusos nyelv – nincs szükség explicit típusdefinícióra

- Mercury típusleírás (komment)

```
% :- type tree --->
%       node(tree, tree)
%     | leaf(int).
```

- A típushoz tartozás ellenőrzése

```
% is_tree(T): T egy bináris fa
is_tree(leaf(V)) :- integer(V).
is_tree(node(Left,Right)) :-
    is_tree(Left),
    is_tree(Right).
```

Bináris fák összegzése

Egy bináris fa levélösszegének kiszámítása:

- egylevelű fa esetén a levélben tárolt egész
- csomópont esetén a két részfa levélösszegének összege

C nyelven

```
% S = tsum(T): T levélösszege S
int tsum(struct tree *tree)
{
 switch(tree->type) {
 case Leaf:
  return tree->u.lf.value;
 case Node:
  return tsum(tree->u.nd.left) +
         tsum(tree->u.nd.right);
   }
}
```

Prologban

```
% tree_sum(Tree, S): Σ Tree = S.
tree_sum(leaf(Value), Value).
tree_sum(node(Left,Right), S) :-
        tree_sum(Left, S1),
        tree_sum(Right, S2),
        S is S1+S2.
| ?- tree_sum(node(leaf(5),
                   node(leaf(3),
                        leaf(2))),S).
S = 10 ? ;
no
| ?- tree_sum(T, 3).
T = leaf(3) ? ;
! Inst. error in argument 2 of is/2
! goal:  3 is _73+_74
```

Tartalom

Predikátumok, klózok

- Példa:

```
% két klózból álló predikátum definíciója, funktora: tree_sum/2
tree_sum(leaf(Val), Val).          %                    1. klóz, tényáll.
tree_sum(node(Left,Right), S) :- %           fej    \
        tree_sum(Left, S1),        % cél     \       |
        tree_sum(Right, S2),       % cél     | törzs | 2. klóz, szabály
        S is S1+S2.                % cél     /       /
```

- Szintaxis:

⟨ Prolog program ⟩	::=	⟨ predikátum ⟩ . . .	
⟨ predikátum ⟩	::=	⟨ klóz ⟩ . . .	{azonos funktorú}
⟨ klóz ⟩	::=	⟨ tényállítás ⟩.␣ |	
		⟨ szabály ⟩.␣	{klóz funktora = fej funktora}
⟨ tényállítás ⟩	::=	⟨ fej ⟩	
⟨ szabály ⟩	::=	⟨ fej ⟩ :- ⟨ törzs ⟩	
⟨ törzs ⟩	::=	⟨ cél ⟩, . . .	
⟨ cél ⟩	::=	⟨ kifejezés ⟩	
⟨ fej ⟩	::=	⟨ kifejezés ⟩	

Prolog kifejezések

- Példa – egy klózfej mint kifejezés:

```
% tree_sum(node(Left,Right), S)    % összetett kif., funktora
% ________ _________________  _                      tree_sum/2
%      |          |            |
% struktúranév    \          argumentum, változó
%                  \- argumentum, összetett kif.
```

- Szintaxis:

⟨ kifejezés ⟩	::=	⟨ változó ⟩ |	{Nincs funktora}
		⟨ konstans ⟩ |	{Funktora: ⟨ konstans ⟩/0}
		⟨ összetett kif. ⟩ |	{Funktor: ⟨ struktúranév ⟩/⟨ arg.sz. ⟩}
		⟨ egyéb kifejezés ⟩	{Operátoros, lista, stb.}
⟨ konstans ⟩	::=	⟨ névkonstans ⟩ |	
		⟨ számkonstans ⟩	
⟨ számkonstans ⟩	::=	⟨ egész szám ⟩ |	
		⟨ lebegőp. szám ⟩	
⟨ összetett kif. ⟩	::=	⟨ struktúranév ⟩ (⟨ argumentum ⟩, . . .)	
⟨ struktúranév ⟩	::=	⟨ névkonstans ⟩	
⟨ argumentum ⟩	::=	⟨ kifejezés ⟩	

Lexikai elemek: példák és szintaxis

```
% változó:              Fakt FAKT _fakt X2 _2 _
% névkonstans:         fakt ≡ 'fakt' 'István' [] ; ',' += ** \= ≡ '\\='
% számkonstans:        0 -123 10.0 -12.1e8
% nem névkonstans:     !=, Istvan
% nem számkonstans:    1e8 1.e2
```

⟨ változó ⟩ ::= ⟨ nagybetű ⟩⟨ alfanum. jel ⟩... |
_ ⟨ alfanum. jel ⟩...

⟨ névkonstans ⟩ ::= ’⟨ idézett kar. ⟩...’ |
⟨ kisbetű ⟩⟨ alfanum. jel ⟩... |
⟨ tapadó jel ⟩... | ! | ; | [] | {}

⟨ egész szám ⟩ ::= {előjeles vagy előjeltelen számjegysorozat}

⟨ lebegőp.szám ⟩ ::= {belsejében tizedespontot tartalmazó
számjegysorozat esetleges exponenssel}

⟨ idézett kar. ⟩ ::= {tetszőleges nem ' és nem \ karakter} |
\ ⟨ escape szekvencia ⟩

⟨ alfanum. jel ⟩ ::= ⟨ kisbetű ⟩ | ⟨ nagybetű ⟩ | ⟨ számjegy ⟩ | _

⟨ tapadó jel ⟩ ::= + | - | * | / | \ | $ | ^ | < | > | = | ` | ~ | : | . | ? | @ | # | &

Prolog programok formázása

- Megjegyzések (comment)
  - A % százalékjeltől a sor végéig
  - A /* jelpártól a legközelebbi */ jelpárig.
- Formázó elemek (komment, szóköz, újsor, tabulátor stb.) szabadon használhatók, kivételek:
  - összetett kifejezésben a név után tilos formázó elemet tenni
  - prefix operátor (ld. később) és „(” között kötelező a formázó elem
  - klózt lezáró pont (.␣): önmagában álló pont (ha előtte tapadó jel áll, akkor a pont elé formázó elemet kell tenni), amit legalább egy formázó elem követ
- Programok javasolt formázása:
  - Az egy predikátumhoz tartozó klózok legyenek egymás mellett a programban, közéjük ne tegyünk üres sort
  - A predikátum elé tegyünk egy üres sort és egy fejkommentet:

```
% predikátumnév(A1, ..., An): A1, ..., An közötti
% összefüggést leíró kijelentő mondat.
```

  - A klózfejet írjuk a sor elejére, minden célt lehetőleg külön sorba, néhány szóközzel beljebb kezdve

Elixir és Prolog: néhány eltérés és hasonlóság

Elixir	**Prolog**
függvény, értéke tetsz. típusú	predikátum, azaz Boole-értékű függvény
arg. bemenő, a fv.érték kimenő	arg.-ok bemenők és kimenők is
egyetlen visszatérési érték	választási pontok, több megoldás lehet
külön ennes, lista típusok	a lista is összetett kifejezés
nincsenek felh. operátorok	felhasználói operátorok definiálhatók
Az = jobb oldalán tömör kif., bal oldalon mintakif.; őrfeltételekkel	az egyesítés szimmetrikus, mindkét oldalon minták

- Néhány hasonlóság:
  - az eljárás is klózokból áll, kiválasztás mintaillesztéssel, sorrendben, de míg Elixirben csak az első illeszkedő klózfej számít, Prologban az összes
  - változóhoz csak egyszer köthető érték
  - lista szintaxisa (de: Elixirben önálló típus), sztring (füzér), atom

Tartalom

Listák összefűzése – az `append/3` eljárás

- Elixir megoldás:

```
def append([],    b) do                    b     end
def append([x|a], b) do c = append(a,b); [x|c] end
```

- Írjuk át a kétargumentumú `append` függvényt egy `app0/3` Prolog eljárássá!

```
% app0(A, B, C): A és B listák összefűzése a C lista, C = A ⊕ B
app0([], B, Ret) :- Ret = B.
app0([X|A], B, Ret) :-
    app0(A, B, C), Ret = [X|C].
```

- Logikailag tiszta Prolog programokban a `Vált = Kif` alakú hívások elhagyhatók, ha `Vált` többi előfordulását `Kif`-re cseréljük.

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

- Mindkét eljárásban a (max) futási idő arányos az 1. arg. hosszával
- Miért jobb az `app/3` mint az `app0/3`?
  - `app/3` jobbrekurzív, ciklussal ekvivalens (nem fogyaszt vermet)
  - `app([1,...,1000],[0],[2,...])` 1, `app0(...)` 1000 lépésben hiúsul meg.
  - `app/3` használható szétszedésre is (lásd később), míg `app0/3` nem.

Lista építése *elölről* – nyílt végű listákkal

- **Ismétlés:** egy `X` Prolog kifejezés nyílt végű lista, ha `X` változó, vagy `X = [_|Farok]` ahol `Farok` nyílt végű lista.

  `| ?- L = [1|_], L = [_,2|_].` $\Longrightarrow$ `L = [1,2|_A] ?`

- A beépített `append/3` azonos az `app/3`-mal:

```
append([], B, B).
append([X|A], B, [X|C]) :-
    append(A, B, C).
```

- Az `append` eljárás már az első redukciónál felépíti az eredmény fejét!
  - Példa-célsorozat: `append([1,2], [3,4,5], Ered), answer(Ered).`
  - Fej: `append([X|A], B, [X|C])`
  - Behelyettesítés: `X = 1, A = [2], B = [3,4,5], Ered = [1|C]`
  - Új célsorozat: `append([2], [3,4,5], C), answer([1|C]).`

  (`Ered` nyílt végű lista, farka még behelyettesítetlen.)

Lista építése *elölről* – a megvalósítás részletei

- A kimenő paraméter behelyettesítését explicitté tehetjük:

```
app1([], B, L) :-                                   % (a1)
    L = B.
app1([X|A], B, L) :-                                % (a2)
    L = [X|C],
    app1(A, B, C).
```

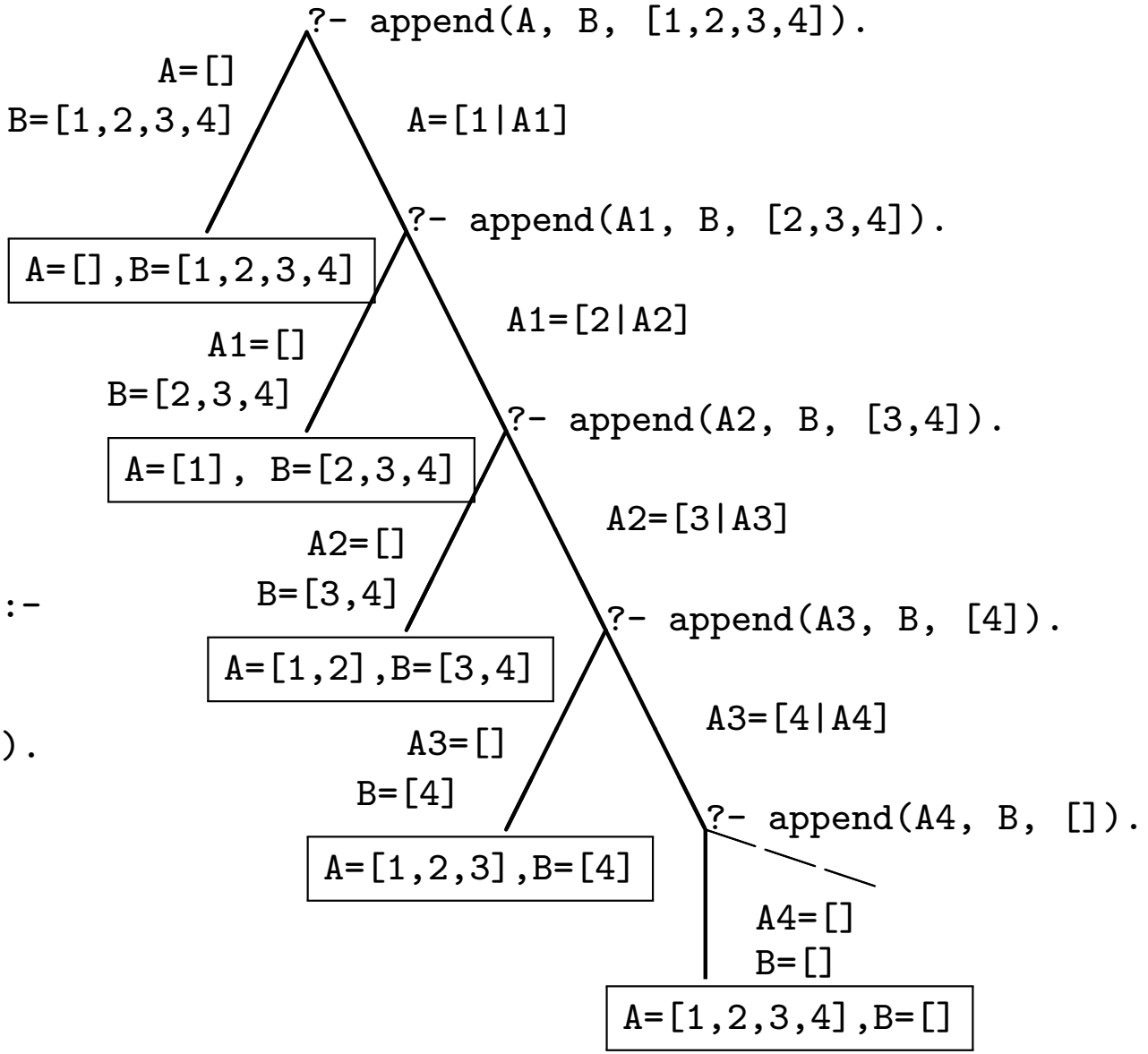
- Egy `app1/3` eljáráshívás redukciós lépései:

```
:- app1([1,2], [3,4,5], Ered), answer(Ered).                    % (cs0)
                                                  % + (a2) =>
:- Ered = [1|C1], app1([2], [3,4,5], C1), answer(Ered).         % (cs1)
                                                  % + BIP  =>
:- app1([2], [3,4,5], C1), answer([1|C1]).                      % (cs2)
                                                  % + (a2) =>
:- C1 = [2|C2], app1([], [3,4,5], C2), answer([1|C1]).          % (cs3)
                                                  % + BIP  =>
:- app1([], [3,4,5], C2), answer([1,2|C2]).                     % (cs4)
                                                  % + (a1) =>
:- C2 = [3,4,5], answer([1,2|C2]).                              % (cs5)
                                                  % + BIP  =>
:- answer([1,2,3,4,5]).                                         % (cs6)
```

Listák szétbontása az append/3 segítségével

```
% append(L1, L2, L3):
% Az L3 lista az L1 és L2
% listák elemeinek egymás
% után fűzésével áll elő.
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).

| ?- append(A, B, [1,2,3,4]).
A = [], B = [1,2,3,4] ? ;
A = [1], B = [2,3,4] ? ;
A = [1,2], B = [3,4] ? ;
A = [1,2,3], B = [4] ? ;
A = [1,2,3,4], B = [] ? ;
no
```



Nyílt végű listák az append változatokban

```
app0([], L, L).
app0([X|L1], L2, R) :-
    app0(L1, L2, L3), R = [X|L3].
```

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).
```

- Ha az 1. argumentum zárt végű (n hosszú), mindkét változat legfeljebb $n+1$ lépésben egyértelmű választ ad, amely lehet nyílt végű:
  `| ?- app0([1,2], L2, L3).` $\Longrightarrow$ `L3 = [1,2|L2] ? ; no`
- A 2. arg.-ot nem bontjuk szét $\Longrightarrow$ mindegy, hogy nyílt vagy zárt végű
- Ha a 3. argumentum zárt végű (n hosszú), akkor az `append` változat legfeljebb $n+1$ megoldást ad, max. $\sim 2n$ lépésben (ld. előző dia); tehát:
  - `append(L1, L2, L3)` keresési tere véges, ha `L1` vagy `L3` zárt
- Ha az 1. és a 3. arg. is nyílt, akkor a válaszhalmaz csak végtelen sok Prolog kifejezéssel fedhető le, pl.
  `_` $\oplus$ `[1] = L` ($\equiv$`L` utolsó eleme `1`): `L = [1]; [_,1]; [_,_,1]; ...`
- `app0` szétszedésre nem jó, mert pl. `app0(L, [1], [])` $\Longrightarrow \infty$ ciklus, hiszen redukálva a 2. klózzal $\Longrightarrow$ `app0(L1, [1], L3), [] = [X|L3].`
- Az `append` eljárás jobbrekurzív, hála a logikai változó használatának

Variációk `append`-re – három lista összefűzése (kieg. anyag)

- `append(L1,L2,L3,L123)`: $(\texttt{L1} \oplus \texttt{L2}) \oplus \texttt{L3} = \texttt{L123}$

```
append(L1, L2, L3, L123) :-
        append(L1, L2, L12), append(L12, L3, L123).
```

- Lassú, pl.: `append([1,...,100],[1,2,3],[1], L)` 103 helyett 203 lépés!
- Szétszedésre nem alkalmas – végtelen választási pontot hoz létre
- Szétszedésre is alkalmas, hatékony változat

```
% L1 ⊕ (L2 ⊕ L3) = L123,
% ahol vagy L1 és L2, vagy L123 adott (zárt végű).
append(L1, L2, L3, L123) :-
        append(L1, L23, L123), append(L2, L3, L23).
```

  - `append(+,+,?,?)` esetén az első `append/3` hívás nyílt végű listát ad:

    `| ?- append([1,2], L23, L).` $\Longrightarrow$ `L = [1,2|L23] ?`
  - Az `L3` argumentum behelyettesítettsége (nyílt vagy zárt végű lista-e) nem számít.

Listák megfordítása

- Naív (négyzetes lépésszámú) megoldás

```
% nrev(L, R): R = L megfordítása.
nrev([], Ret) :- Ret = [].          % def nrev([])    do []  end
nrev([X|L], Ret) :-                 % def nrev([x|l]) do
    nrev(L, RL),                    %         rl = nrev(l);
    append(RL, [X], Ret).           %         append(rl,[x]) end
```

- Lineáris lépésszámú megoldás

```
% revapp(L0, R0, R): L0 megfordítását R0 elé fűzve kapjuk R-t.
revapp([], R0, R) :- R = R0.        % def revapp([], r0)     do r0 end
revapp([X|L0], R0, R) :-            % def revapp([x|l0], r0) do
    revapp(L0, [X|R0], R).          %         revapp(l0, [x|r0])   end

% reverse(R, L): Az R lista az L megfordítása.
reverse(R, L) :-  revapp(L, [], R).
```

- `revapp`-ban *R0*, *R* egy akkumulátorpár: eddigi ill. végeredmény
- A `lists` könyvtár tartalmazza a `reverse/2` eljárás definícióját, betöltése:

```
:- use_module(library(lists)).
```

Listák gyűjtése elölről és hátulról (kieg. anyag)

- Prolog

```
revapp([], L, L).
revapp([X|L0], L2, L3) :-
    revapp(L0, [X|L2], L3).
```

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).
```

- C++

```
struct lnk  { char elem;
              lnk *next;
              lnk(char e): elem(e) {}   };

typedef lnk *list;
list revapp(list L1, list L2)
{ list l = L2;
  for (list p=L1; p; p=p->next)
  { list newl = new lnk(p->elem);
    newl->next = l; l = newl;
  }
  return l;
}
```

```
list append(list L1, list L2)
{ list L3, *lp = &L3;
  for (list p=L1; p; p=p->next)
  { list newl = new lnk(p->elem);
    *lp = newl; lp = &newl->next;
  }
  *lp = L2;  return L3;
}
```

Keresés listában – a `member/2` beépített eljárás

- `member(E, L)`: `E` az `L` lista eleme

```
member(Elem, [Elem|_]).
member(Elem, [_|Farok]) :-
    member(Elem, Farok).
```

- Eldöntendő (igen-nem) kérdés:

```
| ?- member(2, [1,2,3,2]).          ⟹   yes                          DE
| ?- member(2, [1,2,3,2]), R=yes.   ⟹   R=yes ? ; R=yes ? ; no
```

- Lista elemeinek felsorolása:

```
| ?- member(X, [1,2,3]).            ⟹   X = 1 ? ; X = 2 ? ; X = 3 ? ; no
| ?- member(X, [1,2,1]).            ⟹   X = 1 ? ; X = 2 ? ; X = 1 ? ; no
```

- Listák közös elemeinek felsorolása – az előző két hivásformát kombinálja:

```
| ?- member(X, [1,2,3]),
     member(X, [5,4,3,2,3]).        ⟹   X = 2 ? ; X = 3 ? ; X = 3 ? ; no
```

- Egy értéket egy (nyílt végű) lista elemévé tesz, végtelen választás!

```
| ?- member(1, L).                  ⟹   L = [1|_A] ? ; L = [_A,1|_B] ? ;
                                        L = [_A,_B,1|_C] ? ; ...
```

- A `member/2` keresési tere véges, ha 2. argumentuma zárt végű lista.

