Deklaratív programozás

Szeredi Péter[1] Hanák Péter[2]

[1] szeredi@cs.bme.hu
BME Számítástudományi és Információelméleti Tanszék

[2] hanak@emt.bme.hu
BME Számítástudományi és Információelméleti Tanszék

2022. ősz

Az előadók köszönetüket fejezik ki Kápolnai Richárdnak.

A tárgy témája

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

I. rész

Bevezetés

Tartalom

Honlap, Elektronikus TanárSegéd

- Honlap: `https://dp.iit.bme.hu`
  a jelen félév honlapja: `https://dp.iit.bme.hu/dp-current`
- ETS, az Elektronikus TanárSegéd
  `https://dp.iit.bme.hu/ets` (id: `etsuser`, pwd: `ets+2022#`)

Deklaratív programozás: követelmények

Nagy házi feladat (NHF)

- Programozás mind funkcionális, mind logikai nyelven
- Mindenkinek önállóan kell kódolnia (programoznia)!
- Hatékony (időlimit!), jól dokumentált („kommentezett”) programok
- A programokhoz 5–10 oldalas fejlesztői dokumentáció PDF-ben
- A funkcionális progamozási NHF kiadása legkésőbb a 3., a logikai programozási NHF-é a 7. oktatási héten a honlapon
- A funkcionális progamozási NHF beadása a 6. a logikai programozási NHF-é a 10. oktatási héten, elektronikus úton (ld. honlap)
- A beadáskor és a pontozáskor külön-külön tesztsorozatot használunk (nehézségben hasonlókat, de nem azonosakat)
- Azok a programok, amelyek megoldják a tesztesetek 80%-át, *létraversenyen* vesznek részt (hatékonyság, gyorsaság plusz pontokért)
- Azok a hallgatók, akik **mindkét** nyelvből bejutnak a létraversenybe, és a kis házi feladatokra vonatkozó követelményeket (ld. alább) is teljesítik, **megajánlott** jegyet kapnak
- A beadási határidőig többször is beadható, csak az utolsót értékeljük

Deklaratív programozás: követelmények (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 a funkcionális, ill. a logikai programozási szemléletét;
  - 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".

Deklaratív programozás: követelmények (folyt.)

Kis házi feladatok (KHF)

- 3 feladat funkcionális, 3 logikai programozási nyelven,
- Beadás elektronikus úton (ld. honlap)
- Egy KHF beadása érvényes, ha minden tesztesetre lefut
- **Kötelező** legalább 2 KHF érvényes beadása a funkcionális és a logikai részből is
- Minden feladat jó megoldásáért 1-1 jutalompont (azaz a 100 alappont feletti pont) jár
- Ha valaki mindhárom feladatot megoldja az adott nyelven, azzal megduplázza a ponszámát, azaz 3 helyett 6 pontot kap
- Minden KHF-nek külön határideje van, pótlási lehetőség nincs
- A KHF-ek egyre összetettebbek és részben egymásra épülnek – érdemes minél előbb elkezdeni a KHF-ek beadását!
- A házi feladatot önállóan kell elkészíteni! Másolás esetén kötelesek vagyunk fegyelmi eljárást indítani *("Beadandó feladat ... elkészíttetése mással"):*
  `http://www.kth.bme.hu/document/189/original/bme_rektori_utasitas_05.pdf`

Deklaratív programozás: követelmények (folyt.)

Gyakorlatok

- A 2. oktatási héttől kezdődően 2 órás tantermi gyakorlatok:
  - 2. oktatási hét: szeptember 13., szeptember 15. (funkcionális 1)
  - 3. oktatási hét: szeptember 20., szeptember 22. (funkcionális 2)
  - 5. oktatási hét: október 4., október 6. (funkcionális 3)
  - 6. oktatási hét: október 11., október 13. (logikai 1)
  - 8. oktatási hét: október 25., október 27. (logikai 2)
  - 10. oktatási hét: november 8., november 10. (logikai 3)
- A gyakorlatok anyagát a honlapon tesszük közzé, csak annak nyomtatjuk ki, aki kifejezetten kéri
- A gyakorló, ill. a kis házi feladatok megoldását távkonzultációval is segíteni szeretnénk, ezért minden héten csütörtökön 17 és 19 óra között távjelenlétes konzultációt tartunk a Teamsben. A távkonzultációra a Teamsben kell jelentkezni (részletek később).
- A tantermi gyakorlatokon is ajánlott a hordozható számítógép (laptop) használata
- További Prolog gyakorlási lehetőség az ETS, valamint a PLWIN rendszerben (gyakorló feladatok, lásd honlap)

Deklaratív programozás: követelmények (folyt.)

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
- Pótpót-zárthelyit (aláíráspótló vizsgát) NEM tartunk!
- Mind a funkcionális, mind a logikai részből **kötelező** a zárthelyi érvényes teljesítése, kivéve megajánlott jegy esetén (lásd alább)
- A zárthelyiken semmilyen jegyzet, segédlet nem használható
- 40%-os szabály: a maximális pontszám 40%-a kell az érvényességhez
- Zárthelyi időpontok:
  - Funkcionális: 2022. 10. 14. 8:00
  - Logikai: 2022. 11. 18. 8:00
  - 1. PZH (mindkét nyelven): 2022. 12. 02.
  - 2. PZH (mindkét nyelven): később adjuk meg
- A zárthelyik súlya az osztályzatban: 43%-43%
  (a 100 pontból összesen 86)

Deklaratív programozás: követelmények (folyt.)

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

IMSc pontozás

- A tantárgyból kétféle módon szerezhető IMSc pont:
  - egyes zárthelyik során pluszfeladatok megoldásával (maximum 10 pont),
  - a létraversenyen a megajánlott jeles érdemjegyhez szükséges 40%-os teljesítés felett minden további 10% teljesítéséért mindkét nyelv esetén 1 – 1 pont (összesen maximum 12 pont).
- A hallgató a fenti módokon szerzett pontok összegét kapja, de 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 biztosított.

Tartalom

Bevezető feladat: adott értékű kifejezés előállítása

- A feladat: írjunk programot a következő feladvány megoldására:
  - Adott számokból a négy alapművelettel (+, -, *, /) építsünk egy megadott értékű aritmetikai kifejezést!
    (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: $1 + 6 * (3 + 4)$, $(1 + 3)/4 + 6$

Hogyan ábrázoljuk az aritmetikai kifejezéseket?

- Első ötlet: füzér (string)
- Egy könnyebben kezelhető, strukturált ábrázoláshoz írjuk fel az aritmetikai kifejezés – ⟨ akif ⟩ – szintaxisát:

  ⟨ akif ⟩ ::= ⟨ szám ⟩ |
  (⟨ akif ⟩ ⟨ művelet ⟩ ⟨ akif ⟩)
  ⟨ művelet ⟩ ::= + | - | * | /

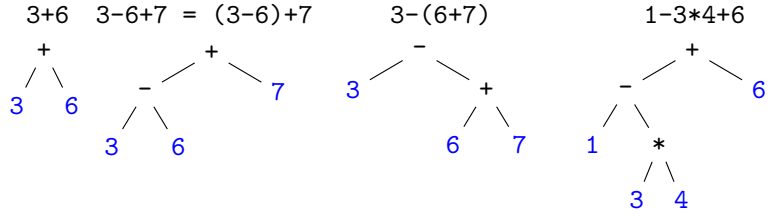
  (az egyértelműség kedvéért minden részkifejezést zárójelbe teszünk)
- Az ⟨ akif ⟩ adatstruktúra egy lehetséges megvalósítása C nyelven:

```
enum akif_fajta {Number, Plus, Minus, Times, Div};
struct akif { enum akif_fajta fajta;
              union { struct { int ertek;
                             } szam;
                      struct { struct akif *bal;
                               struct akif *jobb;
                             } osszetett;
                    } u;
            };
```


Az aritmetikai kifejezések matematikai absztrakciója

Milyen matematikai struktúra feleltethető meg ⟨ akif ⟩-nek?

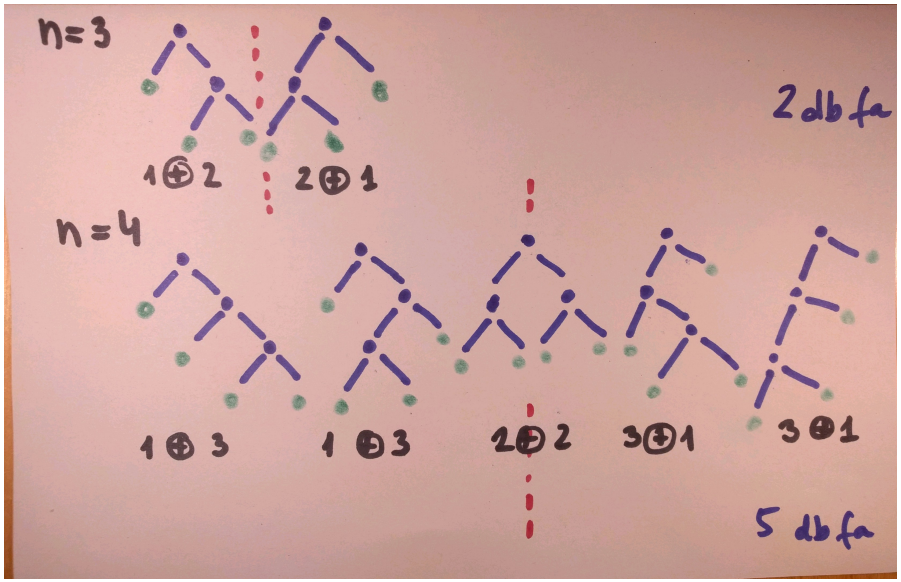
- Egy bináris fa
  - melynek levelei számokkal vannak címkézve
  - csomópontjai pedig a négy alapművelet valamelyikével
- Példák:



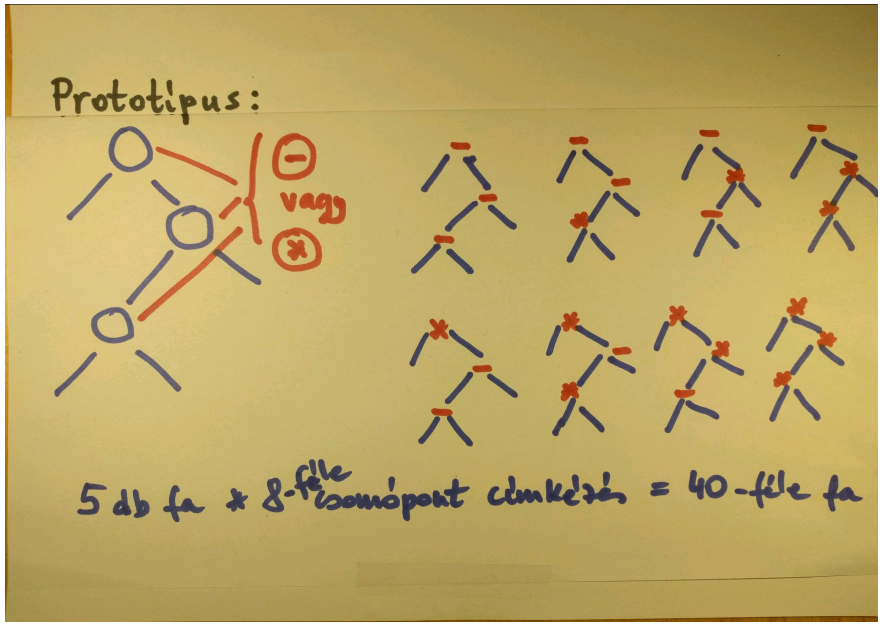
A bevezető feladat megoldásának terve

- Egyszerűsített feladat:
  levelek: `1`, `3`, `4`, `6`; *műveletek:* `-`, `*`; a kifejezés elvárt értéke: `-11`
- Állítsuk elő az adott levelekkel bíró összes ⟨ akif ⟩-et, majd válogassuk ki azokat, amelyek értéke az adott szám (**brute-force, generate-and-test**) (n a levelek, m a műveletek száma, a példában $n = 4, m = 2$):
  1. Állítsuk elő az összes adott levélszámú, címkézetlen bináris fát (legyen f ezek száma, pl. $n = 4$ esetén $f = 5$)
  2. A csomópontokba minden lehetséges módon helyezzünk el műveleti jeleket ($f \cdot m^{n-1}$ fa)
  3. Állítsuk elő a levelek összes permutációját ($n!$ db)
  4. A csomópont-címkézett fa leveleibe írjunk be minden permutációt ($f \cdot m^{n-1} \cdot n!$ darab ⟨ akif ⟩, a példában $5 \cdot 2^{4-1} \cdot 4! = 960$)
- Számítsuk ki minden így előállított ⟨ akif ⟩ értékét, adjuk vissza azokat, amelyekre ez az elvárt számértékkel egyezik
- A feladat megoldásai: `4-(6-1)*3`, `1-(6-3)*4`, `4-3*(6-1)`, `1-4*(6-3)`.

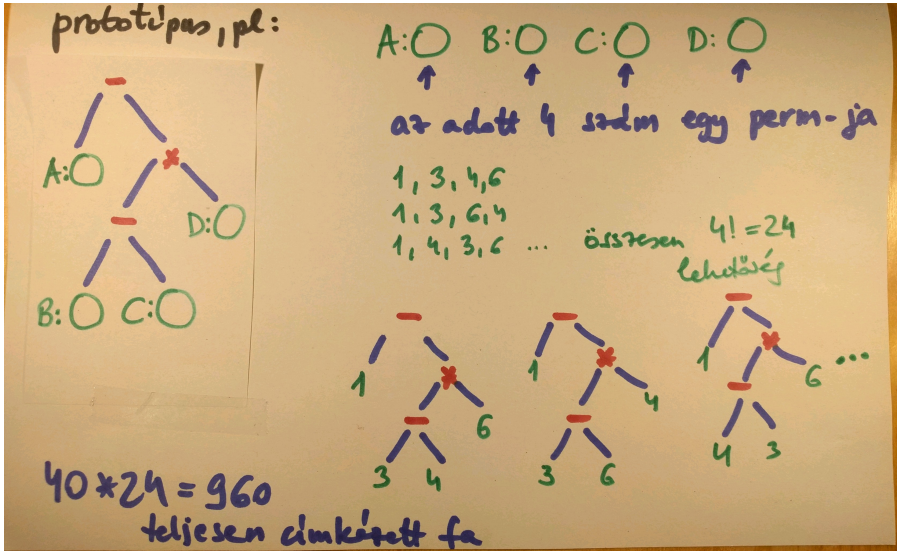
Címkézetlen bináris fák n levéllel



4-levelű fák - műveletek a csomópontokban



4-levelű, teljesen cimkézett fák

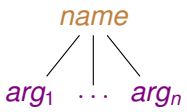


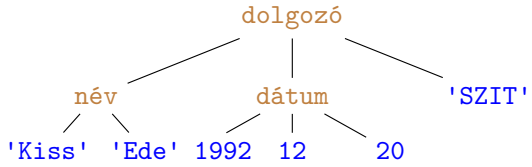
Tartalom

A Prolog nyelv adatfogalma

A Prolog adatokat Prolog kifejezésnek hívjuk (angolul: term). Fajtái:

- egyszerű kifejezés: számkonstans (pl. `3`, `4.25`, `-5`), névkonstans (pl. `alma`, `'SZIT'`, `+`), vagy változó (pl. `X`, `_`)
- összetett kifejezés (struktúra) kanonikus alakja: *name*$(arg_1, \ldots, arg_n)$
  - *name* egy névkonstans, az arg_i mezők tetsz. Prolog kifejezések
  - példa: `dolgozó(név('Kiss','Ede'), dátum(1992,12,20),'SZIT')`.
  - az összetett kifejezések valójában fastruktúrát alkotnak:



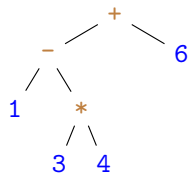


- a Prolog változó a matematikában használt változónak felel meg: egyetlen, esetleg még ismeretlen adatot jelent. A változó legfeljebb egyszer kaphat értéket. Egy Prolog változó összetett kifejezés részeként többször is megjelenhet, pl. `név(X,X)` egy olyan embert jelöl, akinek a vezetékneve és keresztneve ugyanaz (`X`).

Szintaktikus „édesítőszerek” Prologban

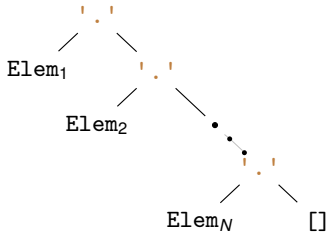
- Egy- és kétargumentumú struktúrák operátoros (infix, prefix stb.) írásmódja: `1+2`≡`+(1,2)`

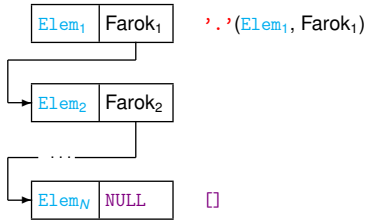
```
| ?- write_canonical(1-3*4+6).
```
(Ez a beépített elj. a kanonikus alakot írja ki:)
```
+(-(1,*(3,4)),6)
```



- Listák mint speciális struktúrák

```
| ?- write_canonical([6,4,3]).
'.'(6,'.'(4,'.'(3,[])))
```





Aritmetikai kifejezések Prologban – 1. lépés: ellenőrzés

Írjunk egy `kif` nevű Prolog eljárást (azaz Boole-értékű függvényt)!
(„igaz” eredmény esetén sikerről, egyébként meghiúsulásról beszélünk.)
A `kif(X)` hívás sikeresen fut le, ha `X` egy olyan kifejezés, amely számokból a négy alapművelet (+, -, *, /) segítségével épül fel (röviden, ha `X` **helyes**).

- Az alábbi sorokat helyezzük el pl. a `kif0.pl` file-ban:

```
% kif(K): K helyes, azaz K egy számokból, alapműveletekkel képzett kifejezés.
kif(K) :-   number(K).      % K helyes, ha K szám. (number beépített elj.)
kif(X+Y) :- kif(X), kif(Y). % X+Y helyes, ha X helyes és Y helyes
kif(X-Y) :- kif(X), kif(Y). % X-Y helyes, ha ...
kif(X*Y) :- kif(X), kif(Y). % ...
kif(X/Y) :- kif(X), kif(Y).
```

- Betöltése: `| ?- compile(kif0).` vagy `| ?- consult(kif0).`
- Futtatás nyomkövetés nélkül és nyomkövetéssel (`consult`-ot követően):

```
| ?- kif(alma).
no
| ?- kif(1+2).
yes
| ?-
```

```
| ?- trace, kif(alma).
        1      1 Call: kif(alma) ?
        2      2 Call: number(alma) ?
        2      2 Fail: number(alma) ?
        1      1 Fail: kif(alma) ?
no
```

Aritmetikai kifejezések ellenőrzése – továbbfejlesztett változat

- A `kif` Prolog eljárás segédeljárást használó változata:

```
% kif2(K): K számokból a négy alapművelettel képzett kifejezés.
kif2(Kif) :-
    number(Kif).
kif2(Kif) :-
    alap4(X, Y, Kif),
    kif2(X), kif2(Y).
```

- Az `alap4` segédeljárás:

```
% alap4(X, Y, Kif): A Kif kifejezés az X és Y kifejezésekből
%                    a négy alapművelet egyikével áll elő.
alap4(X, Y, X+Y).
alap4(X, Y, X*Y).
alap4(X, Y, X-Y).
alap4(X, Y, X/Y).
```