A `member/2` predikátum általánosítása: `select/3`

- `select(E, Lista, M)`: `E` elemet `Lista`ból pont egyszer elhagyva marad `M`.

```
select(E, [E|Marad], Marad).     % Elhagyjuk a fejet, marad a farok.
select(E, [X|Farok], [X|M]) :-   % Marad a fej,
        select(E, Farok, M).     %    a farokból hagyunk el elemet.
```

- Felhasználási lehetőségek:

```
| ?- select(1, [2,1,3,1], L).    % Adott elem elhagyása
    ⟹      L = [2,3,1] ? ; L = [2,1,3] ? ; no
| ?- select(X, [1,2,3], L).      % Akármelyik elem elhagyása
    ⟹      L=[2,3], X=1 ? ;  L=[1,3], X=2 ? ;   L=[1,2], X=3 ? ; no
| ?- select(3, L, [1,2]).        % Adott elem beszúrása!
    ⟹      L = [3,1,2] ? ; L = [1,3,2] ? ; L = [1,2,3] ? ; no
| ?- select(3, [2|L], [1,2,7,3,2,1,8,9,4]).
     % Beszúrható-e 3 az [1,...]-ba úgy, hogy [2,...]-t kapjunk?
    ⟹      no
| ?- select(1, [X,2,X,3], L).
    ⟹      L = [2,1,3], X = 1 ? ; L = [1,2,3], X = 1 ? ; no
```

- A `select/3` eljárás keresési tere véges, ha vagy a 2., vagy a 3. argumentuma zárt végű lista (a `lists` könyvtár tartalmazza az eljárást)

Listák permutációja (kieg. anyag)

- `% perm(+Lista, ?Perm)`: `Lista` permutációja a `Perm` lista.

```
perm0([], []).
perm0([Elso|Lista], Perm) :-
        perm0(Lista, Perm0),          % permutáljuk a bemenet farkát
        select(Elso, Perm, Perm0).    % ebbe beszúrjuk a bemenet fejét
```

- Felhasználási példák:

```
| ?- perm0([1,2], L).
    ⟹     L = [1,2] ? ; L = [2,1] ? ; no

| ?- perm0([a,b,c], L).
    ⟹     L = [a,b,c] ? ; L = [b,a,c] ? ; L = [b,c,a] ? ;
           L = [a,c,b] ? ; L = [c,a,b] ? ; L = [c,b,a] ? ; no

| ?- perm0(L, [1,2]).
    ⟹      L = [1,2] ? ;  végtelen keresési tér
```

- Ha `perm0/2`-ben az első argumentum változó, akkor a rekurzív hívás mindkét argumentuma változó lesz ⇒végtelen sok választás
- A `lists` könyvtárban van egy kétirányban működő `permutation/2` eljárás

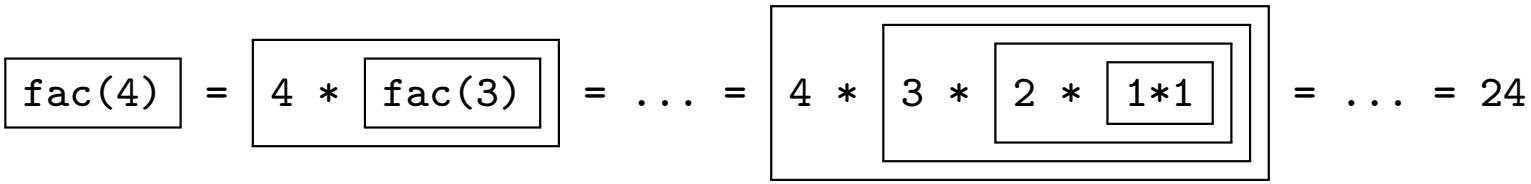
Tartalom

Függvények és eljárások egymásba skatulyázása

- A deklaratív nyelvekben a rekurzió váltja ki a ciklust, így gyakran előfordulnak egymásba **skatulyázott** függvény- ill. eljáráshívások.
- Tekintsük a faktoriális Elixir definícióját!

```
% fac(N) = N faktoriálisa.
def fac(0), do: 1
def fac(n), do: n * fac(n-1)
```

- A `fac(4)` függvényhívás végrehajtásakor pl. az alábbi állapotokat kapjuk:



- A függvényhívásokba való be- és kilépés nyomon követése:

```
Call fac(4)
Call fac(3)
...
Call fac(0)
Exit fac(0) = 1
...
Exit fac(3) = 6
Exit fac(4) = 24
```

A `Call` nyomkövetési információ egy fenti doboz *létrehozásához* kapcsolható, míg az `Exit` a doboz kiértékelésének *befejezéséhez*.

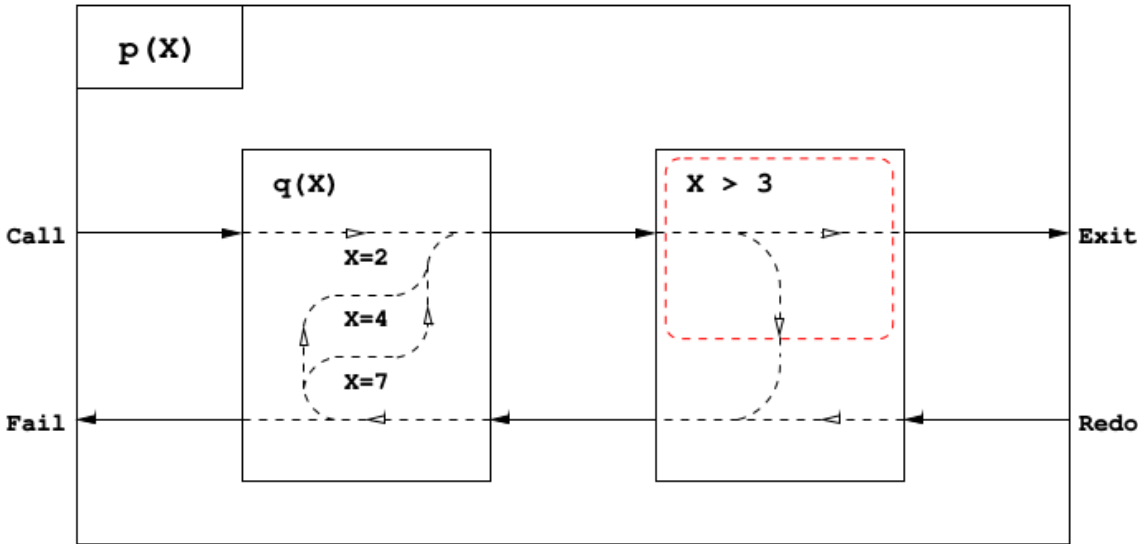
Prolog nyomkövetés eljárás-doboz modellel

- A Prolog doboz alapú nyomkövetésében is az eljárások be- és kilépési pontjain (ún. kapukon, angolul *port*) való áthaladásról kapunk információt:
  - **Call port** (hívás kapu) – belépés az eljárásba, doboz létrehozása
  - **Exit port** (kilépés kapu) – sikeres lefutás, esetleg doboz törlése
  - **Fail port** (meghiúsulás kapu) – sikertelen lefutás, doboz törlése
  - **Redo port** (újra kapu) – új megoldás kérése
- A Prolog eljárás-végrehajtás két fázisa
  - előre menő: egymásba **skatulyázott eljárás-be** és **-ki**lépések
  - visszafelé menő: **új megoldás** kérése egy már lefutott eljárástól
- Prolog végrehajtás objektum-orientált szemléletben (eljárás $\Rightarrow$ objektum):
  - eljárás meghívása (hívás kapu): objektum létrehozása
  - sikeres lefutás (kilépés kapu): változóbehelyettesítések visszaadása
  - sikertelen lefutás (meghiúsulás kapu): meghiúsulás jelzése
  - új megoldás kérése (újra kapu): további választási pont(ok) bejárása

Eljárás-doboz modell – grafikus szemléltetés

Egy egyszerű példaprogram, hívása `| ?- p(X).`

```
q(2).  q(4).  q(7).          p(X) :- q(X), X > 3.
```



Előre: `Call p(X); Call q(X); Exit q(2); Call 2>3; Fail 2>3`

Vissza: `Redo q(2);`

Előre: `Exit q(4); Call 4>3; Exit 4>3; Exit p(4);` siker `X = 4 ? ;`

Vissza: `Redo p(4); Redo 4>3; Fail 4>3; Redo q(4);`

Előre: `Exit q(7); Call 7>3; Exit 7>3; Exit p(7);` siker `X = 7 ? ;`

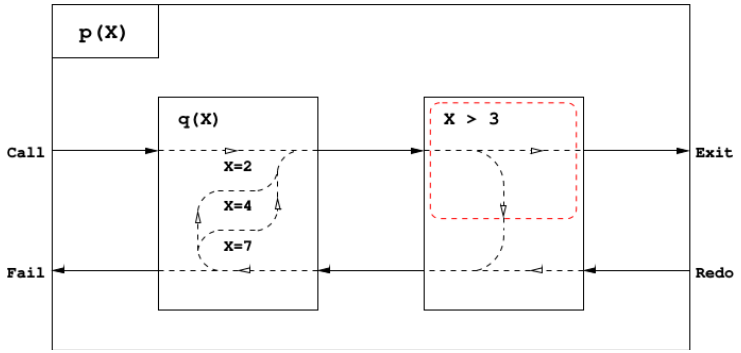
Vissza: `Redo p(7); Redo 7>3; Fail 7>3; ...; Fail p(X);` meghiúsulás `no`

Egy egyszerű nyomkövetési példa (SICStus Prolog)

- SICStusban `?...Exit` jelzi, hogy van választási pont a lefutott eljárásban
- Ha nincs `?` az Exit kapunál, akkor a doboz törlődik (lásd a szaggatott piros téglalapot az `X > 3` hívás körül)

```
q(2).
q(4).
q(7).

p(X) :- q(X), X > 3.
```

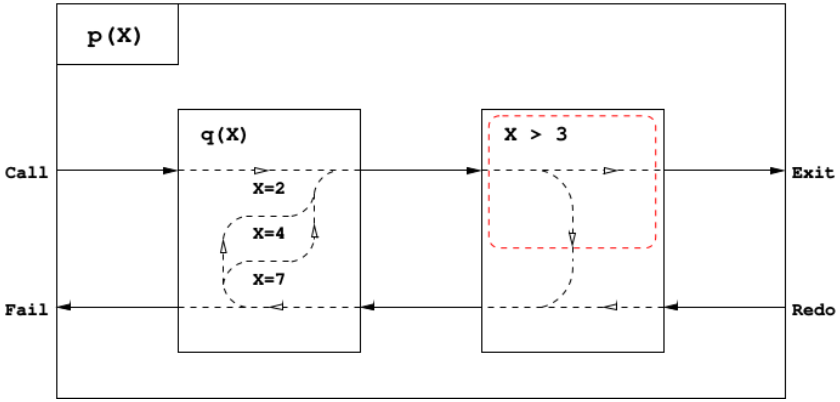



```
% Sorszám      Mélység
| ?- consult(pq0), trace, p(X).     % compile esetén a > /2 hívásokat nem látjuk
        1      1 Call: p(_463) ?
        2      2 Call: q(_463) ?
?       2      2 Exit: q(2) ?      % ? ≡ maradt választási pont q-ban
        3      2 Call: 2>3 ?
        3      2 Fail: 2>3 ?
        2      2 Redo: q(2) ?
?       2      2 Exit: q(4) ?
        4      2 Call: 4>3 ?
        4      2 Exit: 4>3 ?       % nincs ? ⇒ a doboz törlődik, (ld. a szaggatott piros téglalapot)
?       1      1 Exit: p(4) ?
X = 4 ? ;
```

Egy egyszerű nyomkövetési példa (folyt.)

```
q(2).
q(4).
q(7).

p(X) :- q(X), X > 3.
```

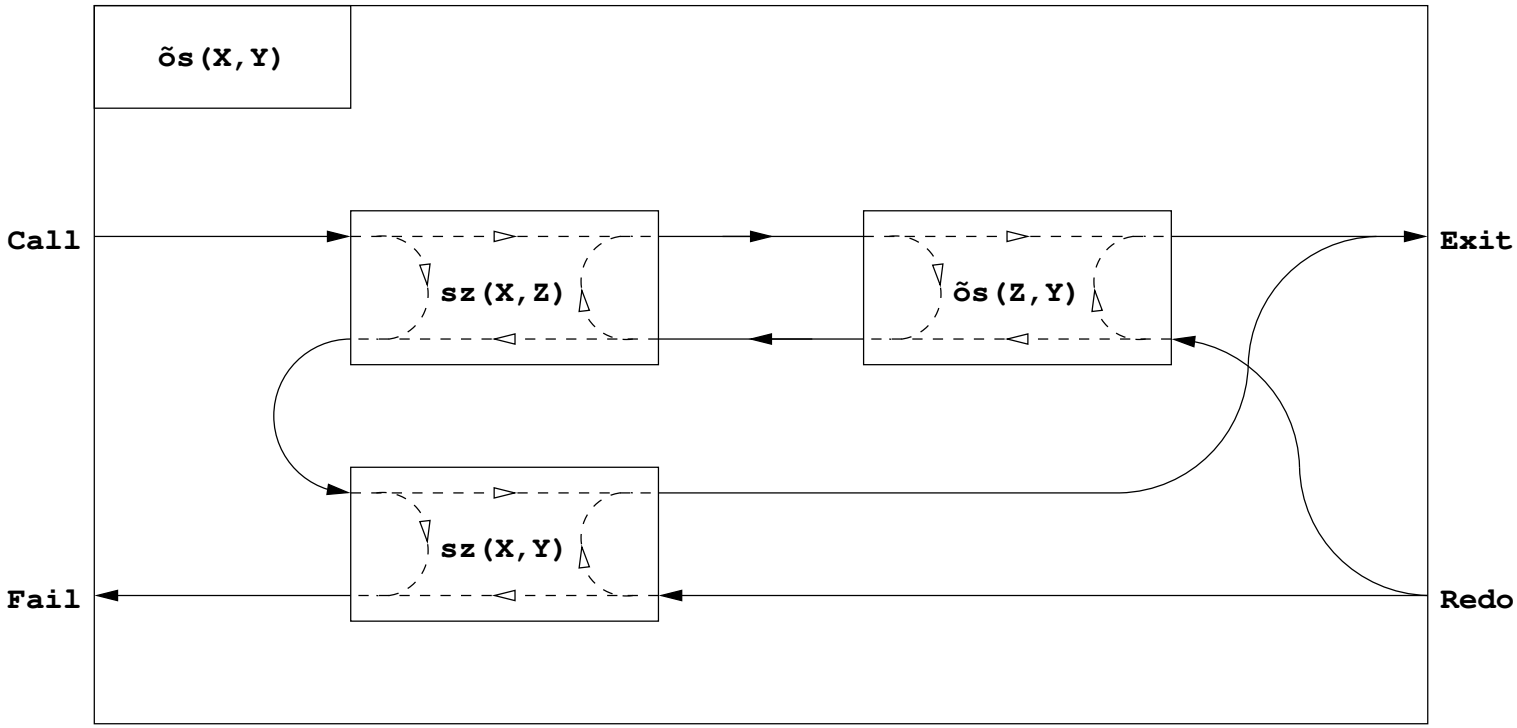



```
| ?- consult(pq0), trace, p(X).
(...)
        4      2 Exit: 4>3 ?        % nincs ? ⇒ a doboz törlődik (*)
?       1      1 Exit: p(4) ?
X = 4 ? ;
        1      1 Redo: p(4) ?
                                    % (*) miatt nem látjuk a Redo-Fail kapukat a 4>3 hívásra
        2      2 Redo: q(4) ?
        2      2 Exit: q(7) ?       % nincs ? ⇒ a doboz törlődik
        5      2 Call: 7>3 ?
        5      2 Exit: 7>3 ?        % nincs ? ⇒ a doboz törlődik
        1      1 Exit: p(7) ?       % nincs ? ⇒ a doboz törlődik
X = 7 ? ;
no
```

Eljárás-doboz: több klózból álló eljárás

```
ős(X,Y) :- sz(X,Z), ős(Z,Y). % X őse Y ha X szülője Z és Z őse Y (a szülő őse ős)
ős(X,Y) :- sz(X,Y).          % X őse Y ha X szülője Y           (a szülő ős)

sz(a,b).   sz(b,c).   sz(b,d).
```



Eljárás-doboz modell – „kapcsolási” alapelvek

- A feladat: „szülő” eljárásdoboz és a „belső” eljárások dobozainak összekapcsolása
- Előfeldolgozás: érjük el, hogy a klózfejekben csak változók legyenek, ehhez a fej-egyesítéseket alakítsuk hívásokká, pl.
  `fakt(0,1).` $\Rightarrow$ `fakt(X,Y) :- X=0, Y=1.`
- Előre menő végrehajtás (balról-jobbra menő nyilak):
  - A szülő Call kapuját az 1. klóz első hívásának Call kapujára kötjük.
  - Egy belső eljárás Exit kapuját
    - a következő hívás Call kapujára, vagy,
    - ha nincs következő hívás, akkor a szülő Exit kapujára kötjük
- Visszafelé menő végrehajtás (jobbról-balra menő nyilak):
  - Egy belső eljárás Fail kapuját
    - az előző hívás Redo kapujára, vagy, ha nincs előző hívás, akkor
    - a következő klóz első hívásának Call kapujára, vagy
    - ha nincs következő klóz, akkor a szülő Fail kapujára kötjük
  - A szülő Redo kapuját mindegyik klóz utolsó hívásának Redo kapujára kötjük
    - mindig abba a klózra térünk vissza, amelyben legutoljára voltunk

Nyomkövetés – legfontosabb parancsok (SICStus + SWI)

- Beépített eljárások
  - `trace`, `debug` – a `c`, `l` paranccsal indítja a nyomkövetést
  - `notrace`, `nodebug` – kikapcsolja a nyomkövetést
  - `spy(P)`, `nospy(P)`, `nospyall` – töréspont be/ki a `P` eljárásra, $\forall$ ki.
- Alapvető nyomkövetési parancsok (SICStus: `<RET>`-tel kell lezárni)
  - `h` (help) – parancsok listázása
  - `c` (creep) vagy csak `<RET>` – lassú futás (minden kapunál megáll)
  - `l` (leap) – csak töréspontnál áll meg
  - `+` ill. `-` – töréspont be/ki a kurrens eljárásra
  - `s` (skip) – eljárástörzs átlépése (Call/Redo $\Rightarrow$ Exit/Fail)
  - `w` (write) – teljes mélységű kiíratás
  - `o` (out) SICStus, `u` (up) SWI – kilépés az eljárástörzsből
  - `r` (retry) – újrakezdi a kurrens hívás végrehajtását
- Információ-megjelenítő és egyéb parancsok
  - `g` (goals) – a kurrens hívást tartalmazó célok kiíratása
  - `b` (break) – új, beágyazott Prolog interakciós szint létrehozása
  - `n` (notrace) – nyomkövető kikapcsolása
  - `a` (abort) – a kurrens futás abbahagyása

Eljárás-doboz modell – OO szemléletben (kieg. anyag)

- Minden eljáráshoz tartozik egy osztály, amelynek van egy konstruktor függvénye (amely megkapja a hívási paramétereket) és egy `next` „adj egy (következő) megoldást” metódusa.
- Az osztály nyilvántartja, hogy hányadik klózban jár a vezérlés
- A metódus első meghívásakor az első klóz első Hívás kapujára adja a vezérlést
- Amikor egy részeljárás Hívás kapujához érkezünk, létrehozunk egy példányt a meghívandó eljárásból, majd
- meghívjuk az eljáráspéldány „következő megoldás” metódusát (*)
  - Ha ez sikerül, akkor a vezérlés átkerül a következő hívás Hívás kapujára, vagy a szülő Kilépési kapujára
  - Ha ez meghiúsul, megszüntetjük az eljáráspéldányt majd ugrunk az előző hívás Újra kapujára, vagy a következő klóz elejére, stb.
- Amikor egy Újra kapuhoz érkezünk, a (*) lépésnél folytatjuk.
- A szülő Újra kapuja (a „következő megoldás” nem első hívása) a tárolt klózsorszámnak megfelelő klózban az utolsó Újra kapura adja a vezérlést.

OO szemléletű dobozok: p/2 C++ kódrészlet (kieg. anyag)

Az ős/2 Prolog eljárásnak (298. dia) megfelelő C++ objektum „köv. megoldás” metódusa:

```
boolean os::next(           { // Return next solution for os/2
  switch(clno)           {
  case 0:                     // first call of the method
    clno = 1;                 // enter clause 1:                os(X,Y) :- sz(X,Z), os(Z,Y).
    szaptr = new sz(x, &z);   // create a new instance of subgoal sz(X,Z)
  redo11:
    if(!szaptr->next()) {     // if sz(X,Z) fails
      delete szaptr;          // destroy it,
      goto cl2;        }      // and continue with clause 2 of os/2
    pptr = new os(z, py);     // otherwise, create a new instance of subgoal os(Z,Y)
  case 1:                     // (enter here for Redo port if clno==1)
    /* redo12: */
    if(!pptr->next())  {      // if os(Z,Y) fails
      delete pptr;            // destroy it,
      goto redo11;     }      // and continue at redo port of sz(X,Z)
    return TRUE;              // otherwise, exit via the Exit port
  cl2:
    clno = 2;                 // enter clause 2:                os(X,Y) :- sz(X,Y).
    szbptr = new sz(x, py);   // create a new instance of subgoal sz(X,Y)
  case 2:                     // (enter here for Redo port if clno==2)
    /* redo21: */
    if(!szbptr->next()) {     // if sz(X,Y) fails
      delete szbptr;          // destroy it,
      return FALSE;    }      // and exit via the Fail port
    return TRUE;              // otherwise, exit via the Exit port
                        } }
```

Tartalom

Diszjunkció

- Ismétlés: klóztörzsben a vessző (‘,’) jelentése „és”, azaz konjunkció
- A ‘;’ operátor jelentése „vagy”, azaz diszjunkció

```
% fakt(+N, ?F): F = N!.
fakt(N, F) :- N = 0, F = 1.
fakt(N, F) :-
    N > 0, N1 is N-1,
    fakt(N1, F1), F is F1*N.
```

```
fakt(N, F) :-
    (   N = 0, F = 1

    ;   N > 0, N1 is N-1,
        fakt(N1, F1), F is F1*N
    ).
```

- A diszjunkciót nyitó zárójel elérésekor választási pont jön létre
  - először a diszjunkciót az első ágára redukáljuk
  - visszalépés esetén a diszjunkciót a második ágára redukáljuk
- Tehát az első ág sikeres lefutása után kilépünk a diszjunkcióból, és az utána jövő célokkal folytatjuk a redukálást
  - azaz a ‘;’ elérésekor a ‘)’-nél folytatjuk a futást
- A ‘;’ skatulyázható (jobbról-balra) és gyengébben köt mint a ‘,’
- Konvenció: a diszjunkciót *mindig* zárójelbe tesszük,
  a skatulyázott diszjunkciót és az ágakat feleslegesen nem zárójelezzük.
  Pl. (a felesleges zárójelek aláhúzva, kiemelve): `(p;(q;r)), (a;(b,c);d)`

A diszjunkció mint szintaktikus édesítőszer

- A diszjunkció egy segéd-predikátummal mindig kiküszöbölhető, pl.:

```
a(X, Y, Z) :-
        p(X, U), q(Y, V),
        (   r(U, T), s(T, Z)
        ;   t(V, Z)
        ;   t(U, Z)
        ),
        u(X, Z).
```

- Kigyűjtjük azokat a változókat, amelyek a diszjunkcióban és azon kívül is előfordulnak(`U`, `V`, `Z`)
- A segéd-predikátumnak ezek a változók lesznek az argumentumai
- A segéd-predikátum minden klóza megfelel a diszjunkció egy ágának

```
seged(U, V, Z) :- r(U, T), s(T, Z).
seged(U, V, Z) :- t(V, Z).
seged(U, V, Z) :- t(U, Z).
```

```
a(X, Y, Z) :-
    p(X, U), q(Y, V),
    seged(U, V, Z),
    u(X, Z).
```

Diszjunkció – megjegyzések (kieg. anyag)

- Az egyes klózok ‘ÉS’ vagy ‘VAGY’ kapcsolatban vannak?
  - A program klózai ÉS kapcsolatban vannak, pl.

    ```
    szuloje('Imre', 'István').     szuloje('Imre', 'Gizella').        % (1)
    ```

    azt állítja: Imre szülője István ÉS Imre szülője Gizella.
  - Az *(1)* klózok alternatív (VAGY kapcsolatú) válaszokhoz vezetnek:

    ```
    :- szuloje('Imre' Ki). ⟹   Ki = 'István' ? ; Ki = 'Gizella' ? ; no
    ```

    „`Imre` szülője `Sz`” ha `( Sz = István` vagy `Sz = Gizella ).`
- Az *(1)* predikátum átalakítható egyetlen, diszjunkciós klózzá:

  ```
  szuloje('Imre', Sz) :-          (   Sz = 'István'
                                  ;   Sz = 'Gizella'
                                  ).                                 % (2)
  ```

  Vö. De Morgan azonosságok: $(A \leftarrow B) \wedge (A \leftarrow C) \equiv (A \leftarrow (B \vee C))$
- Általánosan: tetszőleges predikátum egyklózossá alakítható:
  - a klózokat azonos fejűvé alakítjuk, új változók és =-ek bevezetésével:

    ```
    szuloje('Imre', Sz) :- Sz = 'István'.
    szuloje('Imre', Sz) :- Sz = 'Gizella'.
    ```
  - a klóztörzseket egy diszjunkcióvá fogjuk össze, lásd *(2)*.

A meghiúsulásos negáció (NF – Negation by Failure)

- A `\+ Hívás` vezérlési szerkezet (vö. $\nvdash$ – nem bizonyítható) procedurális szemantikája
  - végrehajtja a `Hívás` hívást,
  - ha `Hívás` sikeresen lefutott, akkor meghiúsul,
  - egyébként (azaz ha `Hívás` meghiúsult) sikerül.
- `\+ Hívás` futása során `Hívás` legfeljebb egyszer sikerül
- `\+ Hívás` sohasem helyettesít be változót
- Példa: Keressünk (adatbázisunkban) olyan gyermeket, aki nem férfi!

```
| ?- sz(X, _Sz), \+ ffi(X). % negált cél ≡ ¬ffi(X)
⟹   X = 'Gizella' ? ; no
```

- Mi történik ha a két hívást megcseréljük?

```
| ?- \+ ffi(X), sz(X, _Sz). % negált cél ≡ ¬(∃X.ffi(X))
⟹   no
```

- $\backslash + H$ logikai megfelelője: $\neg\exists\vec{X}(H)$, ahol $\vec{X}$ a H-ban a hívás pillanatában behelyettesítetlen változókat jelöli. Emiatt $\backslash +$ **érzékeny a sorrendre!!!**

```
| ?- X = 2, \+ X = 1.        ⟹   X = 2 ?
| ?- \+ X = 1, X = 2.        ⟹   no
```

Gondok a meghiúsulásos negációval

- A negált cél jelentése függ attól, hogy mely változók bírnak értékkel
- Mikor nincs gond?
  - Ha a negált cél tömör (nincs benne behelyettesítetlen változó)
  - Ha nyilvánvaló, hogy mely változók behelyettesítetlenek (pl. mert „semmis” változók: _), és a többi változó tömör értékkel bír.

```
% nem_szulo(+Sz): adott Sz nem szulo
nem_szulo(Sz) :- \+ szuloje(_, Sz).
```

- A `\+` művelet a „Zárt Világ” feltételezésen alapul (Closed World Assumption – CWA): ami nem bizonyítható, az nem igaz.

```
| ?- \+ szuloje('Imre', X).        ⟹   no
| ?- \+ szuloje('Géza', X).        ⟹   true ?          (*)
```

  - A klasszikus matematikai logika következményfogalma monoton: ha a premisszák halmaza bővül, a következmények halmaza nem szűkülhet.
  - A CWA alapú logika nem monoton, példa: bővítsük a programot egy `szuloje('Géza', xxx).` alakú állítással $\Rightarrow$`(*)` meghiúsul.

Példa: együttható meghatározása lineáris kifejezésben

- Formula: számokból és az ‘x’ atomból ‘+’ és ‘*’ operátorokkal épül fel.
- Lineáris formula: a ‘*’ operátor (legalább) egyik oldalán szám áll.

```
% egyhat(Kif, E): A Kif lineáris formulában az x együtthatója E.
egyhat(x, 1).
egyhat(Kif, E) :-
        number(Kif), E = 0.
egyhat(K1+K2, E) :-
        egyhat(K1, E1),
        egyhat(K2, E2),
        E is E1+E2.
egyhat(K1*K2, E) :-          % (4)
        number(K1),
        egyhat(K2, E0),
        E is K1*E0.
egyhat(K1*K2, E) :-          % (5)
        number(K2),
        egyhat(K1, E0),
        E is K2*E0.
```

- A fenti megoldás hibás – többszörös megoldást kaphatunk:

```
| ?- egyhat(((x+1)*3)+x+2*(x+x+3), E).   ⟹   E = 8 ?; no
| ?- egyhat(2*3+x, E).                   ⟹   E = 1 ?; E = 1 ?; no
```

- A többszörös megoldás oka:
  az `egyhat(2*3, E)` hívás esetén a (4) és (5) klóz egyaránt sikeres!

Többszörös megoldások kiküszöbölése

- El kell érnünk, hogy ha a (4) sikeres, akkor (5) már ne sikerüljön
- A többszörös megoldás kiküszöbölhető:
  - Negációval – írjuk be (4) előfeltételének negáltját (5) törzsébe:

```
(...)
egyhat(K1*K2, E) :-                                    % (4)
        number(K1), egyhat(K2, E0), E is K1*E0.
egyhat(K1*K2, E) :-                                    % (5)
        \+ number(K1),
        number(K2), egyhat(K1, E0), E is K2*E0.
```

  - hatékonyabban, feltételes kifejezéssel:

```
(...)
egyhat(K1*K2, E) :-
        (   number(K1) -> egyhat(K2, E0), E is K1*E0
        ;   number(K2), egyhat(K1, E0), E is K2*E0
        ).
```

- A feltételes kifejezés hatékonyabban fut, mert:
  - nem kell kétszer futtatni a `number(K1)` feltételt
  - nem hagy választási pontot

Feltételes kifejezés Prologban

- Szintaxis (`felt`, `akkor`, `egyébként` tetszőleges célsorozatok):

```
(...) :-
        ...,
        (   felt -> akkor
        ;   egyébként
        ),
        ....
```

- Deklaratív szemantika: a fenti alak jelentése megegyezik az alábbival, ha a `felt` egy egyszerű feltétel (azaz nem oldható meg többféleképpen):

```
(...) :-
        ...,
        (   felt, akkor
        ;   \+ felt, egyébként
        ),
        ....
```

Feltételes kifejezések (folyt.)

- Procedurális szemantika

  A `(felt->akkor;egyébként),folytatás` célsorozat végrehajtása:

  - Végrehajtjuk a `felt` hívást.
  - Ha `felt` sikeres, akkor az `(akkor,folytatás)` célsorozatra redukáljuk a fenti célsorozatot, a `felt` *első* megoldása által adott behelyettesítésekkel. A `felt` cél többi megoldását nem keressük meg!
  - Ha `felt` sikertelen, akkor az `(egyébként,folytatás)` célsorozatra redukáljuk, behelyettesítés nélkül.

- Többszörös elágaztatás skatulyázott feltételes kifejezésekkel:

```
(   felt1 -> akkor1
;   felt2 -> akkor2
;   ...
)
```

```
(   felt1 -> akkor1
;  (felt2 -> akkor2
;   ...
))
```

  A kiemelt zárójelek feleslegesek!

- Az `egyébként` rész elhagyható, alapértelmezése: `fail`.
- `\+ felt` átírható feltételes kifejezéssé: `( felt -> fail ; true )`

Feltételes kifejezés – példák

- Faktoriális

```
% fakt(+N, ?F): N! = F.
fakt(N, F) :-
        (   N = 0 -> F = 1
        %   N = 0,   F = 1
        ;   N > 0,   N1 is N-1, fakt(N1, F1), F is N*F1
        ).
```

- Jelentése itt azonos a diszjunkciós alakkal (-> helyett , – lásd komment)
- A diszjunkciós alak választási pontot hagy, a feltételes szerkezet nem.
- Szám előjele

```
% Sign = sign(Num)
sign(Num, Sign) :-
        (   Num > 0 -> Sign = 1     %   if Num > 0 then Sign = 1
        ;   Num < 0 -> Sign = -1    % elif Num < 0 then Sign = -1
        ;              Sign = 0     % else              Sign = 0
        ).
```

A vágó eljárás – a feltételes szerkezet megvalósítási alapja

- A vágó beépített eljárás (`!`) mindig sikerül; de mellékhatásként
  1. letiltja az adott predikátum további klózainak választását,

```
első_poz_elem([X|_], X) :- X > 0, !.      % "zöld vágó"
első_poz_elem([X|L], EP) :- X =< 0, első_poz_elem(L, EP).
```

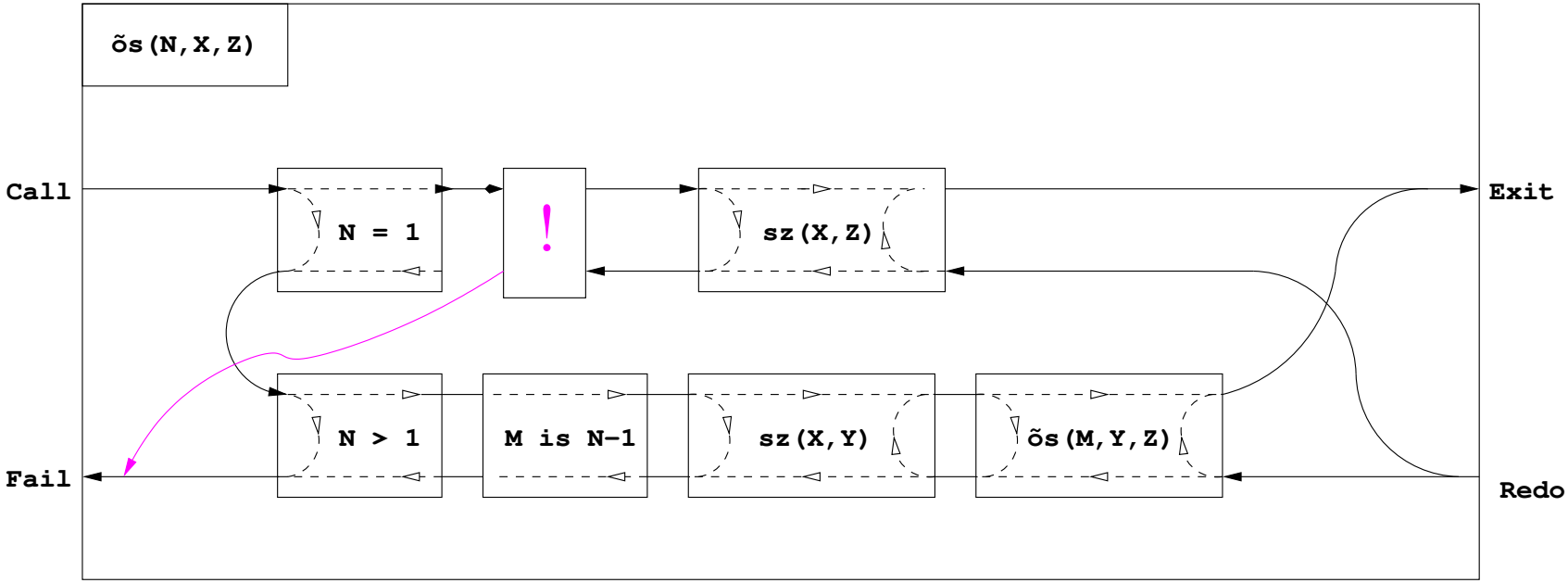
  2. megszünteti a választási pontokat az előtte levő eljáráshívásokban.

```
első_poz_elem(L, EP) :-  member(X, L), X>0, !, EP = X. % "vörös vágó"
```

- A zöld vágó nem változtatja meg az eredmény(eke)t, de gyorsítja a futást
- A vörös vágó megváltoztatja az eredményhalmazt
- Segédfogalom: egy cél szülőjének az őt tartalmazó klóz fejével illesztett hívást nevezzük
  - A 4-kapus modellben a szülő a körülvevő dobozhoz rendelt cél.
  - A fenti vágók szülője lehet pl. az `első_poz_elem([-1,0,3,0,2], P)` cél
- Átfogalmazás: a vágó a keresési térben vág le ágakat:
  - a vágó meghívásától visszafelé egészen a szülő célig – azt is beleértve – megszünteti a választási pontokat.