Aritmetikai kifejezés leveleinek listája

- Az alábbi eljárás ellenőrzi, hogy `Kif` egy számokból alapműveletekkel felépített kifejezés-e, és ha igen, előállítja ennek levéllistáját

```
% kif_levelek(Kif, L): A számokból alapműveletekkel felépülő Kif
%                       kifejezés leveleiben levő számok listája L.
kif_levelek(Kif, L) :-
    number(Kif), L = [Kif].   % L egyelemű, Kif-ből álló lista
    % A = B egy infix alakban írható beépített eljárás, működése:
    % A-t és B-t egyesíti: változóbehelyettesítésekkel azonos alakra hozza.
kif_levelek(Kif, L) :-
    alap4(K1, K2, Kif),
    kif_levelek(K1, L1),
    kif_levelek(K2, L2),
    append(L1, L2, L).
| ?- kif_levelek(2/3-4*(5+6), L).     ⟶ L = [2,3,4,5,6]
```

- Az `append` egy beépített eljárás, fejkommentje és példafutása:

```
% append(L1, L2, L): Az L1 és L2 listák összefűzése az L lista, L1 ⊕ L2 = L.
| ?- append([1,2], [3,4], L).        ⟶ L = [1,2,3,4]
```


Az `append/3` eljárás többirányú használata

- Az `append` eljárás a fejkommentje által leírt *relációt* valósítja meg, sokféle módon használható, és több választ is adhat (új válasz kérése `;`-vel)

```
% append(L1, L2, L): L1 és L2 összefűzése az L lista, azaz L1 ⊕ L2 = L.

| ?- append(L, [3], [1,2,3]).    % Mi L ⊕ [3] = [1,2,3] megoldása?
L = [1,2] ? ; no                 % nincs TÖBB válasz
| ?- append([1,2], L, [1,2,3]). % Mi [1,2] ⊕ L = [1,2,3] megoldása?
L = [3] ? ;  no
| ?- append(L1, L2, [1,2,3]).    % [1,2,3] hogyan bontható két részre?
L1 = [], L2 = [1,2,3] ? ;
L1 = [1], L2 = [2,3] ? ;
L1 = [1,2], L2 = [3] ? ;
L1 = [1,2,3], L2 = [] ? ; no
| ?- append(_, [_X,_X], [3,4,4]). % [3,4,4] utolsó két eleme azonos?
yes
| ?- append(L1, [2], L2).
L1 = [], L2 = [2] ? ;
L1 = [_A], L2 = [_A,2] ? ;
L1 = [_A,_B], L2 = [_A,_B,2] ? ;  % végtelen sok válasz, problémás ...
```


Adott levéllistájú aritmetikai kifejezések előállítása

- A `kif_levelek` eljárás sajnos nem használható „visszafelé”, végtelen ciklusba esik, lásd pl. `| ?- kif_levelek(Kif, [1]).`
- Ez javítható a hívások átrendezésével és új feltételek beszúrásával:

```
% kif_levelek(+Kif, -L):
% Kif levéllistája L.
kif_levelek(Kif, L) :-
    L = [Kif],
    number(Kif).
kif_levelek(Kif, L) :-
    alap4(K1, K2, Kif),

    kif_levelek(K1, L1),
    kif_levelek(K2, L2),
    append(L1, L2, L).
```

```
% kif_levelek_v(-Kif, +L):
% Kif levéllistája L.
kif_levelek_v(Kif, L) :-
    L = [Kif],
    number(Kif).
kif_levelek_v(Kif, L) :-
    append(L1, L2, L),
    L1 \= [], L2 \= [],
    % L1, L2 nem-üres listák
    kif_levelek_v(K1, L1),
    kif_levelek_v(K2, L2),
    alap4(K1, K2, Kif).
```

- Az `X \= Y` beépített eljárás *csakkor* sikerül, ha `X = Y` meghiúsul.

```
| ?- kif_levelek_v(K, [1,3,4]).
K = 1+(3+4) ? ; K = 1-(3+4) ? ; K = 1*(3+4) ? ; K = 1/(3+4) ? ; ...
```

Adott értékű kifejezés előállítása

- Bevezető példánk megoldásához szükségesek további nyelvi elemek
  - A `lists` könyvtárban található `permutation` eljárás:

    `% permutation(L, PL): PL` *az* `L` *lista permutációja.*
  - Az `=:=` beépített aritmetikai eljárás mindkét argumentumában aritmetikai kifejezést vár, ezeket kiértékeli, és *csakkor* sikerül, ha az értékek aritmetikailag megegyeznek, pl.

    `| ?- 4+2 =:= 3*3.` ⟶ `no` `| ?- 4+2 =:= 2*3.` ⟶ `yes`
  - A `catch(Hívás, Kiv, KivHívás)` beépített eljárás végrehajtja `Hívás`-t, és ha ez egy `Kiv` kivételt dob, végrehajtja `KivHívás`-t.
- A „kedvcsináló” feladat megoldása:

```
% levelek_ertek_kif(L, Ertek, Kif): Kif az L listabeli számokból a négy
% alapművelet segítségével felépített olyan kifejezés, amelynek értéke Ertek.
levelek_ertek_kif(L, Ertek, Kif) :-
    permutation(L, PL), kif_levelek_v(Kif, PL),
    catch(Kif =:= Ertek, _, fail).  % fail mindig meghiúsul.

| ?- levelek_ertek_kif([1,3,4], 11, Kif).
Kif = 3*4-1 ? ; Kif = 4*3-1 ? ; no
```


Adott értékű kifejezés előállítása – a teljes kód

```
:- use_module(library(lists), [permutation/2]). % importálás

% Kif az L listabeli számokból felépített, Ertek értékű kifejezés.
levelek_ertek_kif(L, Ertek, Kif) :-
    permutation(L, PL), kif_levelek_v(Kif, PL),
    catch(Kif =:= Ertek, _, fail).

% Az alapműveletekkel felépített Kif levéllistája L.
kif_levelek_v(Kif, L) :-
    L = [Kif], number(Kif).
kif_levelek_v(Kif, L) :-
    append(L1, L2, L), L1 \= [], L2 \= [],
    kif_levelek_v(K1, L1), kif_levelek_v(K2, L2),
    alap4(K1, K2, Kif).

% alap4(X, Y, K): A K kifejezés X-ből és Y-ból egy alapművelettel áll elő.
alap4(X, Y, X+Y).
alap4(X, Y, X-Y).
alap4(X, Y, X*Y).
alap4(X, Y, X/Y).
```


Tartalom

Elixir-kifejezések

- Elixir: nem logikai, hanem *funkcionális* programnyelv
- Összetett kifejezéseket, függvényhívásokat értékel ki:

```
iex> [ 1 - 3 * 4 + 6, 1 - 3 / 4 + 6 ]
[-5, 6.25]
iex> Enum.to_list 1..3
[1, 2, 3]
iex> { 1/2, ?+, '+', 1+1 }
{0.5, 43, '+', 2}
```

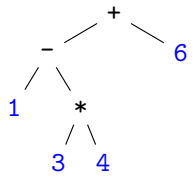
- *Hármas:* $\{k_1, k_2, k_3\}$, ahol k_i tetszőleges Elixir-kifejezés; *pár:* $\{k_1,k_2\}$
- A *listajelölő* (list comprehension) kifejezésnek a *halmazjelölőhöz* (set comprehension) hasonló kifejező ereje van:

```
iex> for x <- [1,2,3], do: x                    # {x | x ∈ {1,2,3}}
[1, 2, 3]
iex> for x <- [1,2,3], x*x >= 4, do: x          # {x | x ∈ {1,2,3}, x² ≥ 4}
[2, 3]
iex> for x <- [1,2,3], (y <- Enum.to_list 1..x), do: {x, y}
                                  # {(x,y) | x ∈ {1,2,3}, y ∈ {1 .. x}}
[{1, 1}, {2, 1}, {2, 2}, {3, 1}, {3, 2}, {3, 3}]
```

Aritmetikai kifejezések ábrázolása

- A Prologgal ellentétben az Elixir automatikusan sem ábrázolni, sem felsorolni nem tudja az aritmetikai kifejezéseket
- A Prolog egy aritmetikai kifejezést faként ábrázol:

```
| ?- write_canonical(1-3*4+6).
+(-(1,*(3,4)),6)
yes
```



- Az Elixirben explicit módon fel kell sorolni és ki kell értékelni az összes fát
- Példaprogramunkban a fenti aritmetikai kifejezést (önkényesen) egymásba ágyazott hármasokkal ábrázoljuk:

```
{{1, '-', {3, '*', 4}}, '+', 6}
```


Adott méretű fák felsorolása

- Faelrendezések felsorolása például csupa 1-esekből és '+' műveletekből
- Összesen 5 db 4 levelű fa van:

```
{1, '+', {1, '+', {1, '+', 1}}}
{1, '+', {{1, '+', 1}, '+', 1}}
{{1, '+', 1}, '+', {1, '+', 1}}
{{1, '+', {1, '+', 1}}, '+' ,1}
{{{1, '+', 1}, '+', 1}, '+', 1}
```

Elixir-kód (file: dp_kif.ex)

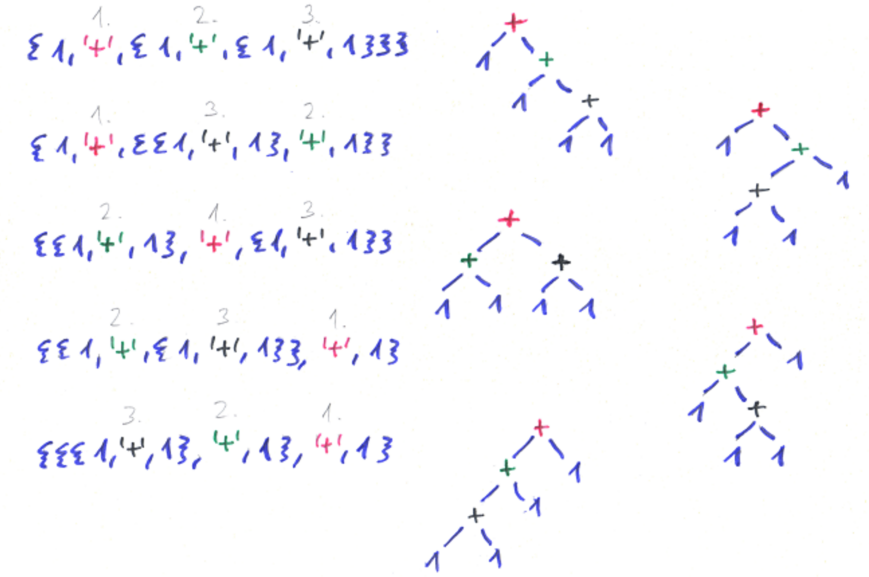
```
@type fa :: 1 | {fa, '+', fa}
@spec fak(n :: integer) :: fs :: [fa]
# Az összes n levelű fa listája fs
def fak(1), do: [1]
def fak(n) do
  for \ # folytatódó sor jelzése
    i   <- (Enum.to_list 1..(n-1)),
    bfa <- fak(i),
    jfa <- fak(n-i),
  do: {bfa, '+', jfa}
end
```