A vágó megvalósítása a 4-kapus doboz modellben

```
% ős(+N, ?X, ?Z): X-nek N-edik generációs őse Z (N>0 adott egész szám)
ős(1, X, Z) :-  !, sz(X, Z).                          % sz(X, Z): X-nek szülője Z
ős(N, X, Z) :-  N > 1, M is N-1, sz(X, Y), ős(M, Y, Z).
```



- A vágó `Fail` kapujából a körülvevő (szülő) doboz `Fail` kapujára megyünk.
- Ugyanilyen doboz keletkezik feltételes szerkezet használatakor:

```
ős2(N, X, Z) :- (   N = 1 -> sz(X, Z)
                ;   N > 1, M is N-1, sz(X, Y), ős2(M, Y, Z).
                ).
```

Tartalom

Magasabbrendű eljárások – listakezelés

- Magasabbrendű (vagy meta-eljárás) egy eljárás,
  - ha eljárásként értelmezi egy vagy több argumentumát
  - pl. `findall/3`, `call/1`: `call(P)` a `P` kifejezést hívásként végrehajtja.
- Listafeldolgozás `findall` segítségével – példák
  - Páros elemek kiválasztása (vö. Elixir filter)

```
% Az L egész-lista páros elemeinek listája Pk.
páros_elemei(L, Pk) :-
    findall(X, (member(X, L), X mod 2 =:= 0), Pk).
```

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

  - A listaelemek négyzetre emelése (vö. Elixir map)

```
% Az L számlista  elemei négyzeteinek listája Nk.
négyzetei0(L, Nk) :-
    findall(Y, (member(X, L), négyzete(X, Y)), Nk).
négyzete(X, Y) :- Y is X*X.
```

```
| ?- négyzetei0([1,2,3,4], Nk).    ⟹   Nk = [1,4,9,16]
```

- A `findall` futása során a megoldásokat le kell másolja – ez nagyobb adatstruktúrák esetén komoly hátrány lehet

Részlegesen paraméterezett eljáráshívások – segédeszközök

- A `négyzetei/2`-nek megfelelő feladatot Elixirben a `map/2` magasabbrendű függvénnyel oldhatjuk meg.
- Prologban ennek a `maplist/3` eljárás felel meg, amely a `library(lists)`-ben található:

```
:- use_module(library(lists)).          %
négyzetei(Xs, Ys) :-
        % Xs minden minden X elemére hívjuk meg a négyzete(X,Y)
        % eljárást, és a kapott Y értékeket gyűjtsük az Ys listába
        maplist(négyzete, Xs, Ys).
```

- A `maplist/3` az esetek nagy többségében visszavezethető a `findall/3`-ra:

```
maplist0(Fun, Xs, Ys) :-
    findall(Y, (member(X, Xs), call(Fun, X, Y)), Ys).
```

- A `négyzete` argumentum a `négyzete/2` részlegesen paraméterezett hívásának tekinthető: `call(négyzete, X, Y)` ≡ `négyzete(X, Y)`
- Általánosan: `call(RPred, A1, A2, ...)` végrehajtása: az `RPred` részleges hívást kiegészíti az `A1, A2, ...` argumentumokkal, és meghívja.
- A `call/N` predikátumok SICStus 4-ben beépített eljárásként állnak rendelkezésre

Részlegesen paraméterezett eljárások – rekurzív `maplist/3`

- Részleges paraméterezéssel a `maplist/3` meta-eljárás rekurzívan is definiálható:

```
% maplist(Pred, Xs, Ys): Az Xs lista elemeire a Pred transzformációt
% alkalmazva kapjuk az Ys listát.
maplist(Pred, [X|Xs], [Y|Ys]) :-
        call(Pred, X, Y), maplist(Pred, Xs, Ys).
maplist(_, [], []).

másodfokú_képe(P, Q, X, Y) :- Y is X*X + P*X + Q.
```

- Példák:

```
| ?- maplist(négyzete, [1,2,3,4], L).          ⟹  L = [1,4,9,16]
| ?- maplist(másodfokú_képe(2,1), [1,2,3,4], L).⟹  L = [4,9,16,25]
```

- A `call/N`-re épülő megoldás előnyei:
  - általánosabb és hatékonyabb lehet, mint a `findall`-ra épülő;
  - alkalmazható akkor is, ha az elemekre elvégzendő műveletek nem függetlenek, pl. `foldl`.

Rekurzív meta-eljárások – `foldl` és `foldr`

- ```
  % foldl(+Xs, :Pred, +Y0, -Y): Y0-ból indulva, az Xs elemeire
  % balról jobbra sorra alkalmazva a Pred által leírt
  % kétargumentumú függvényt kapjuk Y-t.
  % SICStus library(lists)-ben scanlist/4 néven érhető el.
  foldl([X|Xs], Pred, Y0, Y) :-
          call(Pred, X, Y0, Y1), foldl(Xs, Pred, Y1, Y).
  foldl([], _, Y, Y).

  jegyhozzá(Alap, Jegy, Szam0, Szam) :- Szam is Szam0*Alap+Jegy.

  | ?- foldl([1,2,3], jegyhozzá(10), 0, E).  ⟹  E = 123
  ```
- ```
  % foldr(+Xs, :Pred, +Y0, -Y): Y0-ból indulva, az Xs elemeire jobbról
  % balra sorra alkalmazva a Pred kétargumentumú függvényt kapjuk Y-t.
  foldr([X|Xs], Pred, Y0, Y) :-
          foldr(Xs, Pred, Y0, Y1), call(Pred, X, Y1, Y).
  foldr([], _, Y, Y).

  | ?- foldr([1,2,3], jegyhozzá(10), 0, E).  ⟹  E = 321
  ```
- A `foldr` eljárás nem jobbrekurzív, ezért ritkábban használjuk

Tartalom

Keresési feladat Prologban – felsorolás vagy gyűjtés?

- Keresési feladat: adott feltételeknek megfelelő dolgok meghatározása.
- Prolog nyelven egy ilyen feladat alapvetően kétféle módon oldható meg:
  - gyűjtés – az összes megoldás összegyűjtése, pl. egy listába;
  - felsorolás – a megoldások visszalépéses felsorolása: egyszerre egy megoldást kapunk, de visszalépéssel sorra előáll minden megoldás.
- Egyszerű példa: egy egészlista páros elemeinek megkeresése:

% **Gyűjtés:**

```
% páros_elemei(L, Pk): Pk az L
% lista páros elemeinek listája.
páros_elemei([], []).
páros_elemei([X|L], Pk) :-
    (   X mod 2 =:= 0 ->
        Pk = [X|Pk1],
        páros_elemei(L, Pk1)
    ;   páros_elemei(L, Pk)
    ).
```

% **Felsorolás:**

```
% páros_eleme(L, P): P egy páros
% eleme az L listának.

páros_eleme([X|L], P) :-
    (   X mod 2 =:= 0, P = X
    % X akár páros, akár páratlan
    % folytatjuk a felsorolást:
    ;   páros_eleme(L, P)
    ).
% egyszerűbb, deklaratív megoldás:
páros_eleme2(L, P) :-
    member(P, L), P mod 2 =:= 0.
```

Gyűjtés és felsorolás kapcsolata

- Ha adott `páros_elemei`, hogyan definiálható `páros_eleme`?
  - A `member/2` beépített eljárás segítségével, pl.

```
páros_eleme(L, P) :-
    páros_elemei(L, Pk), member(P, Pk).
```

  - Természetesen ez így nem hatékony!
- Ha adott `páros_eleme`, hogyan definiálható `páros_elemei`?
  - Megoldásgyűjtő beépített eljárás segítségével, pl.

```
páros_elemei(L, Pk) :-
    findall(P, páros_eleme(L, P), Pk).
    % páros_eleme(L, P) összes P megoldásának listája Pk.
```

  - a `findall/3` beépített eljárás – és társai – az Elixir listajelölőhöz (komprehenzióhoz) hasonlóak, pl.:

```
% my_numlist(+A, +B, ?L): L = [A,...,B], A és B egészek.
my_numlist(A, B, L) :-
    B >= A-1,
    findall(X, between(A, B, X), L).
```

    vö. $L = \{X | A \leq X \leq B, integer(X)\}$, ahol $B \geq A - 1$

A `findall(?Gyűjtő, :Cél, ?Lista)` beépített eljárás

- Ezt az eljárást már korábban bemutattuk, most részletesen ismertetjük
- Az eljárás végrehajtása (procedurális szemantikája):
  - a `Cél` kifejezést eljáráshívásként értelmezi, meghívja (A `:Cél` annotáció meta- (azaz eljárás) argumentumot jelez);
  - minden egyes megoldásához előállítja `Gyűjtő` egy *másolatát*, azaz a változókat, ha vannak, szisztematikusan újakkal helyettesíti;
  - Az összes `Gyűjtő` másolat listáját egyesíti `Lista`-val.
- Példák az eljárás használatára:

```
| ?- findall(X, (member(X, [1,7,8,3,2,4]), X>3), L).
                ⟹    L = [7,8,4] ? ; no
| ?- findall(Y, member(X-Y, [a-c,a-b,b-c,c-e,b-d]), L).
                ⟹   L = [c,b,c,e,d] ? ; no
```

- Az eljárás jelentése (deklaratív szemantikája):
  `Lista` = { `Gyűjtő` másolat | (∃ `X` ... `Z`)`Cél` igaz }
  ahol `X`, ..., `Z` a `findall` hívásban levő *szabad változók*.

  Szabad változó (definíció): olyan, a hívás pillanatában behelyettesítetlen változó, amely a `Cél`-ban előfordul de a `Gyűjtő`-ben nem.

A `bagof(?Gyűjtő, :Cél, ?Lista)` beépített eljárás

- Példa az eljárás használatára:

```
gráf([a-c,a-b,b-c,c-e,b-d]).
| ?- gráf(_G), findall(B, member(A-B, _G), VegP).      % ld. előző dia
                              ⟹   VegP = [c,b,c,e,d] ? ; no
| ?- gráf(_G), bagof(B, member(A-B, _G), VegPk).
                              ⟹   A = a, VegPk = [c,b] ? ;
                              ⟹   A = b, VegPk = [c,d] ? ;
                              ⟹   A = c, VegPk = [e] ? ; no
```

- Az eljárás végrehajtása (procedurális szemantikája):
  - a `Cél` kifejezést eljáráshívásként értelmezi, meghívja;
  - összegyűjti a megoldásait (a `Gyűjtő`-t és a szabad változók behelyettesítéseit);
  - a szabad változók összes behelyettesítését *felsorolja* és mindegyik esetén a `Lista`-ban megadja az összes hozzá tartozó `Gyűjtő` értéket.
- A `bagof` eljárás jelentése (deklaratív szemantikája):
  `Lista` $=\{$ `Gyűjtő` $\mid$ `Cél` igaz $\}$, `Lista` $\neq$ `[]`, a szabad változók minden lehetséges behelyettesítésére.

A `bagof` megoldásgyűjtő eljárás – folyt. (kieg. anyag)

- Explicit egzisztenciális kvantorok
  - `bagof(Gyűjtő, V1 ^...^Vn ^Cél, Lista)` alakú hívása a `V1, ..., Vn` változókat egzisztenciálisan kvantáltnak tekinti, így ezeket nem sorolja fel.
  - jelentése: `Lista` $=$ { `Gyűjtő` | $(\exists$ `V1, ..., Vn`$)$`Cél` igaz } $\neq$ `[]`.

```
| ?- gráf(_G), bagof(B, A^member(A-B, _G), VegP).
                                  ⟹    VegP = [c,b,c,e,d] ? ; no
```

- Egymásba ágyazott gyűjtések
  - szabad változók esetén a `bagof` nemdeterminisztikus lehet, így érdemes lehet skatulyázni:

```
% A G irányított gráf fokszámlistája FL:
% FL = { A-N | N = |{ V | A-V ∈ G }|, N > 0 }
fokszámai(G, FL) :-
    bagof(A-N, Vk^(bagof(V, member(A-V, G), Vk),
                   length(Vk, N)                ), FL).

| ?- gráf(_G), fokszámai(_G, FL).
                ⟹    FL = [a-2,b-2,c-1] ? ; no
```

A `bagof` megoldásgyűjtő eljárás – folyt. (kieg. anyag)

- Fokszámlista kicsit hatékonyabb előállítása
  - Az előző példában a meta-argumentumban célsorozat szerepelt, ez mindenképpen interpretáltan fut – nevezzük el segédeljárásként
  - A segédeljárás bevezetésével a kvantor is szükségtelenné válik:

```
% pont_foka(?A, +G, ?N): Az A pont foka a G irányított gráfban N>0.
pont_foka(A, G, N) :-
        bagof(V, member(A-V, G), Vk), length(Vk, N).

% A G irányított gráf fokszámlistája FL:
fokszámai(G, FL) :-    bagof(A-N, pont_foka(A, G, N), FL).
```

- Példák a `bagof/3` és `findall/3` közötti kisebb különbségekre:

```
| ?- findall(X, (between(1, 5, X), X<0), L). ⟹   L = [] ? ; no
| ?-   bagof(X, (between(1, 5, X), X<0), L). ⟹   no
| ?- findall(S, member(S, [f(X,X),g(X,Y)]), L).
                              ⟹   L = [f(_A,_A),g(_B,_C)] ? ; no
| ?-   bagof(S, member(S, [f(X,X),g(X,Y)]), L).
                              ⟹   L = [f(X,X),g(X,Y)] ? ; no
```

- A `bagof/3` logikailag tisztább mint a `findall/3`, de költségesebb!

A `setof(?Gyűjtő, :Cél, ?Lista)` beépített eljárás

- Az eljárás végrehajtása:
  - ugyanaz mint: `bagof(Gyűjtő, Cél, L0), sort(L0, Lista)`,
  - `sort(L, RL)` egy univerzális rendező eljárás, amely az `L` listát rendezi az az azonos elemek kiszűrésével (a `@<` általános rendezés szerint, lásd később), és az eredményt `RL`-ban adja vissza.
- Példa a `setof/3` eljárás használatára:

```
gráf([a-c,a-b,b-c,c-e,b-d]).

% Gráf egy pontja P.
pontja(P, Gráf) :- member(A-B, Gráf), ( P = A ; P = B).

% A G gráf pontjainak listája Pk.
gráf_pontjai(G, Pk) :- setof(P, pontja(P, G), Pk).

| ?- gráf(_G), gráf_pontjai(_G, Pk).
⟹    Pk = [a,b,c,d,e] ? ; no
| ?- gráf(_G), bagof(P, pontja(P, _G), Pk).
⟹    Pk = [a,c,a,b,b,c,c,e,b,d] ? ; no
```

Tartalom

Operátoros kifejezések

- Példa: `S is -S1+S2` ekvivalens az `is(S, +(-(S1),S2))` kifejezéssel
- Szintaxis:

  ⟨ összetett kif. ⟩ ::=

⟨ struktúranév ⟩ (⟨ argumentum ⟩, . . .)	{eddig csak ez volt}
| ⟨ argumentum ⟩ ⟨ operátornév ⟩ ⟨ argumentum ⟩	{infix kifejezés}
| ⟨ operátornév ⟩ ⟨ argumentum ⟩	{prefix kifejezés}
| ⟨ argumentum ⟩ ⟨ operátornév ⟩	{posztfix kifejezés}
| (⟨ kifejezés ⟩)	{zárójeles kif.}

  ⟨ operátornév ⟩ ::= ⟨ struktúranév ⟩ {ha operátorként lett definiálva}
- Operátor(ok) definiálása

  `op(Prio, Fajta, OpNév)` vagy `op(Prio, Fajta, [OpNév`$_1$`,...OpNév`$_n$`])`, ahol
  - `Prio` (prioritás): 1–1200 közötti egész
  - `Fajta`: az `yfx`, `xfy`, `xfx`, `fy`, `fx`, `yf`, `xf` névkonstansok egyike
  - `OpNév`$_i$ (az operátor neve): tetszőleges névkonstans
- Az `op/3` beépített predikátum meghívását általában a programot tartalmazó fájl elején, *direktívában* helyezzük el:

  ```
  :- op(800, xfx, [szuloje,nagyszuloje]).  'Imre' szuloje 'István'.
  ```
- A direktívák a programfájl *betöltésekor* azonnal végrehajtódnak.

Operátorok jellemzői

- Egy operátort jellemez a fajtája és prioritása
- A fajta az asszociatívitás irányát és az irásmódot határozza meg:

Fajta			Írásmód	Értelmezés
bal-assz.	jobb-assz.	nem-assz.		
`yfx`	`xfy`	`xfx`	infix	`A f B` ≡ `f(A, B)`
	`fy`	`fx`	prefix	`f A` ≡ `f(A)`
`yf`		`xf`	posztfix	`A f` ≡ `f(A)`

- A zárójelezést a prioritás és az asszociatívitás együtt határozza meg, pl.
  - `a/b+c*d` ≡ `(a/b)+(c*d)` mert `/` és `*` prioritása 400 < 500 (`+` prioritása) (kisebb prioritás = erősebb kötés)
  - `a-b-c` ≡ `(a-b)-c` mert a `-` operátor fajtája `yfx`, azaz bal-asszociatív – balra köt, balról jobbra zárójelez (a fajtanévben az `y` betű mutatja az asszociatívitás irányát)
  - `a^b^c` ≡ `a^(b^c)` mert a `^` operátor fajtája `xfy`, azaz jobb-asszociatív (jobbra köt, jobbról balra zárójelez)
  - `a=b=c` szintaktikusan hibás, mert az `=` operátor fajtája `xfx`, azaz nem-asszociatív

Szabványos, beépített operátorok

Szabványos operátorok

Színkód: már ismert, új aritmetikai

1200	xfx	`:- -->`	
1200	fx	`:- ?-`	
1100	xfy	`;`	diszjunkció
1050	xfy	`->`	if-then
1000	xfy	`','`	
900	fy	`\+`	negáció
700	xfx	`= \=`	
		`< =< > >= =:= =\= is`	
		`@< @=< @> @>= == \== =..`	
500	yfx	`+ - \/ /\`	bitműveletek
400	yfx	`* / // rem`	
		`mod`	modulus
		`<< >>`	léptetések
200	xfx	`**`	hatványozás
200	xfy	`^`	
200	fy	`- \`	bitenkénti negáció

További beépített operátorok SICStus Prologban

1150	fx	`mode public`
		`dynamic block`
		`volatile`
		`discontiguous`
		`initialization`
		`multifile`
		`meta_predicate`
1100	xfy	`do`
900	fy	`spy nospy`
550	xfy	`:`
500	yfx	`\`
200	fy	`+`

Operátorok zárójelezése (kieg. anyag)

- Egy `X` $\texttt{op}_1$ `Y` $\texttt{op}_2$ `Z` zárójelezése, ahol $\texttt{op}_1$ és $\texttt{op}_2$ prioritása n_1 és n_2:
  - ha $n_1 > n_2$ akkor `X` $\texttt{op}_1$ `(Y` $\texttt{op}_2$ `Z)`;
  - ha $n_1 < n_2$ akkor `(X` $\texttt{op}_1$ `Y)` $\texttt{op}_2$ `Z`; (kisebb prio. $\Rightarrow$ erősebb kötés)
  - ha $n_1 = n_2$ és $\texttt{op}_1$ jobb-asszociatív (`xfy`), akkor `X` $\texttt{op}_1$ `(Y` $\texttt{op}_2$ `Z)`;
  - egyébként, ha $n_1 = n_2$ és $\texttt{op}_2$ bal-assz. (`yfx`), akkor `(X` $\texttt{op}_1$ `Y)` $\texttt{op}_2$ `Z`;
  - egyébként szintaktikus hiba
- Érdekes példa: `:- op(500, xfy, +^).  % :- op(500, yfx, +).`

  `| ?- :- write((1 +^ 2) + 3), nl.` $\Rightarrow$ `(1+^2)+3`

  `| ?- :- write(1 +^ (2 + 3)), nl.` $\Rightarrow$ `1+^2+3`

  tehát: konfliktus esetén az első operátor asszociativitása „győz”.
- Alapszabály: egy n prioritású operátor zárójelezetlen operandusaként
  - legfeljebb $n-1$ prioritású operátort fogadunk el az `x` oldalon
  - legfeljebb n prioritású operátort fogadunk el az `y` oldalon
- A zárójelezett kifejezéseket és az alapstruktúra-alakú kifejezéseket feltétel nélkül elfogadjuk operandusként
- Az alapszabály a prefix és posztfix operátorokra is alkalmazandó

Operátorok – további megjegyzések

- Ugyanaz a névkonstans használható többféle fajtájú operátorként is, pl. a ‘-’ és ‘+’ atomok prefix és infix beépített operátorként is definiálva vannak a Prolog ISO szabványában
- A „vessző” jel három szintaktikus helyzetben is használható:
  - összetett kifejezés (struktúra) argumentumait határoló jel pl. `szuloje('István', 'Gizella')`
  - listaelemeket határoló jel, pl. `[1,2,3|T]`
  - 1000 prioritású `xfy` op. pl.: `(p:-a,b,c)`$\equiv$`:-(p,','(a,','(b,c)))`
- A vessző atomként csak a `','`, határolóként csak a `,`, operátorként mindkét formában – `','` vagy `,` – használható.
- `:-(p, a,b,c)` többértelmű: $\stackrel{?}{=}$ `:-(p, (a,b,c))`, ... $\stackrel{?}{=}$ `:-(p,a,b,c)`...
- Egyértelműsítés: argumentumban vagy listaelemben az 1000-nél $\geq$ prioritású operátort tartalmazó kifejezést *zárójelezni kell*:

```
| ?- write_canonical((a,b,c)).  ⟹  ','(a,','(b,c))
| ?- write_canonical(a,b,c).    ⟹  ! write_canonical/3 does not exist
```

Operátorok törlése, lekérdezése (kieg. anyag)

- Egy vagy több operátor törlésére az `op/3` beépített eljárást használhatjuk, ha első argumentumként (prioritásként) 0-t adunk meg.

```
| ?- X = a+b, op(0, yfx, +).  ⟹   X = +(a,b) ? ; no
| ?- X = a+b.                 ⟹   ! Syntax error
                                  ! op. expected after expression
                                  ! X = a <<here>> + b .
| ?- op(500, yfx, +).         ⟹   yes
| ?- X = +(a,b).              ⟹   X = a+b ? ; no
```

- Az adott pillanatban érvényes operátorok lekérdezése:

```
current_op(Prioritás, Fajta, OpNév)
```

```
| ?- current_op(P, F, +).
⟹    F = fy, P = 200 ? ;
      F = yfx, P = 500 ? ;
no
| ?- current_op(_, xfy, Op), write_canonical(Op), write(' '), fail.
; do -> ',' : ^
no
```

Operátorok felhasználása

- Mire jók az operátorok?
  - aritmetikai eljárások kényelmes irására, pl. `X is (Y+3) mod 4`
  - szimbolikus kifejezések kezelésére (pl. szimbolikus deriválás)
  - klózok leírására (`:-` és `','` is operátor), és meta-eljárásoknak való átadására, pl `asserta( (p(X):-q(X),r(X)) )`
  - eljárásfejek, eljáráshívások olvashatóbbá tételére:

    ```
    :- op(800, xfx, [nagyszülője, szülője]).

    Gy nagyszülője N :-  Gy szülője Sz, Sz szülője N.
    ```

  - adatstruktúrák olvashatóbbá tételére, pl.

    ```
    sav(kén, h*2-s-o*4).
    ```

Operátoros példa: polinom behelyettesítési értéke

- Polinom: az ‘x’ atomból és számokból a ‘+’ és ‘*’ op.-okkal felépülő kif.
- A feladat: egy polinom értékének kiszámolása egy adott x érték esetén.

```
% value_of0(P, X, V): A P polinom x=X helyen vett értéke V.
value_of0(x, X, V) :-
    V = X.
value_of0(N, _, V) :-
    number(N), V = N.
value_of0(P1+P2, X, V) :-
    value_of0(P1, X, V1),
    value_of0(P2, X, V2),
    V is V1+V2.
value_of0(P1*P2, X, V) :-
    value_of0(P1, X, V1),
    value_of0(P2, X, V2),
    V is V1*V2.
```

```
| ?- value_of0((x+1)*x+x+2*(x+x+3), 2, V).
V = 22 ? ; no
```

Klasszikus szimbolikuskifejezés-feldolgozás: deriválás

- Írjunk olyan Prolog predikátumot, amely az `x` névkonstansból és számokból a `+`, `-`, `*` műveletekkel képzett kifejezések deriválását elvégzi!

```
% deriv(Kif, D):  Kif-nek az x szerinti deriváltja D.
deriv(x, D) :-                      D = 1.
deriv(C, D) :-                      number(C), D = 0.
deriv(U+V, DU+DV) :-                deriv(U, DU), deriv(V, DV).
deriv(U-V, DU-DV) :-                deriv(U, DU), deriv(V, DV).
deriv(U*V, DU*V + U*DV) :-          deriv(U, DU), deriv(V, DV).
```

```
| ?- deriv(x*x+x, D).
  ⟹     D = 1*x+x*1+1 ? ; no
```

```
| ?- deriv((x+1)*(x+1), D).
  ⟹     D = (1+0)*(x+1)+(x+1)*(1+0) ? ; no
```

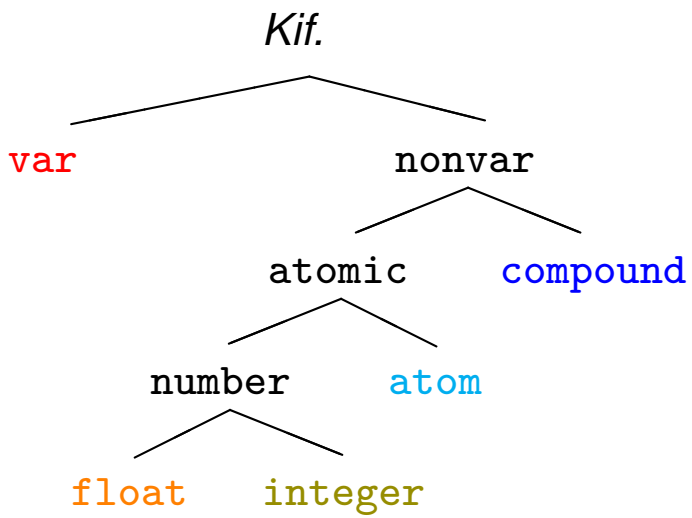
```
| ?- deriv(I, 1*x+x*1+1).
  ⟹     I = x*x+x ? ; no
```

```
| ?- deriv(I, 0).
  ⟹     no
```

Tartalom

Kifejezések osztályozása

- Kifejezésfajták – osztályozó beépített eljárások (ismétlés)



Szabványos eljárások:

`var(X)`	`X` változó
`nonvar(X)`	`X` nem változó
`atomic(X)`	`X` konstans
`compound(X)`	`X` struktúra
`number(X)`	`X` szám
`atom(X)`	`X` atom
`float(X)`	`X` lebegőpontos szám
`integer(X)`	`X` egész szám

- További osztályozó eljárások:
  - `simple(X)`: `X` nem összetett (konstans vagy változó);
  - `callable(X)`: `X` atom vagy struktúra (nem szám és nem változó);
  - `ground(X)`: `X` tömör, azaz nem tartalmaz behelyettesítetlen változót.

Osztályozó eljárások: a `length/2` példája (kieg. anyag)

- Példa: a `length/2` beépített eljárás egy lehetséges megvalósítása

```
% length(?L, ?N): Az L lista N hosszú.
length(L, N) :-    var(N), length(L, 0, N).
length(L, N) :- nonvar(N), dlength(L, 0, N).
```

```
% length(?L, +I0, -I):
%   Az L lista I-I0 hosszú.
length([], I, I).
length([_|L], I0, I) :-
        I1 is I0+1,
        length(L, I1, I).
```

```
% dlength(?L, +I0, +I):
%   Az L lista I-I0 hosszú.
dlength([], I, I).
dlength([_|L], I0, I) :-
        I0<I,I1 is I0+1,
        dlength(L, I1, I).
```

```
| ?- length([1,2], Len).   (length/3)   ⇒  Len = 2 ? ; no
| ?- length([1,2], 3).     (dlength/3)   ⇒  no
| ?- length(L, 3).         (dlength/3)   ⇒  L = [_A,_B,_C] ?;no
| ?- length(L, Len).       (length/3)    ⇒  L = [], Len = 0 ? ;
                                             L = [_A], Len = 1 ? ;
                                             L = [_A,_B], Len = 2 ? ; ...
```

Kifejezések szétszedése és összerakása – motiváló példa

- `Polinom ::= x` | szám | `Polinom + Polinom` | `Polinom * Polinom`
- Egy `P` polinom kiértékelése adott `x` behelyettesítés mellett (ismétlés):

```
% value_of(+P, +XV, ?V): az x = XV helyettesítéssel P értéke V.
```

```
value_of0(x, X, V) :- V = X.
value_of0(N, _, V) :-
    number(N), V = N.

value_of0(P1+P2, X, V) :-
    value_of0(P1, X, V1),
    value_of0(P2, X, V2),
    V is V1+V2.
value_of0(Polinom, X, V) :-
    Polinom = *(P1,P2),
    value_of0(P1, X, V1),
    value_of0(P2, X, V2),
    PolinomV = *(V1,V2),
    V is PolinomV.
```

```
value_of(x, X, V) :- V = X.
value_of(N, _, V) :-
    number(N), V = N.

value_of(Polinom, X, V) :-
    Polinom =.. [Func,P1,P2],
    value_of(P1, X, V1),
    value_of(P2, X, V2),
    PolinomV =.. [Func,V1,V2],
    V is PolinomV.
```

- `value_of/3` minden az `is/2` által elfogadott bináris függvényre működik!

```
| ?- value_of(exp(100,min(x,1/x)), 2, V).   ⟹   V = 10.0 ? ; no
```

Az *univ* beépített eljárás

- Kiindulás: `| ?- K=F(A,B).` $\Rightarrow$ szintaxis-hiba, helyette: `K=..[F,A,B]`, pl.:

```
| ?- el(a,b,10) =.. L.        ⟹   L = [el,a,b,10]
| ?- Kif =.. [el,a,b,10].     ⟹   Kif = el(a,b,10)
| ?- alma =.. L.              ⟹   L = [alma]
```

- Az *univ* eljárás hívási mintái: `+Kif =..  ?Lista`
  `-Kif =..  +Lista` (`Lista` zárt végű!)
- Az eljárás jelentése:
  - `Kif` = $Fun(A_1, \ldots, A_n)$ és `Lista` = $[Fun, A_1, \ldots, A_n]$, ahol Fun egy névkonstans és $A_1, \ldots, A_n$ tetszőleges kifejezések; vagy
  - `Kif` = C és `Lista` = $[C]$, ahol C egy (szám- vagy név)konstans.
- További példák:

```
| ?- Kif =.. [1234].          ⟹   Kif = 1234
| ?- Kif =.. L.              ⟹   hiba
| ?- f(a,g(10,20)) =.. L.    ⟹   L = [f,a,g(10,20)]
| ?- Kif =.. [/,X,2+X].      ⟹   Kif = X/(2+X)
| ?- [a,b,c] =.. L.          ⟹   L = ['.',a,[b,c]]

          (SWI Prologban:)  ⟹   L=['[|]',a,[b,c]]
```

Indexelés (áttekintés)

- Mi az indexelés?
  - egy hívásra alkalmazható (illeszthető fejű) klózok gyors kiválasztása,
  - egy eljárás klózainak fordítási idejű csoportosításával.
- A legtöbb Prolog rendszer, így a SICStus Prolog is, az első fej-argumentum alapján indexel (first argument indexing).
- Az indexelés alapja az első fejargumentum külső funktora:
  - `C` szám vagy névkonstans esetén `C/0`;
  - `R` nevű és `N` argumentumú struktúra esetén `R/N`;
  - változó esetén nem értelmezett (minden funktorhoz besoroljuk).
- Az indexelés megvalósítása:
  - Fordításkor minden funktor $\Rightarrow$az alkalmazható klózok listája
  - Futáskor konstans idő alatt elő tudjuk vennie a megfelelő klózlistát
  - *Fontos:* ha egyelemű a lista, nem hozunk létre választási pontot!
- Például `szuloje('István', X)` kételemű klózlistára szűkít, de `szuloje(X, 'István')` mind a 6 klózt megtartja (mert a SICStus Prolog csak az első argumentum szerint indexel)

Struktúrák kezelése: a `functor/3` eljárás (kieg. anyag)

- `functor/3`: kifejezés funktorának, adott funktorú kifejezésnek az előállítása
  - Hívási minták: `functor(-Kif, +Név, +Argszám)`
    `functor(+Kif, ?Név, ?Argszám)`
  - Jelentése: `Kif` egy `Név`/`Argszám` funktorú kifejezés.
    - A konstansok 0-argumentumú kifejezésnek számítanak.
    - Ha `Kif` kimenő, az adott funktorú legáltalánosabb kifejezéssel egyesíti (argumentumaiban csupa különböző változóval).
- Példák:

```
| ?- functor(el(a,b,1), F, N).     ⟹   F = el, N = 3
| ?- functor(E, el, 3).            ⟹   E = el(_A,_B,_C)
| ?- functor(alma, F, N).          ⟹   F = alma, N = 0
| ?- functor(Kif, 122, 0).         ⟹   Kif = 122
| ?- functor(Kif, el, N).          ⟹   hiba
| ?- functor(Kif, 122, 1).         ⟹   hiba
| ?- functor([1,2,3], F, N).      ⟹   F = '.', N = 2
| ?- functor(Kif, ., 2).           ⟹   Kif = [_A|_B]
```

Struktúrák kezelése: az `arg/3` eljárás (kieg. anyag)

- `arg/3`: kifejezés adott sorszámú argumentuma.
  - Hívási minta: `arg(+Sorszám, +StrKif, ?Arg)`
  - Jelentése: A `StrKif` struktúra `Sorszám`-adik argumentuma `Arg`.
  - Végrehajtása: `Arg`-ot az adott sorszámú argumentummal **egyesíti**.
  - Az `arg/3` eljárás így nem csak egy argumentum elővételére, hanem a struktúra változó-argumentumának behelyettesítésére is használható (ld. a 2. példát alább).
- Példák:

```
| ?- arg(3, el(a, b, 23), Arg).    ⟹    Arg = 23
| ?- K=el(_,_,_), arg(1, K, a),
     arg(2, K, b), arg(3, K, 23).  ⟹    K = el(a,b,23)
| ?- arg(1, [1,2,3], A).           ⟹    A = 1
| ?- arg(2, [1,2,3], B).           ⟹    B = [2,3]
```

- Az *univ* visszavezethető a `functor` és `arg` eljárásokra (és viszont), például:

```
Kif =.. [F,A1,A2]   ⟺    functor(Kif, F, 2),
                         arg(1, Kif, A1), arg(2, Kif, A2)
```

Alkalmazás: részkifejezések keresése (kieg. anyag)

- A feladat: adott egy tetszőleges kifejezés, soroljuk fel a benne levő számokat, és minden szám esetén adjuk meg az ún. *kiválasztóját*!
- Egy részkifejezés kiválasztója egy olyan lista, amely megadja, hogy sorra mely argumentumpozíciók mentén juthatunk el hozzá.
- Az `[`$i_1, i_2, \ldots, i_k$`]` $k \geq 0$ lista egy `Kif`-ből az i_1-edik argumentum i_2-edik argumentumának, ... i_k-adik argumentumát választja ki.
  (Az `[]` kiválasztó `Kif`-ből `Kif`-et választja ki.)
- Pl. `a*b+f(5,8,7)/c`-ben `b` kiválasztója `[1,2]`, `7` kiválasztója `[2,1,3]`.

```
% kif_szám(?Kif, ?N, ?Kiv): Kif Kiv kiválasztójú része az N szám.
kif_szám(X, X, []) :- number(X).
kif_szám(X, N, [I|Kiv]) :-  compound(X),
        X =.. [_F|Args], nth1(I, Args, X1),                    % (*)
        kif_szám(X1, N, Kiv).
```

```
| ?- kif_szám(f(5,[8,b]), Sz, K).⟹   Sz = 5, K = [1]  ? ;
                                     Sz = 8, K = [2,1] ? ; no
```

- A `(*)` sor helyett ez is állhat:

```
        functor(X, _F, ArgNo), between(1, ArgNo, I), arg(I, X, X1),
```

Atomok szétszedése és összerakása

- `atom_codes/2`: névkonstans és karakterkód-lista közötti átalakítás
  - Hívási minták: `atom_codes(+Atom, ?KódLista)`
    `atom_codes(-Atom, +KódLista)`
  - Jelentése: `Atom` karakterkódjainak a listája `KódLista`.
- Példák:

```
| ?- atom_codes(ab, Cs).          ⟹  Cs = [97,98]
| ?- atom_codes(ab, [0'a|L]).     ⟹  L = [98]
| ?- Cs="bc", atom_codes(Atom, Cs). ⟹  Cs = [98,99], Atom = bc
| ?- atom_codes(Atom, [0'a|L]).   ⟹  hiba
```

- Az `atom_codes(Atom, KódLista)` beépített eljárás végrehajtása:
  - Ha `Atom` adott (bemenő), és a $c_1 c_2 \ldots c_n$ karakterekből áll, akkor `KódLista`-t egyesíti a $[k_1, k_2, \ldots, k_n]$ listával, ahol k_i a c_i karakter kódja.
  - Ha `KódLista` egy adott karakterkód-lista, akkor ezekből a karakterekből összerak egy névkonstanst, és azt egyesíti `Atom`-mal.