Halmazjelölővel

Az n levelű fák halmaza $\mathbb{F}(n)$, melyre

- $\mathbb{F}(1) == \{1\}$
- $\mathbb{F}(n) == \{ (b, \text{'+'}, j) \mid i \in 1..n-1, b \in \mathbb{F}(i), j \in \mathbb{F}(n-i) \}$

Csupa 1-esekből és '+' műveletekből álló fák



1-esekből és '+' műveletekből álló, 3 és 5 levelű fák

- Állítsuk elő a három- és ötlevelű fákat a `fak` függvény meghívásával!

```
iex> Dp.Kif.fak 3
[ {1, '+', {1, '+', 1}},
  {{1, '+', 1}, '+', 1}
]
iex> Dp.Kif.fak 5
[ {1, '+', {1, '+', {1, '+', {1, '+', 1}}}},
  {1, '+', {1, '+', {{1, '+', 1}, '+', 1}}},
  {1, '+', {{1, '+', 1}, '+', {1, '+', 1}}},
  {1, '+', {{1, '+', {1, '+', 1}}, '+', 1}},
  {1, '+', {{{1, '+', 1}, '+', 1}, '+', 1}},
  {{1, '+', 1}, '+', {1, '+', {1, '+', 1}}},
  {{1, '+', 1}, '+', {{1, '+', 1}, '+', 1}},
  {{1, '+', {1, '+', 1}}, '+', {1, '+', 1}},
  {{{1, '+', 1}, '+', 1}, '+', {1, '+', 1}},
  {{1, '+', {1, '+', {1, '+', 1}}}, '+', 1},
  {{1, '+', {{1, '+', 1}, '+', 1}}, '+', 1},
  {{{1, '+', 1}, '+', {1, '+', 1}}, '+', 1},
  {{{1, '+', {1, '+', 1}}, '+', 1}, '+', 1},
  {{{{1, '+', 1}, '+', 1}, '+', 1}, '+', 1}
]
```

- Ennyi alapismerettel már belefoghatunk a feladvány megoldásába. Kezdjük!

Adott levéllistájú aritmetikai kifejezések felsorolása

- Segédfv: egy lista összes lehetséges kettévágása nem üres listákra

```
iex> Dp.Kif.kettevagasok([1,3,4,6])
[ {[1], [3,4,6]}, {[1,3], [4,6]}, {[1,3,4], [6]} ]
```

- Kifejezések adott számokból *adott sorrendben*, 4 alapműveletből:

Elixir-kód

```
@type muv :: '+' | '-' | '*' | '/'
@type kif :: integer | {kif, muv, kif}
@spec kifek(xs::[integer]) :: ks::[kif]
# xs-beli számokból épített kif-ek listája ks
kifek([x]), do: [x]
kifek(xs),  do:
  for \
    {xbs, xjs} <- kettevagasok(xs),
    b          <- kifek(xbs),
    j          <- kifek(xjs),
    m          <- ['+', '-', '*', '/'],
    do: {b, m, j}
```

Halmazjelölővel

Az L listából képezhető kifejezések halmaza $\mathbb{K}(L)$, melyre

- $\mathbb{K}([X]) == \{X\}$
- $\mathbb{K}(L) ==$
  $\{ (b, m, j) \mid$
  $L_B \oplus L_J = L,$
  $L_B \neq [], L_J \neq [],$
  $b \in \mathbb{K}(L_B),$
  $j \in \mathbb{K}(L_J),$
  $m \in \{+, -, *, /\}$
  $\}$

Utolsó lépések: kifejezés kiértékelése, permutációk generálása

- Kifejezés kiértékelése

```
@spec ertek(k :: kif) -> e :: integer
# A k kifejezés értéke e
def ertek({bfa, mjel, jfa}) do
  op = fn '+' -> &+/2; '-' -> &-/2; '*' -> &*/2; '/' -> &//2 end
  op.(mjel).(ertek(bfa), ertek(jfa))
end
def ertek(i) -> i # A klózok sorrendje itt számít, fontos!
```

- Permutációk előállítása

```
@spec permutaciok(xs :: [any]) -> pss :: [[any]]
# Az xs lista elemeinek összes permutációját tartalmazó lista pss
```

- Példák:

```
iex> (&+/2).(1, 3)
4
iex> Dp.Kif.ertek({ {1, '-', { 3, '*', 4 }}, '+' , 6})
-5
iex> Dp.Kif.ertek({1,'/',0})
** (ArithmeticError) bad argument in arithmetic expression: 1 / 0...
iex> Dp.Kif.permutaciok([1,3,4])
[[1,3,4], [1,4,3], [3,1,4], [3,4,1], [4,1,3], [4,3,1]]
```


Adott értékű kifejezések felsorolása – teljes kód 1

`Dp.Kif.megoldasok([1,3,4,6], 24)`

```
@type muv :: '+' | '-' | '*' | '/'
@type kif :: integer | {kif, muv, kif}

@spec megoldasok(xs :: [integer], e :: integer) :: ks :: [kif]
# Az xs számokkal az e eredményt adó kifejezések listája ks
def megoldasok(xs, ered), do:
  for pss <- permutaciok(xs), # generátor
      kif <- kifek(pss),      # generátor
      (try do ertek(kif) catch _,_ -> nil end) == ered, # feltétel
    do: kif

@spec kifek(xs :: [integer]) :: ks :: [kif]
# Az xs listában lévő számokból alapműveletekkel előálló
# összes lehetséges kifejezés listája ks
def kifek([x]), do: [x]
def kifek(xs),  do:
  for xbs, xjs <- kettevagasok(xs),
      b          <- kifek(xbs),
      j          <- kifek(xjs),
      m          <- ['+', '-', '*', '/'],
    do: b, m, j
```


Adott értékű kifejezések felsorolása – teljes kód 2

`Dp.Kif.megoldasok([1,3,4,6], 24)`

```
@spec ertek(k :: kif) :: e :: integer
# A k kifejezés értéke e
def ertek(bfa, mjel, jfa) do
  op = fn '+' -> &+/2; '-' -> &-/2; '*' -> &*/2; '/' -> &//2 end
  op.(mjel).(ertek(bfa), ertek(jfa))
  end
def ertek(i), do: i    # A klózok sorrendje itt számít, fontos!

@spec kettevagasok(xs :: [integer]) :: \
                      pls :: [bls :: [integer], jls :: [integer]]
# Az összes olyan nemüres bls és jls listákból álló párok listája
# pls, amelyek páronként összefűzve kiadják az xs listát
def kettevagasok(xs), do:
  for i          <- 1..((length xs) - 1),
      {bls, jls} <- [Enum.split(xs, i)],
    do: {bls, jls}

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

II. rész

Elixir alapok

Tartalom

Funkcionális programozás, nyelvek

Hallott már a funkcionális programozásról?

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

Mi jut a Google „eszébe”, ha a *books* és *functional programming* szavakra keresek rá? [1]

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

Könyvek a funkcionális programozásról 1

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

Könyvek a funkcionális programozásról 2

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

Könyvek a funkcionális programozásról 3

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

Funkcionális programozási nyelvek, nyelvcsaládok

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

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

Funkcionális programozás (FP): mi az?

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

Funkcionális programozási szemlélet

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

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

Az Erlang nyelv

- 1985: megszületik „Ellemtelben” (Ericsson–Televerket labor)
  - A név eredete: A. K. Erlang dán matematikus, ill. Ericsson language
  - 1985-86: első interpreter Prologban! **(Joe Armstrong)**
- 1991: első megvalósítás, első projektek
- 1997: első OTP (Open Telecom Platform) + **BEAM virtuális gép** (B's – Bogdan's, Björn's – Erlang Abstract Machine)
- 1998-tól: nyílt forráskódú, szabadon használható `http://www.erlang.org/`
- Funkcionális alapú (functionally based)
- 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)
  - Beépitett eszközkészlet támogatja a fejlesztést

Elixir-szakirodalom

Könyvek:

- Dave Thomas: *Programming Elixir* $\geq$ *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*, 2019. `https://www.manning.com/books/elixir-in-action-second-edition`

További olvasnivalók:

- Elixir Home: `https://elixir-lang.org/`
- 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 (gyakorlatok): `https://exercism.org/tracks/elixir`

Elixir: telepítés, használat

- Elixir Install: `https://elixir-lang.org/install.html`
- Elixir with 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`

  A dokkerizált elixir használatára később mutatunk néhány példát.
- 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`

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

`dp_bev.ex` – Faktoriális

```
defmodule Dp.Bev 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 "dp_bev.ex"   # fordítás
[Dp.Bev]
iex(2)> Dp.Bev.fac(5)   # futtatás
120
iex(3)> fac(5)          # a modulnevet ki kell írni
** (CompileError) iex:3: undefined function fac/1
iex(4)> Dp.Bev.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
dp_bev.ex      dp_bev.exs      ...
iex(2)> c "dp_bev.ex"
[Dp.Bev]
iex(3)> exports Dp.Bev
fac/1
4> Dp.Bev.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
dp_bev.ex      dp_bev.exs
# iex dp_bev.ex
Erlang/OTP ...
Interactive Elixir ...
iex(1)> Dp.Bev.fac 5
120
iex(2)> Ctrl+C kétszer
$root@...:/home# ^D exit
$
```


Listakezelés – rövid példák 1

```
iex(1)> xs = [10,20,30] # érték kötése változóhoz (mintaillesztéssel)
[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]} # más érték kötése változóhoz
{[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' ('pin' operátor)
** (MatchError) no match of right hand side value: '\a\b\t'[2]
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([])
```

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

Listakezelés – rövid példák 2

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

Listakezelés – rövid példák 3

dp_bev.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 "dp_bev.ex"
warning: redefining module Dp.Bev (current version defined in memory)
  dp_bev.ex:1
[Dp.Bev]
iex(16)> xs = [10, 20.5, 30.5]
[10, 20.5, 30.5]
iex(17)> Dp.Bev.sum xs
61.0
iex(18)> Dp.Bev.sum tl xs
51.0
iex(19)> Dp.Bev.sum(tl(tl(tl xs)))
0
iex(20)> Dp.Bev.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")
    dp_bev.ex:13: Dp.Bev.sum/1
iex(21)> Dp.Bev.sum 'abc'  # 'abc' === [97, 98, 99]: 'abc' karakterkódok listája
294
```


Listakezelés – rövid példák 4

dp_bev.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 "dp_bev.ex"
[Dp.Bev]
iex(23)> xs
[10, 20.5, 30.5]
iex(24)> Dp.Bev.append(xs, [:a,:b,:c,:d])
[10, 20.5, 30.5, :a, :b, :c, :d]
iex(25)> Dp.Bev.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.