Atomok kezelése: példák (kieg. anyag)

- Keresés névkonstansokban

```
% Atom-ban a Rész nem üres részatom kétszer ismétlődik.
dadogó_rész(Atom, Rész) :-
    atom_codes(Atom, Cs),
    Ds = [_|_],
    append([_,Ds,Ds,_], Cs), % append/2, lásd library(lists)
    atom_codes(Rész, Ds).
```

```
| ?- dadogó_rész(babaruhaha, R).    ⟹    R = ba ? ; R = ha ? ; no
```

- Atomok összefűzése

```
% atom_concat(+A, +B, ?C): A és B névkonstansok összefűzése C.
% (Szabványos beépített eljárás atom_concat(?A, ?B, +C) módban is.)
atom_concat(A, B, C) :-
    atom_codes(A, Ak), atom_codes(B, Bk),
    append(Ak, Bk, Ck),
    atom_codes(C, Ck).
```

```
| ?- atom_concat(abra, kadabra, A). ⟹    A = abrakadabra ?
```

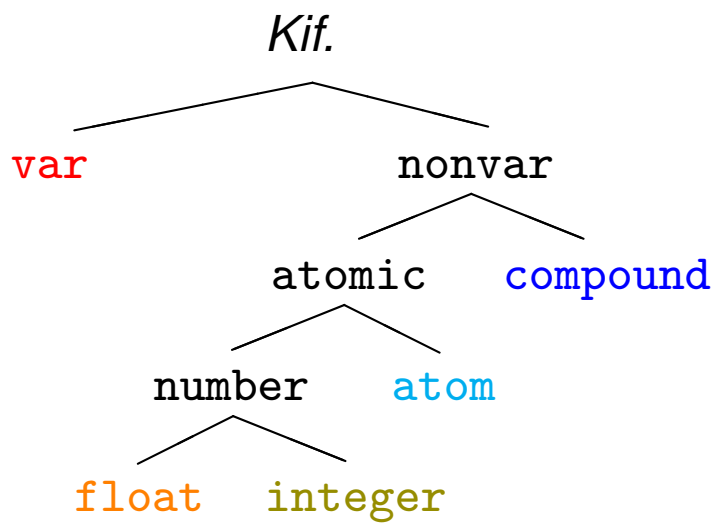
Számok szétszedése és összerakása

- `number_codes/2`: szám és karakterkód-lista közötti átalakítás
  - Hívási minták: `number_codes(+Szám, ?KódLista)`
    `number_codes(-Szám, +KódLista)`
  - Jelentése: Igaz, ha `Szám` tizes számrendszerbeli alakja a `KódLista` karakterkód-listának felel meg.
- Példák:

```
| ?- number_codes(12, Cs).          ⟹  Cs = [49,50]
| ?- number_codes(0123, [0'1|L]).   ⟹  L = [50,51]
| ?- number_codes(N, " - 12.0e1").  ⟹  N = -120.0
| ?- number_codes(N, "12e1").       ⟹  hiba (nincs .0)
| ?- number_codes(120.0, "12e1").   ⟹  no (mert a szám adott! :-)
```

- A `number_codes(Szám, KódLista)` beépített eljárás végrehajtása:
  - Ha `Szám` adott (bemenő), és a $c_1 c_2 \ldots c_n$ karakterekből áll, akkor `KódLista`-t egyesíti a $[k_1, k_2, \ldots, k_n]$ kifejezéssel, ahol k_i a c_i karakter kódja.
  - Ha `KódLista` egy adott karakterkód-lista, akkor ezekből a karakterekből összerak egy számot (ha nem lehet, hibát jelez), és azt egyesíti `Szám`-mal.

Prolog kifejezések általános rendezése: a $\prec$ reláció



A különböző kif.-fajták sorrendje:

$$\texttt{var} \prec \texttt{float} \prec \texttt{integer} \prec \texttt{atom} \prec \texttt{compound}$$

Egy kifejezésfajtán belüli sorrendezés szabályai:

- Változók: rendszerfüggő (pl. memóriacím alapján)
- Egész és lebegőpontos számok: szokásosan ($x \prec y \Leftrightarrow x < y$)
- Atomok: lexikografikus sorrend ($\texttt{abc} \prec \texttt{abcd}$, $\texttt{abcv} \prec \texttt{abcz}$)
- Összetett kif.-ek: $\texttt{név}_a(a_1, \ldots, a_n) \prec \texttt{név}_b(b_1, \ldots, b_m) \Leftrightarrow$
  1. $n < m$, pl. $\texttt{p(x,s(u,v,w))} \prec \texttt{a(b,c,d)}$, vagy
  2. $n = m$, és $\texttt{név}_a \prec \texttt{név}_b$ (lexikografikusan), pl. $\texttt{a(x,y)} \prec \texttt{p(b,c)}$, vagy
  3. $n = m$, $\texttt{név}_a = \texttt{név}_b$, és az első olyan i-re melyre $a_i \neq b_i$, $a_i \prec b_i$, pl. $\texttt{r(1,u+v,3,x)} \prec \texttt{r(1,u+v,5,a)}$

Kifejezések összehasonlítása – beépített eljárások

- Beépített eljárások tetszőleges kifejezések összehasonlítására:

hívás	igaz, ha
`Kif1 @< Kif2`	`Kif1` $\prec$ `Kif2`
`Kif1 @=< Kif2`	`Kif2` $\not\prec$ `Kif1`
`Kif1 @> Kif2`	`Kif2` $\prec$ `Kif1`
`Kif1 @>= Kif2`	`Kif1` $\not\prec$ `Kif2`
`Kif1 == Kif2`	`Kif1` $\not\prec$ `Kif2` $\wedge$ `Kif2` $\not\prec$ `Kif1`
`Kif1 \== Kif2`	`Kif1` $\prec$ `Kif2` $\vee$ `Kif2` $\prec$ `Kif1`

- Az összehasonlítás mindig a belső (kanonikus) alak szerint történik:

  `| ?- [1, 2, 3, 4] @< struktúra(1, 2, 3).` $\Longrightarrow$ **yes**

- Beépített elj. tetszőleges lista rendezésére: `sort(+L, ?S)`
  Jelentése: az `L` lista `@<` szerinti rendezése `S`,
  `==/2` szerint azonos elemek ismétlődését kiszűrve.

```
        | ?- sort([a,c,a,b,b,c,c,b,d,a(2,3),c(1),2.0,1,X], S).
        S = [X,2.0,1,a,b,c,d,c(1),a(2,3)] ? ; no
(SWI):S = [X,1,2.0,a,b,c,d,c(1),a(2,3)]. :-(
```

Összefoglalás: a Prolog egyenlőség-szerű beépített eljárásai

- `U = V`: `U` egyesítendő `V`-vel.
  Soha sem jelez hibát.
- `U == V`: `U` azonos `V`-vel.
  Soha sem jelez hibát és soha sem helyettesít be.
- `U =:= V`: Az `U` és `V` aritmetikai kifejezések értéke megegyezik.
  Hibát jelez, ha `U` vagy `V` nem (tömör) aritmetikai kifejezés.
- `U is V`: `U` egyesítendő a `V` aritmetikai kifejezés értékével.
  Hiba, ha `V` nem (tömör) aritmetikai kifejezés.
- (`U =.. V`: `U` „szétszedettje” a `V` lista)

```
| ?- X = 1+2.       ⟹   X = 1+2
| ?- 3 = 1+2.       ⟹   no
| ?- X == 1+2.      ⟹   no
| ?- 3 == 1+2.      ⟹   no
| ?- +(1,2)==1+2    ⟹   yes
| ?- X =:= 1+2.     ⟹   hiba
| ?- 1+2 =:= X.     ⟹   hiba
| ?- 2+1 =:= 1+2.   ⟹   yes
| ?- 2.0 =:= 1+1.   ⟹   yes
| ?- 2.0 is 1+1.    ⟹   no
| ?- X is 1+2.      ⟹   X = 3
| ?- 1+2 is X.      ⟹   hiba
| ?- 3 is 1+2.      ⟹   yes
| ?- 1+2 is 1+2.    ⟹   no
| ?- 1+2 =.. X.     ⟹   X = [+,1,2]
| ?- X =.. [f,1].   ⟹   X = f(1)
```

Összefoglalás: a Prolog nem-egyenlő jellegű beépített eljárásai

A nem-egyenlőség jellegű eljárások soha sem helyettesítenek be változót!

- `U \= V`: `U` nem egyesíthető `V`-vel.
  Soha sem jelez hibát.

```
| ?- X \= 1+2.        ⟹   no
| ?- +(1,2) \= 1+2.   ⟹   no
```

- `U \== V`: `U` nem azonos `V`-vel.
  Soha sem jelez hibát.

```
| ?- X \== 1+2.       ⟹   yes
| ?- 3 \== 1+2.       ⟹   yes
| ?- +(1,2)\==1+2     ⟹   no
```

- `U =\= V`: Az `U` és `V` aritmetikai kifejezések értéke különbözik.
  Hibát jelez, ha `U` vagy `V` nem (tömör) aritmetikai kifejezés.

```
| ?- X =\= 1+2.       ⟹   hiba
| ?- 1+2 =\= X.       ⟹   hiba
| ?- 2+1 =\= 1+2.     ⟹   no
| ?- 2.0 =\= 1+1.     ⟹   no
```

A Prolog (nem-)egyenlőség jellegű beépített eljárásai – példák

		Egyesítés		*Azonosság*		*Aritmetika*		
`U`	`V`	`U = V`	`U \= V`	`U == V`	`U \== V`	`U =:= V`	`U =\= V`	`U is V`
`1`	`2`	*no*	`yes`	*no*	`yes`	*no*	`yes`	*no*
`a`	`b`	*no*	`yes`	*no*	`yes`	error	error	error
`1+2`	`+(1,2)`	`yes`	*no*	`yes`	*no*	`yes`	*no*	*no*
`1+2`	`2+1`	*no*	`yes`	*no*	`yes`	`yes`	*no*	*no*
`1+2`	`3`	*no*	`yes`	*no*	`yes`	`yes`	*no*	*no*
`3`	`1+2`	*no*	`yes`	*no*	`yes`	`yes`	*no*	`yes`
`X`	`1+2`	`X=1+2`	*no*	*no*	`yes`	error	error	`X=3`
`X`	`Y`	`X=Y`	*no*	*no*	`yes`	error	error	error
`X`	`X`	`yes`	*no*	`yes`	*no*	error	error	error

Jelmagyarázat: `yes` – siker; *no* – meghiúsulás, error – hiba.

Az egyesítés kiegészítése: előfordulás-ellenőrzés, *occurs check*

- Kérdés: `X` és `s(X)` egyesíthető-e?
  - A matematikai válasz: *nem*, egy változó nem egyesíthető egy olyan struktúrával, amelyben előfordul (ez az előfordulás-ellenőrzés).
  - Az ellenőrzés költséges, ezért alaphelyzetben nem alkalmazzák (emiatt ún. ciklikus kifejezések keletkezhetnek)
  - Szabványos eljárásként rendelkezésre áll: `unify_with_occurs_check/2`
  - Kiterjesztés (pl. SICStus): az előfordulás-ellenőrzés elhagyása miatt keletkező ciklikus kifejezések tisztességes kezelése.
- Példák:

```
| ?- X = s(1,X).
        X = s(1,s(1,s(1,s(1,s(...))))) ?
| ?- unify_with_occurs_check(X, s(1,X)).
        no
| ?- X = s(X), Y = s(s(Y)), X = Y.
        X = s(s(s(s(s(...))))), Y = s(s(s(s(s(...))))) ?
```

XVIII. rész

Haladó Prolog

Tartalom

Nyelvi eszközök a keresési tér szűkítésére

- Az első Prolog rendszerektől kezdve: vágó, szabványos jelölése `!`
- Későbbi kiterjesztés: az `( if -> then ; else )` feltételes szerk.
- Feltételes szerkezet – procedurális szemantika (ismétlés)
  A `(felt->akkor;egyébként),folyt` célsorozat végrehajtása:
  - Végrehajtjuk a `felt` hívást (egy önálló végrehajtási környezetben).
  - Ha `felt` sikeres $\Longrightarrow$ „`akkor,folyt`” célsorozattal folytatjuk, a `felt` első megoldása által eredményezett behelyettesítésekkel.
    A `felt` cél többi megoldását nem keressük meg!
  - Ha `felt` meghiúsul $\Longrightarrow$ „`egyébként,folyt`” célsorozattal folytatjuk.
- A vágó beépített eljárás – procedurális szemantika (ismétlés)
  - mindig sikerül; de mellékhatásként megszünteti a választási pontokat egészen a szülő célig, azt is beleértve. (Egy *C* cél szülője az a cél, amelyet, a *C*-t tartalmazó klóz fejével illesztettünk.)
- A vágó „színe”:
  - zöld, ha csak olyan ágakat vág le, amelyek nem vezetnek megoldáshoz (nem módosítja a megoldások halmazát);
  - piros, ha megoldást tartalmazó ágat is levág (módosítja a megoldáshalmazt)

Példa: első_poz_elem(+L, ?P): P az L lista első pozitív eleme

- Rekurzív megoldás (mérnöki)

```
első_poz_elem1([X|L], EP) :- (   X > 0 -> EP = X
                             ;   első_poz_elem1(L, EP)
                             ).
```

- Nem-rekurzív, deklaratív, de lassú megoldás (matematikusi):

```
első_poz_elem2(L, EP) :-                % L első pozitív eleme EP ha
    append(Pref, [EP|_], L),            % Pref ++ [EP] ++ _ = L,
    EP>0, \+ ( member(X, Pref), X>0 ). % EP>0, Pref-nek nincs pozitív eleme
```

- Itt amikor `EP>0` sikerül, a felsorolás leállítható, hiszen egy hátrábbi `EP` nem lehet `L` első pozitív eleme:

```
első_poz_elem3(L, EP) :-
    append(Pref, [EP|_], L), EP>0, !, \+ ( member(X, Pref), X>0 ).
```

- Ha `EP` kimenő (`(+,-)` mód), akkor a vágó elérésekor `Pref`-nek nyilván nem lehet pozitív eleme. További egyszerűsítések:

```
első_poz_elem4(L, EP) :- append(_Pref, [EP|_], L), EP>0, !.
első_poz_elem5(L, EP) :- member(EP, L), EP>0, !.
```

- A 4.-5. változat `(+,+)` módban hibás: `első_poz_elem4([1,2], 2)` $\Longrightarrow$ `yes`
- A vágás **alapszabálya**: kimenő paraméter a vágó után kapjon értéket:

```
első_poz_elem6(L, EP) :- member(X, L), X>0, !, EP=X. % (+,+) módban is jó!
```

A 3.–6, megoldás épít az `append/2`, ill. `member/2` felsorolási sorrendjére!

A vágás alapszabálya a mérnöki megoldásban.

- Az első megoldásban a felt. szerkezetet írjuk át többklózos formára:

```
első_poz_elem7([X|_], X) :- X > 0.
első_poz_elem7([X|L], EP) :- X =< 0, első_poz_elem7(L, EP).
```

- Szúrjunk be egy (zöld) vágót!

```
első_poz_elem8([X|_], X) :- X > 0, !.
első_poz_elem8([X|L], EP) :- X =< 0, első_poz_elem8(L, EP).
```

- A második klózból hagyjuk el a „felesleges” negált vizsgálatot

```
első_poz_elem9([X|_], X) :- X > 0, !.
első_poz_elem9([X|L], EP) :- első_poz_elem9(L, EP).
```

- Ez a változat is hibás `(+,+)` módban: `első_poz_elem9([1,2], 2)` $\Longrightarrow$ `yes`
- A **vágás alapszabályának** betartásával kijavítható a hiba:

```
első_poz_elem10([X|_], EP) :- X > 0, !, EP = X.
első_poz_elem10([X|L], EP) :- első_poz_elem10(L, EP).
```

- Ez nemcsak általánosabban használható, hanem hatékonyabb kód is: csak akkor helyettesíti be a kimenő paramétert, ha tudja, hogy pozitív
- Az alapszabály betartásakor az ún. indexelés is hatékonyabb lesz

További példák a vágás alapszabályának betartására

```
% fakt(+N, ?F): N! = F.
fakt(0, F) :- !, F = 1.
fakt(N, F) :- N > 0, N1 is N-1, fakt(N1, F1), F is N*F1.

% last(+L, ?E): az L nem üres lista utolsó eleme E.
last([E], Last)    :- !, Last = E.
last([_|L], Last)  :- last(L, Last).

% páros_elemei(L, Pk): Pk az L egészlista páros elemeinek listája.
páros_elemei([], []).
páros_elemei([X|L], Pk) :-
    X mod 2 =:= 0, !, Pk = [X|Pk1], páros_elemei(L, Pk1).
páros_elemei([_X|L], Pk) :-
    % _X mod 2 =:= 1,
    páros_elemei(L, Pk).
```

- Kezdő Prolog programozóknak **a diszjunktív feltételes szerkezet használatát javasoljuk** a vágó helyett.

Példa: `max(X, Y, Z)`: `X` és `Y` maximuma `Z` (kieg. anyag)

- 1. változat, tiszta Prolog. Lassú (előre-behelyettesítés, két hasonlítás), választási pontot hagy.

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- Y > X.
```

- 2. változat, zöld vágóval. Lassú (előre-behelyettesítés, két hasonlítás), nem hagy választási pontot.

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y) :- Y > X.
```

- 3. változat, vörös vágóval. Gyorsabb (előre-behelyettesítés, egy hasonlítás), nem hagy választási pontot, de nem használható ellenőrzésre, pl. `| ?- max(10, 1, 1)` sikerül.

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

- 4. változat, vörös vágóval. Helyes, nagyon gyors (egy hasonlítás, nincs előre-behelyettesítés) és nem is hoz létre választási pontot.

```
max(X, Y, Z) :- X >= Y, !, Z = X.
max(X, Y, Y) /* :- Y > X */.
```

Tartalom

Determinizmus

- Egy hívás determinisztikus, ha (legfeljebb) egyféleképpen sikerülhet.
- Egy eljáráshívás egy sikeres végrehajtása determinisztikusan futott le, ha nem hagyott választási pontot a híváshoz tartozó részfában:
  - vagy választásmentesen futott le, azaz létre sem hozott választási pontot (figyelem: ez a Prolog megvalósítástól függ!);
  - vagy létrehozott ugyan választási pontot, de megszüntette (kimerítette, levágta).
- A SICStus Prolog nyomkövetésében ? jelzi a nemdeterminisztikus lefutást:

```
p(1, a).
p(2, b).
p(3, b).
```

```
| ?- p(1, X).                          %  det. hívás,
                1 1 Exit: p(1,a)      %    det. lefutás
| ?- p(Y, a).                          %  det. hívás,
              ? 1 1 Exit: p(1,a)      %    nemdet. lefutás
| ?- p(Y, b), Y > 2.                   %  nemdet. hívás
              ? 1 1 Exit: p(2,b)      %    nemdet. lefutás
                1 1 Exit: p(3,b)      %    det. lefutás
```

A determinisztikus lefutás és a választásmentesség

- Mi a determinisztikus lefutás haszna?
  - a futás gyorsabb lesz,
  - a tárigény csökken,
  - más optimalizálások (pl. jobbrekurzió) alkalmazhatók.
- Hogyan ismerheti fel a fordító a választásmentességet
  - egyszerű feltételes szerkezet (vö. Elixir őrfeltétel)
  - indexelés (indexing)
  - vágó és indexelés kölcsönhatása
- Az alábbi definíciók esetén SICStusban a `p(`*`Nonvar`*`, Y)` hívás választásmentes, azaz nem hoz létre választási pontot:

Egyszerű feltétel

```
p(X, Y) :-
    (   X =:= 1 -> Y = a
    ;   Y = b
    ).
```

Indexelés

```
p(1, a).
p(2, b).
```

Indexelés és vágó

```
p(1, Y) :- !,
    Y = a.
p(_, b).
```

Választásmentesség feltételes szerkezetek esetén

- Feltételes szerkezet végrehajtásakor általában választási pont jön létre.
- A SICStus Prolog a „`( felt -> akkor ; egyébként )`” szerkezetet választásmentesen hajtja végre, ha a `felt` konjunkció tagjai csak:
  - aritmetikai összehasonlító eljáráshívások (pl. `<`, `=<`, `=:=`), és/vagy
  - kifejezés-típust ellenőrző eljáráshívások (pl. `atom`, `number`), és/vagy
  - általános összehasonlító eljáráshívások (pl. `@<`, `@=<`,`==`).
- Választásmentes kód keletkezik a „`fej :- felt, !, akkor.`” klózból, ha `fej` argumentumai különböző változók, és `felt` olyan mint fent.
- Például választásmentes kód keletkezik az alábbi definíciókból:

```
vektorfajta(X, Y, Fajta) :-
  (   X =:= 0, Y =:= 0
      % X=0, Y=0 nem lenne jó
  ->  Fajta = null
  ;   Fajta = nem_null
  ).
```

```
vektorfajta(X, Y, Fajta) :-
  X =:= 0, Y =:= 0, !,
  Fajta = null.
vektorfajta(_X, _Y, nem_null).
```

Indexelés

- Mi az indexelés?
  - egy adott hívásra illeszthető klózok gyors kiválasztása,
  - egy eljárás klózainak fordítási idejű csoportosításával.
- A legtöbb Prolog rendszer, így a SICStus Prolog is, az első fej-argumentum alapján indexel (first argument indexing).
- Az indexelés alapja az első fejargumentum külső funktora:
  - `C` szám vagy névkonstans esetén `C/0`;
  - `R` nevű és `N` argumentumú struktúra esetén `R/N`;
  - változó esetén nem értelmezett.
- Az indexelés megvalósítása:
  - Fordítási időben: funktor $\Rightarrow$ illeszthető fejű klózok részhalmaza.
  - Futási időben: a részhalmaz lényegében konstans idejű kiválasztása (hash tábla használatával).
  - Fontos: ha egyelemű a részhalmaz, nincs választási pont!

Példa indexelésre

```
p(0, a).            /* (1) */
p(X, t) :- q(X).    /* (2) */
p(s(0), b).         /* (3) */
p(s(1), c).         /* (4) */
p(9, z).            /* (5) */
```

```
q(1).
q(2).
```

- A `p(A, B)` hívással illesztendő klózok:
  - ha `A` változó, akkor `(1) (2) (3) (4) (5)`
  - ha `A = 0`, akkor `(1) (2)`
  - ha `A` fő funktora `s/1`, akkor `(2) (3) (4)`
  - ha `A = 9`, akkor `(2) (5)`
  - minden más esetben `(2)`

- Példák hívásokra:
  - `p(1, Y)` nem hoz létre választási pontot.
  - `p(s(1), Y)` létrehoz választási pontot, de determinisztikusan fut le.
  - `p(s(0), Y)` nemdeterminisztikusan fut le.

Struktúrák, változók a fejargumentumban

- Ha a klózok szétválasztásához szükség van az első (struktúra) argumentum részeire is, akkor érdemes segédeljárást bevezetni.
- Pl. `p/2` és `q/2` ekvivalens, de `q(`*`Nonvar`*`, Y)` determinisztikus lefutású!

```
p(0, a).
p(s(0), b).
p(s(1), c).
p(9, z).
```

```
q(0, a).
q(s(X), Y) :-
    q_seged(X, Y).
q(9, z).
```

```
q_seged(0, b).
q_seged(1, c).
```

- Az indexelés figyelembe veszi a törzs elején szereplő egyenlőséget: `p(X, ...) :- X = Kif, ...` esetén `Kif` funktora szerint indexel.
- Példa: lista hosszának reciproka, üres lista esetén 0:

```
rhossz([], 0).
rhossz(L, RH) :- L = [_|_], length(L, H), RH is 1/H.
```

Indexelés – további tudnivalók

- Indexelés és aritmetika
  - Az indexelés nem foglalkozik aritmetikai vizsgálatokkal.
  - Pl. az `N = 0` és `N > 0` feltételek esetén a SICStus Prolog nem veszi figyelembe, hogy ezek kizárják egymást.
  - Az alábbi `fakt/2` eljárás lefutása nem-determinisztikus:

```
fakt(0, 1).
fakt(N, F) :- N > 0, N1 is N-1, fakt(N1, F1), F is N*F1.
```

- Indexelés és listák
  - Gyakran kell az üres és nem-üres lista esetét szétválasztani.
  - A bemenő lista-argumentumot célszerű az első argumentum-pozícióba tenni.
  - Az `[]` és `[...|...]` eseteket az indexelés megkülönbözteti (funktoruk: `'[]'/0` ill. `'.'/2`).
  - A két klóz sorrendje nem érdekes (feltéve, hogy zárt listával hívjuk az első pozíción) – de azért tegyük a leálló klózt mindig előre.

Listakezelő eljárások indexelése: példák

- Az `append/3` választásmentesen fut le, ha első argumentuma zárt végű.

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-  append(L1, L2, L3).
```

- A `last/2` közvetlen megfogalmazása nemdeterminisztikusan fut le:

```
% last(L, E): Az L lista utolsó eleme E.
last([E], E).
last([_|L], E) :- last(L, E).
```

- Érdemes segédeljárást bevezetni, `last2/2` választásmentesen fut

```
last2([X|L], E) :- last2(L, X, E).

% last2(L, X, E): Az [X|L] lista utolsó eleme E.
last2([], E, E).
last2([X|L], _, E) :- last2(L, X, E).
```

Az indexelés és a vágó kölcsönhatása

- Hogyan vehető figyelembe a vágó az indexelés fordításakor?
- Példa: a `p(1, A)` hívás választásmentes, de a `q(1, A)` nem!

```
p(1, Y) :- !, Y = 2.   % (1)
p(X, X).               % (2)
Arg1=1 → (1), Arg1≠1 → (2)
```

```
q(1, 2) :- !.          % (1)
q(X, X).               % (2)
Arg1=1 → {(1),(2)}, Arg1≠1 → (2)
```

- A fordító figyelembe veszi a vágót az indexelésben, ha garantált, hogy egy adott fő funktor esetén a vágót elérjük. Ennek feltételei:
  - 1. arg. változó, konstans, vagy csak változókat tartalmazó struktúra,
  - a további argumentumok változók,
  - a fejben az összes változóelőfordulás különböző,
  - a törzs első hívása a vágó (előtte megengedve egy fejillesztést kiváltó egyenlőséget).
- Ez egy újabb érv a vágás alapszabálya mellett:

A kimenő paraméterek értékadását mindig a vágó után végezzük!

A vágó és az indexelés hatékonysága (kieg. anyag)

- Fibonacci-szerű sorozat: $f_1 = 1;\ \ f_2 = 2;\ \ f_n = f_{\lfloor 3n/4 \rfloor} + f_{\lfloor 2n/3 \rfloor},\ \ n > 2$

```
% determ. xx=''
fib(1, 1).
fib(2, 2).
fib(N, F) :-
    N > 2, N2 is N*3//4, N3 is N*2//3,
    fibxx(N2, F2), fibxx(N3, F3),
    F is F2+F3.
```

```
% determ. lefut. xx='c'
fibc(1, 1) :- !.
fibc(2, 2) :- !.
fibc(N, F) :-
    N > 2, N2 is N*3//4, N3 is N*2//3,
    fibxx(N2, F2), fibxx(N3, F3),
    F is F2+F3.
```

```
% választásmentes, xx='ci'
fibci(1, F) :- !, F = 1.
fibci(2, F) :- !, F = 2.
fibci(N, F) :-
    N > 2, N2 is N*3//4, N3 is N*2//3,
    fibxx(N2, F2), fibxx(N3, F3),
    F is F2+F3.
```

- Futási idők $N = 6000$ esetén

	`fib`	`fibc`	`fibci`
futási idő	1.25 sec	1.22 sec	1.13 sec
meghiúsulási idő	0.29 sec	0.03 sec	0.00 sec
összesen	1.54 sec	1.25 sec	1.13 sec
nyom-verem mérete	37.4Mbyte	18.7 Mbyte	240 byte

- `fibc` esetén a meghiúsulási idő azért nem 0, mert a rendszer a nyom-vermet (trail-stack) dolgozza fel. (A nyom-verem tárolja a változó-értékadások visszacsinálási információit.)

Tartalom

Jobbrekurzió (farok-rekurzió, tail-recursion) optimalizálás

- Az általános rekurzió költséges, helyben és időben is.
- Jobbrekurzióról beszélünk, ha
  - a rekurzív hívás a klóztörzs utolsó helyén van, vagy az utolsó helyen szereplő diszjunkció egyik ágának utolsó helyén stb., és
  - a rekurzív hívás pillanatában nincs választási pont a predikátumban (a rekurzív hívást megelőző célok determinisztikusan futottak le, nem maradt nyitott diszjunkciós ág).
- Jobbrekurzió optimalizálás: az utolsó hívás végrehajtása előtt az eljárás által lefoglalt hely felszabadul ill. szemétgyűjtésre alkalmassá válik.
- Ez az optimalizálás nemcsak rekurzív hívás esetén, hanem minden **utolsó** hívás esetén megvalósul – a pontos név: utolsó hívás optimalizálás (last call optimisation).
- A jobbrekurzió így tehát nem növeli a memória-igényt, korlátlan mélységig futhat – mint a ciklusok az imperatív nyelvekben. Példa:

```
ciklus(Állapot) :- lépés(Állapot, Állapot1), !, ciklus(Állapot1).
ciklus(_Állapot).
```

Predikátumok jobbrekurzív alakra hozása – listaösszeg

- A listaösszegzés „természetes”, nem jobbrekurzív definíciója:

```
% sum0(+L, ?S): L elemeinek összege S (S = 0+L_n+L_{n-1}+...+L_1).
sum0([], 0).
sum0([X|L], S):-        sum0(L,S0), S is S0+X.
```

- Jobbrekurzív lista-összegző:

```
% sum(+L, ?S): L  elemeinek összege S (S = 0+L_1+L_2+...+L_n).
sum(L, S):-             sum(L, 0, S).
% sum(+L, +S0, ?S): L elemeit S0-hoz adva kapjuk S-t. (≡  Σ L = S-S0)
sum([], S, S).
sum([X|L], S0, S):-     S1 is S0+X, sum(L, S1, S).
```

- A jobbrekurzív `sum` eljárás több mint 3-szor gyorsabb mint a `sum0`!
- Az akkumulátor a mutábilis (angolul *mutable* – azaz megváltoztatható értékű) változó fogalmának deklaratív megfelelője:
  - A `sum/3`-ban az `S0` és `S` argumentumok akkumulátorpárt alkotnak.
  - Az akkumulátorpár két része az adott változó mennyiség (a példában az összeg) különböző időpontokban vett értékeit mutatja:
    - `S0` az összeg a `sum/3` meghívásakor: a mut. változó kezdőértéke;
    - `S` az összeg a `sum/3` lefutása után: a mut. változó végértéke.

Az akkumulátorok használata

- Az akkumulátorokkal általánosan több egymás utáni változtatást is leírhatunk:

```
p(..., A0, A):-
        q0(..., A0, A1), ...,
        q1(..., A1, A2), ...,
        qn(..., An, A).
```

- A `sum/3` második klóza ilyen alakra hozva:

```
sum([X|L], S0, S):- plus(X, S0, S1), sum(L, S1, S).

plus(X, S0, S) :- S is S0+X.
```

- Akkumulátorváltozók elnevezési konvenciója: kezdőérték: *Vált*`0`; közbülső értékek: *Vált*`1`, `...`, *Vált*`n`; végérték: *Vált*.
- A Prolog akkumulátorpár nem más mint a funkcionális programozásból ismert gyűjtőargumentum és a függvény eredményének együttese.

További akkumulátoros példák (kieg. anyag)

- Többszörös akkumulálás – lista összege és négyzetösszege

```
% sum2(+L, +S0, ?S, +Q0, ?Q): S-S0 =Σ Li, Q-Q0 = Σ Li^2
sum2([], S, S, Q, Q).
sum2([X|L], S0, S, Q0, Q):-
        S1 is S0+X, Q1 is Q0+X*X, sum2(L, S1, S, Q1, Q).
```

- Többszörös akkumulátorok összevonása egyetlen állapotváltozóvá

```
% sum3(+L, +S0/Q0, ?S/Q): S-S0 =Σ Li, Q-Q0 = Σ Li^2
sum3([], SQ, SQ).
sum3([X|L], SQ0, SQ) :-
        plus3(X, SQ0, SQ1), sum3(L, SQ1, SQ).
        % teljesen analóg a "sima" összegzővel

plus3(X, S0/Q0, S/Q) :- S is S0+X, Q is Q0+X*X.
```

Tartalom

Hogyan írjunk át imperatív nyelvű algoritmust Prolog programmá?

- Példafeladat: Hatékony hatványozási algoritmus
  - Alaplépés: a kitevő felezése, az alap négyzetre emelése.
  - A kitevő kettes számrendszerbeli alakja szerint hatványoz.
- Az algoritmust megvalósító C nyelvű függvény:

```
/* hatv(a, h) = a**h */
int hatv(int a, unsigned h)
{
  int e = 1;
  while (h > 0)
  {
    if (h & 1) e *= a;
    h >>= 1; a *= a;
  }
  return e;
}
```

- Az ciklusban három változót használunk: `a`, `h`, `e`:
  - `a` és `h` végértékére nincs szükség,
  - `e` végső értéke szükséges (ez a függvény eredménye).

A `hatv` C függvénynek megfelelő Prolog eljárás

- Kétargumentumú C függvény $\Longrightarrow$ 2+1-argumentumú Prolog eljárás.
- A függvény eredménye $\Longrightarrow$ utolsó arg.: `hatv(+A, +H, ?E)`: $A^H = E$.
- Ciklus $\Longrightarrow$ segédeljárás: `hatv(+A0, +H0, +E0, ?E)`: $A0^{H0} * E0 = E$.
- »`a`« és »`h`« C változók $\Longrightarrow$ »`+A0`« és »`+H0`« bemenő *paraméterek* (nem kell végérték),
  »`e`« C változó $\Longrightarrow$ »`+E0, ?E`« *akkumulátorpár* (kezdőérték, végérték).