Tartalom

Típusok

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[3]	*Range*
Reguláris kifejezés	*Regular expression (regex)*
Sztring	*string*

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

A felsorolás nem teljes.

[3] A dőlt betűs típusok más alaptípusokra épülnek.

Szám

- Egész (integer)
  - Decimális, pl. 1234
  - Hexadecimális, pl. 0xcafe
  - Oktális, pl. 0o765
  - Bináris, pl. 0b1010
  - Tagolható, pl. `123_456_789`
  - Korlátlan pontosságú, pl. `12345678901234567890123456789`
  - 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[4]

[4] 64 bit, kb. 16 számjegy, max. exponens kb. 10^{308}

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`

Függvény (Function) 1 (fájl: dp_bev.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 = &Dp.Bev.fac/1 # &: capture operator
&Dp.Bev.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: dp_gen.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> Dp.Gen.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: dp_gen.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> Dp.Gen.sum_of_sqrs_b5 3, 4.5
29.25
iex> Dp.Gen.sum_of_sqrs_b5 3
34
def sum_of_sqrs_b5(a \\ 6, b \\ 5), do: sqr(a) + sqr(b)
iex> Dp.Gen.sum_of_sqrs_a6b5 3
34
iex> Dp.Gen.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} = # kötések mintaillesztéssel
  {&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[5] definiálására, használatára:

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

[5] 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']
```


Kulcs-érték lista (Keyword lists)

- Egy kulcs-érték párt kételemű enneskent írhatunk le: {:key, :value}
- 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", város: "Prága", szerelme: "jazz-zongorista"]
[név: "Szöszi", város: "Prága", szerelme: "jazz-zongorista"]
iex> inspect név: "Szöszi", szerelme: "jazz-zongorista", város: "Prága"[6]
"[név: \"Szöszi\", város: \"Prága\", szerelme: \"jazz-zongorista\"]"
iex> [:cseh_film, név: "Szöszi", város: "Prága", szerelme: "zongorista"]
[:cseh_film, {név: "Szöszi", szerelme: "zongorista", város: "Prága"}]
iex> {:cseh_film, név: "Szöszi", szerelme: "zongorista", város: "Prága"}
{:cseh_film, [név: "Szöszi", szerelme: "zongorista", város: "Prága"]}
```

- A kulcs-érték párokat szótárként is tárolhatjuk, ezzel később foglalkozunk

[6] inspect == Kernel.inspect =/= IO.inspect

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]
'erl'
iex> [31,101,114,108]
[31, 101, 114, 108]
iex> [a,101,114,108]# szabad változó tilos tömör kif-ben
** (CompileError) iex:254: undefined function a/0
iex> [:a,101,114,108]
[:a, 101, 114, 108]
iex> 'erl' ++ 'ang' # konkatenálható
'erlang'
```

- A karakterlánc NEM sztring!

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 később lesz még 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* (`"...#{<expr>}..."`):

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

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

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" => "Ukrajna", "SK" => "Szlovákia", "AT" => "Ausztria"}
%{"AT" => "Ausztria", "SK" => "Szlovákia", "UA" => "Ukrajna"}
iex> msgs = %{{:error,:enoent} => :fatal, {:error,:busy} => :retry}
%{{:error, :busy} => :retry, {:error, :enoent} => :fatal}
iex> colors =
  %{:red => 0xff0000, :green => 0x00ff00, :blue => 0x0000ff}
%{blue: 255, green: 65280, red: 16711680}
iex> colors = %{red: 0xff0000, green: 0x00ff00, blue: 0x0000ff}
%{blue: 255, green: 65280, red: 16711680}
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"]
"Ukrajna"
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}`[7] vagy `~r{regexp}options`
- A reguláris kifejezés szintaxisa a PCRE[8] 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

[7] A `~r{...}` jelölés egy ún. *szigil*, azaz bűvös jelölés. Egy szigilben határolójelként a `{...}` helyett többnyire használható a `<...>`, `[...]`, `(...)`, `|...|`, `/.../`, `"..."` és `'...'` is. A szigilekről részletek a Kernel dokumentációjában találhatók.

[8] 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 $ 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 whitespace-eket 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"]
```

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[9])
    ```
    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
    ```

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

Azonosító (identifier)

- Kisbetűvel vagy aláhúzásjellel (_) kezdődő, betűket, számjegyeket[10] 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)

[10] UTF-8 kódolású betű, ill. decimális számjegy;
`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

- 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*[11]
- Változó hatásköre
  - A változók 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
```

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

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 aláhúzásjellel kezdődő változónév és maga az aláhúzásjel (_) is mindenre illeszkedik, de nem lehet hivatkozni rájuk
- 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)
- Mintaillesztéskor a minta szabad változói kötötté válnak
- A kötés **nem** értékadás!
- Függvényhíváskor az aktuális paramétereket *illesztjük* a formális paraméterekre
- Figyelem: a Prologban a funkcionális nyelvekkel ellentétben *kétirányú* mintaillesztés van, *egyesítés* a neve

Példák mintaillesztésre 1

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

Példák mintaillesztésre 2

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


Példák mintaillesztésre 3

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


Függvénydefiniálás mintaillesztéssel (fájl: dp_gen.ex)

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

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


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 ⇝
1.999999999999993
0.0000000000000001 + 1 == 1 ⇝ false
0.00000000000000001 + 1 == 1 ⇝ true
```

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

Logikai műveletek

- Tisztán logikai operátorok: `not`, `and`, `or`; infix helyzetűek, operandusaik és eredményük is `boolean` típusú
- Igazságértéket eredményül adó operátorok: `!`, `&&`, `||`; infix helyzetűek, eredményük `boolean` típusú, operandusaik típusa tetszőleges[12]
- Mind lusta kiértékelésű, ún. *short-circuit* műveletek, azaz ha már a bal operandus eldönti az eredményt, a jobb operandus kiértélésére nem kerül sor
- Példák:

```
iex> false and div(3,0) === 2
false
iex> true and div(3,0) === 2
** (ArithmeticError) bad argument in arithmetic expression: div(3, 0)
iex> true && rem(3,2)
1
iex> false && rem(3,2)
false
iex> nil && rem(3,2)
nil
```

[12] 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)`[13]
  - Sztring, bináris: `bit_size(string)`, `byte_size(string)`
  - Szótár: `map_size(map)`
  - Ennes: `tuple_size(tuple)`, `elem(tuple, index)`, `put_elem(tuple, index, value)`[14]
  - Lista: `length(list)`, `hd(list)`, `tl(list)`
- Az operátorok is BIF-ek a `Kernel`-ben, pl. `Kernel.*(3,4)`

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

[14] Megjegyzés: `0 ≤ index ≤ tuple_size(tuple)-1`

Egyéb alapfügvények (típusvizsgálat és - konverzió)

- 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)`,

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 szringműveleteket is összefoglaljuk később egy dián

Műveletek listákon 2

- Lista feje, farka, hossza: `hd(xs)`, `tl(xs)`, `length(xs)`[15]
- 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
```

[15] `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. 97. dia), pl. `List.to_string`, `Tuple.to_list`
- Van három tesztelő függvény is:
  - `improper?(list)` igaz, ha `list` nem valódi lista[16]
  - `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
```

[16] 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.

Műveletek sztringeken 1

Mint tudjuk, 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*[17]

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

[17] Az U+0302 Unicode karakter az ún. *Combining Circumflex Accent*. A *pdflatex* nem tudja megjeleníteni, ezért vannak pontok (.) a példában. Az IEx jól jeleníti meg – próbálja ki!

Műveletek sztringeken 2

- <> a konkatenálás jele (már találkoztunk vele): `"ál"<>"om"`
- Sztring első / utolsó grafémája, grafémáinak száma: `first(string)`, `last(string)`, `length(string)`
- Graféma a sztring `pos` pozíciójában: `at(string, pos)`
- Tartalmazza-e sztring a `patts` elemeit: `contains?(string, patts)`
- Sztring elejéről / végéről / mindkettőről levágja az UTF-8 whitespace-eket: `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"])
true
iex> String.contains?(str, ["ü","ty","n"])
true
iex> String.contains?(str, ["é","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:[18]: `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
```

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

Tartalom

Projektszervezés Elixirben: `mix` 1

Itt az ideje, hogy 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. `Dialyxir`, `Benchee`) utólag kell telepíteni, ha és amikor szükség van rájuk.
- Magának a fordítónak (`elixir`, `elixirc`, `iex`) is, a moduloknak is több verziója 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.
- Projektkezelésre a `mix`-et használják az Elixirhez. A `mix` az Elixir-csomag része.

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

Projektszervezés Elixirben: `mix` 2

Használjuk most a dokkeres Elixirt, indítsuk el a `/tmp` mappából:

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

  /home# 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.
```


Projektszervezés Elixirben: `mix` 3

Hozzunk létre egy új projektet egy új mappában. A projekt és a mappa neve legyen `dp`, a modulé *Dp* (ez lenne egyébként a neve akkor is, ha nem használnánk a `-module` opciót).

```
# mix new dp --module Dp
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/dp.ex
* creating test
* creating test/test_helper.exs
* creating test/dp_test.exs

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

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


Projektszervezés Elixirben: `mix` 4

Nézzük, mi van a `mix.exs` fájlban[19]:

```
# cd dp
dp# cat mix.exs

defmodule DP.MixProject do
  use Mix.Project
  def project do
    [
      app: :dp,
      version: "0.1.0",
      elixir: "~> 1.12",
      start_permanent: Mix.env() == :prod,
      deps: deps()
    ]
  end
  # Run "mix help compile.app" to learn about applications.
  def application do
    [
      extra_applications: [:logger]
    ]
  end
```

[19] Mi más lenne, ha nem Elixir-kód. :-)

Projektszervezés Elixirben: `mix` 5

```
 # 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 és egy privát 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.
- Két új modult fogunk használni, a `Dialyxir`-t és a `Benchee`-t:
  `https://github.com/jeremyjh/dialyxir`
  `https://github.com/bencheeorg/benchee`
- A következő dián a `mix.exs` fájlt láthatjuk ismét az új függőségekkel.