```
int hatv(int a, unsigned h)
{       int e = 1;

  ism:  if (h > 0)
        {  if (h & 1)

             e *= a;
 /* else e is unchanged; */

           h >>= 1;
           a *= a;
           goto ism;
        } else return e;
}
```

```
hatv(A, H, E) :-
    hatv(A, H, 1, E).

hatv(A0, H0, E0, E) :- H0 > 0, !,
    (   H0 /\ 1 =:= 1
        % /\ ≡ bitenkénti "és"
    ->  E1 is E0*A0
    ;   E1 = E0
    ),
    H1 is H0 >> 1,
    A1 is A0*A0,
    hatv(A1, H1, E1, E).
hatv(_, _, E, E).
```

A C ciklus és a Prolog eljárás kapcsolata

- A ciklust megvalósító Prolog eljárás minden pontján minden C változónak megfeleltetethető egy Prolog változó (pl. `h`-nak `H0, H1, ...`):
  - A ciklusmag elején a C változók a megfelelő Prolog argumentumban levő változónak felelnek meg.
  - Egy C értékadásnak egy új Prolog változó bevezetése felel meg, az ez után következő kódban az új változó felel meg a C változónak.
  - Ha a diszjunkció, vagy if-then-else egyik ága megváltoztat egy változót, akkor a többi ágon is be kell vezetni az új Prolog változót, a régivel azonos értékkel (ld. `if (h & 1) ...`).
- A C ciklusmag végén a Prolog eljárást vissza kell hívni, argumentumaiban az egyes C változóknak pillanatnyilag megfeleltetett Prolog változóval.
- A C ciklus ciklus-invariánsa nem más mint a Prolog eljárás fejkommentje, a példában:

  `% hatv(+A0, +H0, +E0, ?E):` $A0^{H0} * E0 = E$.

Programhelyesség-bizonyítás (kieg. anyag)

- Egy algoritmus (függvény) specifikációja:
  - előfeltételek: a bemenő paramétereknek teljesíteniük kell ezeket,
  - utófeltételek: a paraméterek és az eredmény kapcsolatát írják le.
- Egy algoritmus helyes, ha minden, az előfeltételeket kielégítő adatra a függvény hibátlanul lefut, és eredményére fennállnak az utófeltételek.
- Példa: `x = mfoku_gyok(a,b,c)`
  - előfeltételek: `b*b-4*a*c >= 0`, $a \neq 0$
  - utófeltétel: `a*x*x+b*x+c = 0`
  - a program:

```
double mfoku_gyok(a, b, c)
double a, b, c;
{  double d = sqrt(b*b-4*a*c);
   return (-b+d)/2/a;
}
```

- A program helyességének bizonyítása lineáris kódra viszonylag egyszerű.

Ciklikus programok helyességének bizonyítása (kieg. anyag)

- A ciklusokat „fel kell vágni” egy ciklus-invariánssal, amely:
  - az előfeltételekből és a ciklust megelőző értékadásokból következik,
  - ha a ciklus elején fennáll, akkor a ciklus végén is (indukció),
  - belőle és a leállási feltételből következik a ciklus utófeltétele.

```
int hatv(int a0, unsigned h0) /*utófeltétel: hatv(a0, h0) = a0^h0 */
{ int e = 1, a = a0, h = h0;
  while /*ciklus-invariáns: a0^h0 == e*a^h */ (h > 0)
  {
    /* induláskor a kezdőértékek alapján triviálisan fennáll */
    if (h & 1) e *= a;          /* e' = e * a^(h&1) */
    h >>= 1;                    /* h' = (h-(h&1))/2 */
    a *= a;                     /* a' = a*a */
  }                             /*indukció: e'*a'^h' = ... = e*a^h */
  return e;
  /* Az invariánsból h = 0 miatt következik az utófeltétel */
}
```

Tartalom

Vezérlési eljárások, a `call/1` beépített eljárás

- Vezérlési eljárás: A Prolog végrehajtáshoz kapcsolódó beépített eljárás.
- A vezérlési eljárások többsége **magasabbrendű** eljárás, azaz olyan eljárás, amely egy vagy több argumentumát eljáráshívásként értelmezi.
- A meta-eljárások fő képviselője a `call(+Cél)`:
  - `Cél` egy struktúra vagy névkonstans (vö. `callable/1`).
  - Jelentése (deklaratív szemantika): `Cél` igaz.
  - Hatása (procedurális szemantika): a `Cél` kifejezést hívássá alakítja és végrehajtja.
- A klóztörzsben célként megengedett egy `X` változó használata, ezt a rendszer egy `call(X)` hívássá alakítja át.

```
| kétszer(X) :- call(X), X.

| ?- kétszer(write(ba)), nl.   ⟹    baba
| ?- listing(kétszer).         ⟹    kétszer(X) :-
                                       call(X), call(X).
```

Vezérlési szerkezetek mint eljárások

- A `call/1` argumentumában szerepelhetnek vezérlési szerkezetek is, mert ezek beépített eljárásként is jelen vannak a Prolog rendszerben:
  - `(',')/2`: konjunkció.
  - `(;)/2`: diszjunkció.
  - `(->)/2`: if-then; `(;)/2`: if-then-else.
  - `(\+)/1`: meghiúsulásos negáció.
- A `call`-ban szereplő vezérlési szerkezetek ugyanúgy futnak, mint az interpretált (azaz `consult`-tal betöltött) kód.
- A `Cél`-beli vágó csak a `call` belsejében vág (szülője a `call(Cél)` hívás).
- Példák:

```
| ?- _Cél = (kétszer(write(ba)), write(' ')), kétszer(_Cél), nl.
baba baba
| ?- kétszer((member(X, [a,b,c,d]), write(X), fail ; nl)).
abcd
abcd
```

call/1 példa: futási időt mérő meta-eljárás

```
% Kiírja Goal első megoldásának előállításához vagy a meghiúsuláshoz
% szükséges időt, a Txt szöveg kiséretében.
time(Txt, Goal) :-
        statistics(runtime, [T0,_]),  % T0 az indítás óta eltelt CPU idő,
                                      % msec-ban (szemétgyűjtés nélkül).
        (   call(Goal) -> Res = true
        ;   Res = false
        ),
        statistics(runtime, [T1,_]), T is T1-T0,
        format('~w futási idő = ~3d sec\n', [Txt,T]),
                % ~w formázó: kiírás a write/1 segítségével
                % ~3d formázó: I egész kiírása I/1000-ként, 3 tizedesre
        Res = true.                   % meghiúsul, ha Goal meghiúsult
```

További beépített vezérlési eljárások

- `once(Cél)`: `Cél` igaz, és csak az első megoldását kérjük. Definíciója:

  ```
  once(X) :- call(X), !.
  ```

  vagy, feltételes szerkezettel

  ```
  once(X) :- (   call(X) -> true   ).
  ```

- `true`: azonosan igaz, `fail`: azonosan hamis (mindig meghiúsul).
- `repeat`: végtelen sokszor igaz (végtelen választási pont). Definíciója:

  ```
  repeat.
  repeat :- repeat.
  ```

- A `repeat` eljárást egy mellékhatásos eljárás ismétlésére használhatjuk.
- Példa (egyszerű kalkulátor):

  ```
  bc :-  repeat, read(Expr),
              (   Expr = end_of_file -> true
              ;   Res is Expr, write(Expr = Res), nl, fail
              ),
         !.
  ```

- A végtelen választási pontot kötelező egy vágóval semlegesíteni!

Példa: magasabbrendű reláció definiálása (kieg. anyag)

- Az implikáció (`P` $\Rightarrow$ `Q`) megvalósítása negáció segítségével:

```
% P minden megoldása esetén Q igaz.
forall(P, Q) :-
     \+ (P, \+Q). % Szintaktikus emlékeztető:
                  % az első \+ után kötelező a szóköz!

| ?- _L = [1,2,3],
     % _L minden eleme pozitív:
     forall(member(X, _L), X > 0).
true ?
| ?- _L = [1,-2,3], forall(member(X, _L), X > 0).
no
| ?- _L = [1,2,3],
     % _L szigorúan monoton növő:
     forall(append(_,[A,B|_], _L), A < B).
true ?
```

- `forall/2` csak eldöntendő kérdés esetén használható.

Tartalom

A DCG (Definite Clause Grammars) formalizmus

- DCG: előfeldolgozó eszköz nyelvtani elemzők írásához.
- DCG szabály: *Fej*`-->`*Törzs*. $\Longrightarrow$ *Fej*(A_0, A_m)`:-`*Törzs*(A_0, A_m). A törzsben:
  - `{Cél}` $\Longrightarrow$ `Cél` (akkumulálást nem végző cél)
  - `[`E_1`,`E_2`,...,`E_k`]`, $k \geq 0$ $\Longrightarrow$ A_n `= [`E_1`,`E_2`,...,`E_k`|`A_{n+1}`]` (elemek akk.-a)
  - `p(`X_1`,`X_2`,...,`X_j`)`, $l \geq 0$ $\Longrightarrow$ `p(`X_1`,`X_2`,...,`X_j`,`A_n`,`A_{n+1}`)` (akk.-t végző cél)
  - Vezérlés: konj. (`,`), diszj. (`;`), ha-akkor (`->`), vágó (`!`), negáció (`\+`)
- Példa: egy lista pozitív elemeinek kigyűjtése

```
% pe(L, Pk0, Pk): Az L számlista pozitív elemeinek listája Pk0-Pk.
% Másszóval: L pozitív elemeinek listáját Pk elé füzve kapjuk Pk0-t
pe([], Pk0, Pk) :-     Pk0 = Pk.
pe([X|L], Pk0, Pk) :-  (   X > 0 -> Pk0 = [X|Pk1], pe(L, Pk1, Pk)
                       ;   pe(L, Pk0, Pk)
                       ).
```

- A DCG jelölést használó, a fentivel azonos kódot eredményező eljárás:

```
pe2([]) -->            [].
pe2([X|L]) -->         (   {X > 0} -> [X], pe2(L)
                       ;   pe2(L)
                       ).
```

A DCG formalizmus használata nyelvtani elemzésre

- Példa – decimális számok elemzését végző `szám(L0, L)` Prolog eljárás
- Az L0, L paraméterek: karakterkódok listái

```
% szám(L0, L): Az L0-L különbséglista számjegykódok nem-üres listája
% Másszóval: L0 elejéről leelemezhető egy szám, és marad L
szám --> számjegy, számmaradék.

% számmaradék(L0, L): Az L0-L különbséglista számjegykódok listája
számmaradék --> számjegy, számmaradék ; "".            % "" ≡ []

% számjegy(L0, L): L0 = [K|L], ahol K egy számjegy kódja
számjegy --> "0";"1";"2";"3";"4";"5";"6";"7";"8";"9".  % "9" ≡ [0'9]
```

- A `számjegy/2` eljárás egy másik megvalósítása:

```
számjegy --> [K], {decimális_jegy_kódja(K)}.                (*)

% K egy számjegy kódja.
decimális_jegy_kódja(K) :- K >= 0'0, K =< 0'9.
```

- A fenti (*) DCG szabály Prolog megfelelője:

```
számjegy(L0, L) :-
        L0 = [K|L],                 % K a következő listaelem
        decimális_jegy_kódja(K).    % megfelelő-e a K?
```

DCG nyelvtani elemzés – további részletek

- Az elemzés – a Prolog végrehajtás miatt – nem-determinisztikus, pl.

```
| ?- szám("123 abc", L).
L = " abc" ?  ;          % leelemeztük a 123 számot
L = "3 abc" ?  ;         % leelemeztük a 12 számot
L = "23 abc" ?  ;        % leelemeztük az 1 számot
no
```

- A számmaradék eljárás determinisztikus változata

```
% számmaradék2(L0, L): L0-L számjegykódok maximális listája
számmaradék2 --> (   számjegy -> számmaradék2
                 ;   ""
                 ).
```

vagy

```
számmaradék3 --> számjegy, !, számmaradék3. % A vágó köré nem kell {}
számmaradék3 --> "".
```

- Futás:

```
| ?- szám2("123 abc", L).
L = " abc" ? ;           % leelemeztük a (lehető leghosszabb) 123 számot
no
```

Az elemző kiegészítése jelentéshordozó argumentumokkal

- Egy DCG szabály az elemzéssel párhuzamosan további (kimenő) argumentum(ok)ban felépítheti a kielemzett dolog „jelentését”
- Példa: szám elemzése és értékének kiszámítása:

```
% Leelemezhető egy Sz értékű nem-üres számjegysorozat
szám(Sz) --> számjegy(J), számmaradék(J, Sz).

% Leelemezhető számjegyek egy esetleg üres listája, amelynek
% az eddig leelemzett Sz0-val együtt vett értéke Sz.
számmaradék(Sz0, Sz) -->
    számjegy(J), !, {Sz1 is Sz0*10+J}, számmaradék(Sz1, Sz).
számmaradék(Sz0, Sz0) --> [].

% leelemezhető egy J értékű számjegy.
számjegy(J) --> [K], {decimális_jegy_kódja(K), J is K-0'0}.
```

```
| ?- szám(Sz, "102 56", L). ⟹   L = " 56", Sz = 102; no
```

- A `számmaradék` DCG szabály Prolog alakja:

```
számmaradék(Sz0, Sz, L0,L) :-
    számjegy(J, L0,L1),!, Sz1 is Sz0*10+J, számmaradék(Sz1, Sz, L1,L).
számmaradék(Sz0, Sz0, L0,L) :- L=L0.
```

- Itt két akkumulátorpár van: egy „kézi” (`Sz`) és egy DCG-ből generált (`L`).

Aritmetikai kifejezések elemzése

- Egyszerű aritmetikai kifejezések elemzése és kiértékelése.

```
% kif0(Z, L0, L): L0-L egy Z aritmetikai kifejezéssé elemezhető ki
kif0(X+Y) --> tag0(X), "+", !, kif0(Y).
kif0(X-Y) --> tag0(X), "-", !, kif0(Y).
kif0(X) -->   tag0(X).

tag0(X) --> szám(X).                % egyelőre
```

`| ?- kif0(Z, "4-2+1", []).` $\Longrightarrow$ `Z = 4-(2+1)` Jobbról balra elemez!

- Egy lehetséges javítás

```
kif(Z) --> tag(X), kifmaradék(X, Z).

kifmaradék(X, Z) --> "+", tag(Y), !, kifmaradék(X+Y, Z).
kifmaradék(X, Z) --> "-", tag(Y), !, kifmaradék(X-Y, Z).
kifmaradék(X, X) --> [].

tag(Z) --> szám(X), tagmaradék(X, Z).

tagmaradék(X, Z) --> "*", szám(Y), !, tagmaradék(X*Y, Z).
tagmaradék(X, Z) --> "/", szám(Y), !, tagmaradék(X/Y, Z).
tagmaradék(X, X) --> [].
```

`| ?- kif(Z, "5*4-2+1", []), Val is Z.` $\Longrightarrow$ `Z = 5*4-2+1, Val = 19 ? ; no`

Tartalom

Egy nagyobb DCG példa: „természetes” nyelvű beszélgetés

```
% mondat(Alany, Áll, L0, L): L0-L kielemezhető egy Alany alanyból és Áll
% állítmányból álló mondattá. Alany lehet első vagy második személyű
% névmás, vagy egyetlen szóból álló (harmadik személyű) alany.
mondat(Alany, Áll) -->
        {én_te(Alany, Ige)}, én_te_perm(Alany, Ige, Áll).
mondat(Alany, Áll) -->
        szó(Alany), szavak(Áll).

% én_te(Alany, Ige):
% Az Alany első/második személyű névmásnak megfelelő létige az Ige.
én_te("én", "vagyok").
én_te("te", "vagy").

% én_te_perm(Ki, Ige, Áll, L0, L): L0-L kielemezhető egy Ki
% névmásból, Ige igealakból és Áll állítmányból álló mondattá.
én_te_perm(Alany, Ige, Áll)  -->
        (   szó(Alany), szó(Ige), szavak(Áll)
        ;   szó(Alany), szavak(Áll), szó(Ige)
        ;   szavak(Áll), szó(Ige), szó(Alany)
        ;   szavak(Áll), szó(Ige)
        ).
```

Példa: „természetes” nyelvű beszélgetés – szavak elemzése

```
% szó(Sz, L0, L): L0-L egy Sz betűsorozatból álló (nem üres) szó.
szó(Sz) --> betű(B), szómaradék(SzM), {illik([B|SzM], Sz)}, köz.

% szómaradék(Sz, L0, L): L0-L egy Sz kódlistából álló (esetleg üres) szó.
szómaradék([B|Sz]) --> betű(B), !, szómaradék(Sz).
szómaradék([]) --> [].

% illik(Szó0, Szó): Szó0 = Szó, vagy a kezdő kis-nagy betűben különböznek.
illik([B0|L], [B|L]) :-
        (   B = B0 -> true
        ;   abs(B-B0) =:= 32
        ).

% köz(L0, L): L0-L nulla, egy vagy több szóköz.
köz --> (   " " -> köz ; ""   ).

% betű(K, L0, L): L0-L egy K kódú "betű" (különbözik a " .?" jelektől)
betű(K) --> [K], {\+ member(K, " .?")}.

% szavak(SzL, L0, L): L0-L egy SzL szó-lista.
szavak([Sz|Szk]) -->          szó(Sz), (   szavak(Szk)
                                       ;   {Szk = []}
                                       ).
```

Példa: „természetes” nyelvű beszélgetés – párbeszéd-szervezés

```
% :- type mondás --> kérdez(szó) ; kijelent(szó,list(szó)) ; un.

% Megvalósít egy párbeszédet.
párbeszéd :-
        repeat,
            read_line(L),   % beolvas egy sort, L a karakterkódok listája
            (   menet(Mondás, L, []) ->  feldolgoz(Mondás)
            ;   write('Nem értem\n'), fail
            ),
        Mondás = un, !.

% menet(Mondás, L0, L): Az L0-L kielemzett alakja Mondás.
menet(kérdez(Alany)) -->
        {kérdő(Szó)}, mondat(Alany, [Szó]), "?".
menet(kijelent(Alany,Áll)) -->
        mondat(Alany, Áll), ".".
menet(un) --> szó("unlak"), ".".

% kérdő(Szó): Szó egy kérdőszó.
kérdő("mi").
kérdő("ki").
kérdő("kicsoda").
```

Példa: „természetes” nyelvű beszélgetés – válaszok előállítása

```
:- dynamic tudom/2.

% feldolgoz(Mondás): feldolgozza a felhasználótól érkező Mondás üzenetet.
feldolgoz(un) :-
        write('Én is.\n').
feldolgoz(kijelent(Alany, Áll)) :-
        assertz(tudom(Alany,Áll)),
        write('Felfogtam.\n').
feldolgoz(kérdez(Alany)) :-
        tudom(Alany, _), !,
        válasz(Alany).
feldolgoz(kérdez(_)) :-
        write('Nem tudom.\n').

% Felsorolja az Alany ismert tulajdonságait.
válasz(Alany) :-
        tudom(Alany, Áll),
        (   member(Szó, Áll), format('~s ', [Szó]), fail
        ;   nl
        ), fail.
válasz(_).
```

Beszélgetős DCG példa – egy párbeszéd

```
| ?- párbeszéd.
|: Magyar legény vagyok én.
Felfogtam.
|: Ki vagyok én?
Magyar legény
|: Péter kicsoda?
Nem tudom.
|: Péter tanuló.
Felfogtam.
|: Péter jó tanuló.
Felfogtam.
|: Péter kicsoda?
tanuló
jó tanuló
|: Boldog vagyok.
Felfogtam.
|: Én vagyok Jeromos.
Felfogtam.
|: Te egy Prolog program vagy.
Felfogtam.
|: Ki vagyok én?
Magyar legény
Boldog
Jeromos
|: Okos vagy.
Felfogtam.
|: Ki vagy  te?
egy Prolog program
Okos
|: Valóban?
Nem értem
|: Unlak.
Én is.
```