Projektszervezés Elixirben: `mix` 6

```
# Run "mix help deps" to learn about dependencies.
defp deps do
 [
  {:benchee, "~> 1.0", only: :dev},
  {:dialyxir, "~> 1.0", only: [:dev], 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![20]

```
# mix do deps.get, deps.compile

* creating /root/.mix/archives/hex-0.21.3
Resolving Hex dependencies...
Dependency resolution completed:
New:
  benchee 1.0.1
  deep_merge 1.0.0
  dialyxir 1.1.0
  erlex 0.2.6
...
Compiling ...
```

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

Projektszervezés Elixirben: `mix dialyzer`

- A projekt forrásfájljainak a `lib` mappában kell lenniük: másoljuk be ide a `dp_bev.ex` fájlt, rakjunk bele egy hibás specifikációt, és dializáljuk.

```
@spec sum(xs::[integer]) :: s::[integer]
def sum([]), do: 0
def sum(xs)  do x = hd(xs); rs = tl(xs); x + sum rs end
```

- Az első futtatás soká tart, mert a *dialyzer* egy csomó ún. PLT-fájlt telepít a modulokhoz tartozó típusszignatúrákkal (PLT = Persistent Lookup Table).
- A dializálás a `.beam` fájlt elemzi, ezért ha változott a forrásfájl, az elemzés előtt fordítás készül belőle.

```
# mix dialyzer
lib/dp_bev.ex:18:invalid_contract
The @spec for the function does not match the success typing of the fun...

Function:
Dp.Bev.sum/1

Success typing:
@spec sum(maybe_improper_list()) :: number()
```


Projektszervezés Elixirben: `mix compile`; `iex` with `mix`

- Fordítani a `mix compile`-lal lehet, a lefordított fájl az `_build/dev/lib/dp/ebin/` mappába kerül, esetünkben `Elixir.Dp.Bev.beam` néven.
- Az `iex`-nek az indításakor megadhatjuk a projektünk elérési útjait és függőségeit: `iex -S mix`
- Most már egyszerű elindítani a programunkat az `r` parancssal (korrektebben: az `r` helper funkcióval):

```
iex> r Dp.Bev
```

- Most már meghívhatjuk a `Dp.Bev` modulban definiált függvényeket, pl.

```
iex> Dp.Bev.sum [1,2,3,4,5]
15
```

- A modult záró `end` után lehetnek olyan függvényhívások, melyeket az `iex` a betöltésükkor kiértékel; az `IO.inspect` függvény ezek eredményét kiírja a képernyőre, pl. `IO.inspect(Dp.Bev.sum(Enum.to_list 1..5))`. Ha újraindítjuk a programot az `r` helper függvénnyel, a művelet eredménye megjelenik a képernyőn:

```
iex> r Dp.Bev
...
15
```


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, hogy az Elixirben *minden* kifejezés – az összes többi funkcionális nyelvhez hasonlóan
- A kifejezés lehet:
  - Összetett kifejezés: tetszőleges adatstruktúra, elvégzendő műveletek/függvényhívások is lehetnek benne – sok ilyen példát láttunk
  - Term: mint az összetett kifejezés, de tömörnek, azaz 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
  - Feltételes kifejezés – ez is új, később tárgyaljuk

Kifejezés kiértékelése

- Funcionális nyelvek esetében mindig *kiértékelésről* beszélünk, sohasem *végrehajtásról*; az utóbbi használatát meghagyjuk az imperatív nyelveknek
- A kifejezés *kiértékelése* alapvetően *mohó* az Elixirben (eager, strict evaluation)
- Vannak kivételek, azaz vannak *lusta* kiértékelésű kifejezések (lazy, non-strict evaluation), pl. kétoperandusú logikai műveletek, ill. teljes modulok, pl. `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)
** (ArithmeticError) bad argument in arithmetic expression: 1 / 0 ...
iex> nevezo == 0 && ( 1 / nevezo ) < 1
** (ArithmeticError) bad argument in arithmetic expression: 1 / 0 ...
```


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”, amire az IEx figyelmeztet
- 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=7; x+x)]
[6]
iex> "x értéke"; x
warning: code block contains unused literal "x értéke" ...
iex> IO.write "x értéke "; x
x értéke 7
```


Kifejezések csoportosítása

- A szekvenciális kifejezés egyfajta csoportosítást jelent
- A `do ...end` blokk a csoportosítás egy másik változata
- `do ...end` blokkot kell használnunk pl. függvénydefinícióban, feltételes kifejezésben, komprehenzióban (jelölőben)
- 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: dp_gen.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}?"
  end
```

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: dp_gen.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> Dp.Gen.fac_2 -1
** (FunctionClauseError) no function clause matching in ...
```


Tartalom

Jelölő (Comprehension) 1

- Számos példát láttunk már eddig is a listajelölő[21] használatára. Most vegyünk sorra mindent, amit a jelölőről tudni kell az Elixirben
- Jelölő: `for` q_1[, q_2, ..., q_n][, `into:` coll], `do:` exp, ahol
  - A szögletes zárójelek itt azt jelentik, hogy: opcionális
  - A q_i vagy `pattern <- list` alakú *generátor*, vagy *predikátum* (igazságérték-eredményű függvény)
  - A jelölőben legalább egy generátornak lennie kell
  - A `pattern` minta a `list` lista olyan eleme, amely kielégíti az adott `pattern <- list` generátortól jobbra álló összes q_i *predikátumot*, azaz feltételt
  - Az `exp` tetszőleges, a `pattern` mintától függő vagy tőle független kifejezés

[21] Ebben a tantárgyban 2020-ig *listanézetnek* neveztük. Ez félrevezető, mert nem speciális nézetről, inkább speciális jelölésről van szó, ami – mint a kedvcsináló feladványban láttuk – a halmazelméletből került át a programozási nyelvekbe (set comprehension, set-builder notation). Mostantól jelölőnek vagy – az angol megnevezést átvéve – komprehenziónak foguk nevezni.

Jelölő (Comprehension) 2

- Jelölő: `for` q_1[, q_2, ..., q_n][, `into:` coll], `do:` exp, ahol
  - A generátorban a minta előállítására lista helyett más felsorolható kollekciót, leggyakrabban tartomány-típusú értéket is megadhatunk
  - Az opcionális `into:` kulcs után álló `coll`-lal megadhatjuk, hogy *milyen típusú* felsorolható kollekciót hozzon létre a jelölő; ha elhagyjuk, alapértelmezés szerint lista jön létre
  - A jelölő értéke az `exp` kifejezések értékének kollekciója
  - `coll`-ként üres kollekciót kell megadni, pl. `""` (sztring), `%{}` (szótár), `[]` (lista)
  - A jelölőben definiált változók lokálisak
- A komprehenzió ma már sokféle programozási nyelvben megtalálható, lásd: https://en.wikipedia.org/wiki/List_comprehension

Jelölő: kis példák

```
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}]
```


Jelölő: nagyobb példák (fájl: dp_gen.ex)

- Pitagoraszi számhármasok

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

- Hányszor fordul elő egy elem egy listában?

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


Gyorsrendezés (Quicksort) listajelölővel (fájl: dp_gen.ex)

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

Példák:

```
iex> Dp.Gen.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> Dp.Gen.qsort [:ab, :acb, :aca, :bca, :bbca, :bac, :abc, :a, :b, :c]
[:a, :ab, :abc, :aca, :acb, :b, :bac, :bbca, :bca, :c]
iex> Dp.Gen.qsort 'the quick brown fox jumps over the lazy dog'
'        abcdeeefghhijklmnoooopqrrsttuuvwxyz'
iex> Dp.Gen.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], 'ba', "ba"]
```

Permutáció listajelölővel (fájl: dp_gen.ex)

```
@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> Dp.Gen.perms [:a, :b]
[ [:a, :b], [:b, :a] ]
iex> Dp.Gen.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

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*
- 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

`if` és `unless` 1

- Az `if`-nek és az `unless`-nek két *paramétere* van: egy feltétel és egy kulcs-érték lista, amelyben két kulcs van: `do:` és `else:`
- Ha a feltétel igaz, az `if` a `do:` érték-párját, ha hamis, akkor az `else:` érték-párját adja eredményül; az `unless` pont fordítva csinálja
- Az `else:` ág, azaz az `{:else, érték}` pár elhagyható, ilyenkor a feltétel hamis volta esetén az `if` kiértékelése `nil`-t eredményez
- `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[22]
"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"
```

[22] Ez nem szép az Elixirtől: a funkcionális nyelvekben nem szokás valamelyik ág elhagyása.

if és unless 2

- Az if és az unless makrók[23]
- Az if-ben és az unless-ben a feltételben igazságérték-eredményű, egyébként tetszőleges függvények hívhatók meg
- 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 új 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
```

[23] Az Elixir a makrót fordításkor fejti ki.

cond (fájl: dp_cond.exs) 1

- A cond feltétel-érték párok sorozatát dolgozza fel
- A cond-ban – az if-hez és az unless-hez hasonlóan – a feltételben igazságérték-eredményű, egyébként tetszőleges függvények hívhatók meg
- 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 eredménye egy hexadecimális számjegy decimális értéke; kivételt dob, ha nem hexadec számjeggyel hívjuk meg

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


cond (fájl: dp_cond.exs) 2

- A kivételdobást elkerülhetjük a nil érték visszaadásával:

```
      ... ?a <= hexval and hexval <= ?f -> hexval - ?a + 10
          true                          -> nil ...
iex> to_decval.(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> to_decval.(?z)
:error
```

- A dp_cond.exs fájlban van a három változat to_decval, to_decval_2 és to_decval_3 néven
- Az exs arra utal, hogy a fájlban egy Elixir szkript van, amit az IEx nem fordít le, hanem közvetlenül interpretál
- Az IEx azonnal ki is értékeli a betöltött szkriptet, ha tehát látni akarjuk a függvények meghívásának eredményét, akkor ki is kell íratni őket

cond (fájl: dp_cond.exs) 3

- Mivel ennes is van az eredmények között, az IO.inspect/1-et célszerű használni, ami mindenféle adatstuktúrát ki tud írni

```
IO.inspect to_decval.(?f)
IO.inspect to_decval_2.(0)
IO.inspect to_decval_3.(?z)
```

- A fájlt Linuxon az iex dp_cond.exs paranccsal tölthetjük be, vagy ha már fut az IEx, akkor a c "dp_cond.exs"-szel

```
iex> c "dp_cond.exs"
15
nil
:error
[]
```

- dp_cond.exs-ben nincs moduldefiníció (bár lehetne), ezért marad üresen a betöltés után a modulnevek listája az utolsó sorban
- Az IEx shellből nem lehet hivatkozni egy modulnév nélküli szkriptben definiált dolgokra
- Ha lenne neve, ezeket akkor sem hívhatnánk meg, mert változókhoz kötöttünk névtelen függvényeket, változókat pedig nem lehet exportálni

case (fájl: dp_case.exs) 1

- A case adott értékre próbál meg mintákat illeszteni, az illesztés őrkifejezéssel finomítható

```
defmodule Dp.Case do
  def to_decval(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
  def test(0) do
    IO.inspect to_decval(?f)
    IO.inspect to_decval(?6)
    IO.inspect to_decval(?z)
  end
end
Dp.Case.test(0)
```


case (fájl: dp_case.exs) 2

- Betöltés után kiírja az előre definiált tesztek eredményét a fájl utolsó sorában lévő, a modult záró end utáni Dp.Case.test(0) sor hatására

```
iex> c "dp_case.exs"
...
15
6
nil
[Dp.Case]
```

- A Dp.Case.to_decval/1 függvényt nyilvánosként, nem privátként definiáltuk, ezért meg tudjuk hívni az IEx-ből

```
iex> Dp.Case.to_decval ?b
11
```

- Ismét hangsúlyozzuk, hogy bár vannak feltételes kifejezések az Elixirben (ahogy a többi funkcionális nyelvben is), célszerű helyettük, ahol csak lehet, mintaillesztést használni, esetleg őrökkel kiegészítve
- A mintaillesztést és esetleg őröket használó függvényklózokkal sokkal elegánsabban, átláthatóbban lehet megoldani az ilyen jellegű feladatok nagy részét

cond helyett case? (fájl: dp_case.exs)

- A kiértékelendő kifejezés kiválasztására
  - a case kifejezést illeszt mintákra, esetleg őrkifejezéssel finomítva az illesztést
  - a cond igazságérték-eredményű, egyébként tetszőleges függvényt elfogad
- cond helyett tehát csak akkor használható case, ha a választást őrökkel oldjuk meg
- Példa:

```
def to_decval_1(hexval) do
    case hexval do
      _ when ?0 <= hexval and hexval <= ?9 -> hexval - ?0
      _ when ?a <= hexval and hexval <= ?f -> hexval - ?a + 10
      _                                    -> nil
    end
end
iex> Dp.Case.to_decval_1(?c)
12
```

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

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

Névtelen függvény 1

- A névtelen függvény 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[24]. A klóztörzs tetszőleges szekvenciális kifejezés:

```
paraméterlista [when őrkifejezés]
```

- Példa (fájl: dp_fun.exs):

```
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
```

[24] A szögletes zárójelek itt az opcionálisat jelölik.

Névtelen függvény 2

- A második klóz törzsében egy szekvenciális kifejezés van, amit szabad, de a névtelen függvényben – ellentétben a neves függvénnyel, lásd alább – 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:
  - amikor egy függvénydefinícióban lokális függvényre van szükség,
  - amikor függvény-paraméterrel kell meghívni egy függvényt
- Példák (fájl: dp_fun.exs):

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


Névtelen függvény 3

- 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 (fájl: dp_fun.exs):

```
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 lehet rekurzív, azaz nem hívhatja önmagát.

Névtelen függvény 4

- A kis névtelen függvények használata annyira gyakori, hogy az Elixirben rövidítő jelölés van a definiálásukra
- Nézzük az előbbi két példát a rövidítő jelöléssel, továbbá a második egy változatát, amelyben a névtelen függvénynek három paramétere van:
- Példák (fájl: dp_fun.exs):

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

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

```
iex> sumtr_2 = &(&1.(&2) + &1.(&3))
iex> (&(&1.(&2) + &1.(&3))).(&(2 * &1 + 1), 3, 5)
18
```


Neves függvény (fájl: dp_fun.exs) 1

- Neves függvényt csak modulban lehet definiálni a `def` vagy a `defp` kulcsszóval
- A `defp`-vel definiált függvény privát (azaz lokális), csak a modul más függvényeiből hívható meg, a 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 modulszinten hivatkozunk egy függvényre, akkor mindig ki kell írni a modulnevet a függvénynév elé, akkor is, ha a függvény ugyanabban a modulban van definiálva
- A neves függvény definíciója – a névtelenhez hasonlóan – egy vagy több *klózból* áll
- A neves függvény definíciójában *minden* klóznak vagy a `def`, vagy a `defp` kulcsszóval[25] kell kezdődnie:

```
def klóz
def klóz ...
```

[25] A szintaxisleírásban `def`-et használunk, de `defp`-vel is ugyanaz a szintaxis.

Neves függvény (fájl: dp_fun.exs) 2

- A klóz *fejből* és *törzsből* áll, a klóztörzset általában vesszővel kell elválasztani a klózfejtől:

  klózfej`,` *klóztörzs*

- A klózfej *névből*, *paraméterlistából* – mintakifejezésből – és opcionálisan *őrkifejezésből* áll[26]. A paraméterlistát az egyértelműség kedvéért sokszor zárójelbe kell rakni[27]:

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

- A klóztörzs a `do:` kulcsszóval bevezetett pedig tetszőleges szekvenciális kifejezés[28]:

  `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

[26] A `[` és `]` ezen a dián is az opcionálisat jelölik, a `|` pedig az alternatívát.
[27] A *név* és a nyitó zárójel között nem lehet szóköz!
[28] A pontosvesszővel elválasztott vagy új sorba írt részkifejezéseket zárójelbe kell rakni.

Neves függvény (fájl: dp_fun.exs) 3

- A függvényt a neve és paramétereinek száma (az ún. aritása) azonosítja. Ugyanazzal a névvel több különböző aritású függvény definiálható
- A neves függvények lehetnek rekurzívak
- Már eddig is sok példát láttunk neves függvényekre, nézzünk még egyet
- **A feladat:** válogassuk ki egy `xs` listából a `d`-vel osztható egészeket, szorozzuk meg őket rendre `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 lista üres
  2. A soron következő elem `d`-vel osztható egész
  3. A soron következő elem nem egész vagy nem osztható `d`-vel

Neves függvény (fájl: dp_fun.exs) 4

- A három klózból álló privát függvény:

```
defp picked([], _d, _i), do: []
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)
```

- A 2. és 3. klóz sorrendje fontos, fordított sorrendben minden listaelemet eldobna, a `d`-vel osztható egészeket is
- A 2. klózban listát építünk: az eredménylista feje a soron következő elem és a számláló szorzata lesz, az erdeménylista farkát pedig úgy kapjuk meg, hogy a `picked` függvényt rekurzívan meghívjuk a soron következő elem utáni maradéklistára
- A 2. és a 3. klózban minden rekurzív lépésben eggyel rövidül a maradéklista. Mivel a lista korlátos, előbb-utóbb eljutunk az üres listáig, ami a rekurzió végét jelenti
- Az eredménylista nincs rendezve! Rendezzük csökkenő sorrendben:

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


Neves függvény (fájl: dp_fun.exs) 5

- 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 (fájl: dp_fun.exs) 6

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

```
defp filt([], _d), do: []
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)
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([], _i), do: []
defp mult([x|xs], i), do: [x * i | mult(xs, i+1)]
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 |> operátorral (fájl: dp_fun.exs) 7

- 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
- Az Elixirben van egy eszköz, amellyel elkerülhetjük az egymásba skatulyázást és az új változók bevezetését: ez a `|>` (pipe) operátor (a Unix/Linux `|`-vel jelöli, több más nyelven is van hasonló):

```
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 |> operátorral (fájl: dp_fun.exs) 8

- 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 – láttuk már, hogy függvényt is lehet paraméterként átadni –, akkor rugalmasabb lenne a megoldásunk.

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

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

```
defp filt([], _d), do: []
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)
```

- A 2. klózban a konkrét őrt cseréljük le függvényhívásra – a függvényt paraméterként adjuk át:

```
defp filt_2([], _f?), do: []
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?)
```

  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, `filt_2` és `oper` (fájl: dp_fun.exs) 10

- Őrkifejezés helyett tehát őr nélküli *feltételes kifejezést* kell használnunk, összevonva a korábbi 2. és 3. klózt:

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

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

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

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

  Ez sem elég általános; később megnézzük, lehet-e általánosabb megoldást írni ilyen jellegű feladatokra.
- A következő dián összerakjuk a teljes megoldást az új részmegoldásokból

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

- A teljes megoldás új verziója az új részmegoldásokból

```
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`

- 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 egyik előző példában látott `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 (fájl: dp_fun.exs)

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

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

- Működésük megértéséhez már nincs szükség magyarázatra[29]
- „Hivatalos” verziójuk az Elixirben: `Enum.map/2` és `Enum.filter/2`, tetszőleges felsorolható kollekcióra alkalmazhatók

[29] *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 129. dia). Ezt `map/2`-vel könnyű sztringek listájává konvertálni:

```
iex> (Dp.Gen.perms 'tér') |> (Enum.map &List.to_string/1)[30]
["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]
```

[30] `|>` túl nagy precedenciájú, ezért kellenek a zárójelek.

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:
  $((((1+3)+6)+10)+15)+21$, $(1+(3+(6+(10+(15+21)))))$
- Nem asszociatív művelet esetén a kétféle műveletvégzési sorrendnek különböző az eredménye:
  $((((1-3)-6)-10)-15)-21 == -54$,
  $1-(3-(6-(10-(15-21)))) == -12$
- A kétféle sorrendet megvalósító függvényt `foldl`-nek, ill. `foldr`-nek szokás hívni (a másodikat `reduce`-nak is); a specifikációjuk azonos:

  @spec fold(xs::[any], f::((x::any, y::any) -> r::any), acc::any) :: rs::[any]*

  `foldl` balról jobbra, `foldr` jobbról balra halad végig a listán. Az `f` kétoperandusú függvényt minden egyes lépésben a soron következő listaelemre és az `acc` akkumulátorra alkalmazzák.

`foldr/3` és `foldl/3` definíciója (fájl: dp_fun.exs) és használata

R

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

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

L

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

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

- 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)))
-2
iex> Dp.Fun.foldl([1,2,3,4], 0, &-/2) # 4-(3-(2-(1-0)))
2
```

- „Hivatalos” verziójuk az Elixirben: `List.foldr/3`, `List.foldl/3`, valamint felsorolható kollekciókra alkalmazható `Enum.reduce/3` és `Enum.reduce/2`

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

Példák listák összefűzésére

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

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 rekurzót

```
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 felfejtésével bajlódnunk

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 (fájl: `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 (fájl: `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 nem üres
def top(:empty), do: nil
def top({_s, x}), do: x
```


Kis példák a verem használatára 1 (fájl: `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 ügyesebben, tömörebben – érthetőbben (?)

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

Kis példák a verem használatára 2 (fájl: `lifo.ex`)

Fordítsunk meg egy karakterlistát!

1. lépés: a verembe betesszük az elemeket

```
iex> 'szöveg'
[115, 122, 246, 118, 101, 103]
iex> lifo = List.foldl('szöveg', Dp.Lifo.empty, \
             fn(c, a) -> Dp.Lifo.push(a, c) end)
{{{{{:empty, 115}, 122}, 246}, 118}, 101}, 103}
```

Tömörebben:

```
iex> lifo = '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 (fájl: `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 (fájl: `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 (fájl: 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 (fájl: dp_tree.ex)

- Legyen most ez a bináris fánk típusa:
  @type btree2 :: :lf | {any, btree2, btree2}
- Fa létrehozása

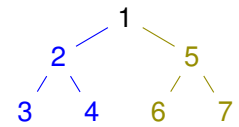
```
@spec empty() :: btr::(:lf)
# btr az egyetlen levélből álló üres fa
def empty(), do: :lf
@spec node(v: any, lt::btree2, rt::btree2) :: btr::btree2
# btr a v értékből, az lt és az rt fákból összerakott fa
node(v, lt, rt), do: {v, lt, rt}
```

- 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 (fájl: dp_tree.ex)

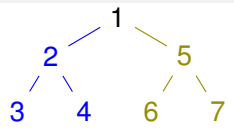
```
e=empty(), t=node(1, node(2, node(3,e,e),
                             node(4,e,e)),
                     node(5, node(6,e,e),
                             node(7,e,e)))
```




```
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 (fájl: 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 (fájl: 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))
```

```
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}}}
```


III. rész

Prolog alapok

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 az idén 50 éves Prolog = Programozás logikában (Programming in logic)
    - az elsőrendű logika egy erősen megszorított résznyelve 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 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 ⇒ Civakodó Henrik, B. Gizella ⇒ 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(Arg1, ..., Argn).` (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 :- cél1, ..., célk.      % klózfej ← cél1 ∧ ... ∧ célk
        % ^--kloztörzs--^
```

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

  `relációnév(Arg1, ..., Argn)`
- A „nagyszülője” kapcsolatot definiáló szabály:

```
% Gyerek nagyszülője Nagyszulo.
nsz(Gyerek, Nagyszulo) :-      % Gyerek nagyszülője Nagyszulo ha
      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?
| ?- nsz('Imre', NA), ffi(NA).      NA = 'Civakodó Henrik' ? ;
                                    NA = 'Géza' ? ; no
%    Ki Géza unokája?
| ?- nsz(U, 'Géza').                U = 'Imre' ? ; no
%    Ki Imre nagyszülője?
| ?- nsz('Imre', NSz).              NSz = 'Burgundi Gizella' ? ;
                                    NSz = 'Civakodó 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:
    $\forall Gy, NSz, Sz(\texttt{nsz}(Gy, NSz) \leftarrow \texttt{sz}(Gy, Sz) \wedge \texttt{sz}(Sz, NSz))$
  - Ekvivalens alak:
    $\forall Gy, NSz \quad (\texttt{nsz}(Gy, NSz) \leftarrow \exists Sz(\texttt{sz}(Gy, Sz) \wedge \texttt{sz}(Sz, NSz)))$
- 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

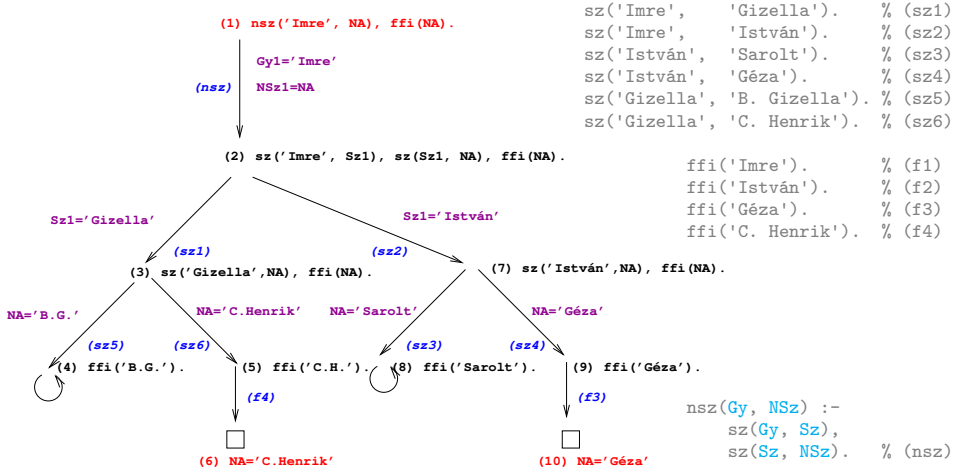
```
% sz(Gy, Sz): Gy szülője Sz.
  sz('Imre',    'Gizella').    % (sz1)
  sz('Imre',    'István').     % (sz2)
  sz('István',  'Sarolt').     % (sz3)
  sz('István',  'Géza').       % (sz4)
  sz('Gizella', 'B. Gizella'). % (sz5)
  sz('Gizella', 'C. Henrik'). % (sz6)

% 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 Prolog adatfogalma, a Prolog kifejezés (term)

- konstans (`atomic`)
  - 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: ⟨ struktúranév ⟩(⟨ $\arg_1$ ⟩, . . . , ⟨ $\arg_n$ ⟩)
    - a ⟨ struktúranév ⟩ egy névkonstans,
      az ⟨ $\arg_i$ ⟩ argumentumok tetszőleges Prolog kifejezések
    - a kifejezés funktora: ⟨ struktúranév ⟩/n
    - példák: `person(ian,smith,2003)`, `<(X,Y)`, `is(X, +(Y,1))`
  - szintaktikus „édesítőszerek”, pl. operátorok és listák:
    `X is Y+1` ≡ `is(X, +(Y,1))`, ill. `[1,2,3|X]` ≡ `.(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 ≡ 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)

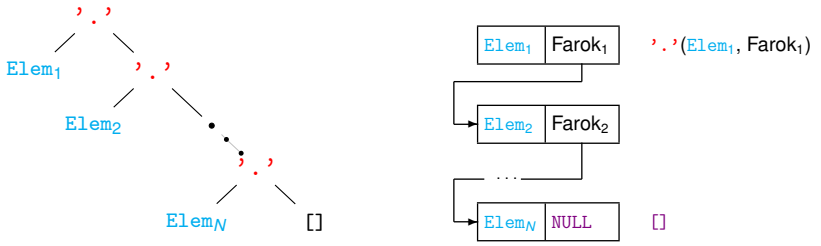
```
| ?- U+V = 1+(2*3).   ⟹   U = 1, V = 2*3
| ?- U-V = (8-4)-2.   ⟹   U = 8-4, V = 2
| ?- U+1 = 4+V.       ⟹   U = 4, V = 1 ?      % Kétirányú a mintaillesztés!
| ?- U+1 \= V+4.      ⟹   yes % szimbolikus kifejezések, a + nem kommutatív!
```

- Típusvizsgálatot végző beépített predikátumok
  - `var(X)`: `X` változó (`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:
  $[Elem_1|[...|[Elem_N|Farok]...]] \Longrightarrow [Elem_1,...,Elem_N|Farok]$
- Ha a farok `[]`, a „`|[]`” jelsorozat elhagyható:
  $[Elem_1,...,Elem_N|[]] \Longrightarrow [Elem_1,...,Elem_N]$
- 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
```

Aritmetikai beépített eljárások

- Egy aritmetikai kifejezés[31] (`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        % X-et nem értékeli ki!
| ?- X = 1, Y is (X-27) rem (X+2).    ⟹   X = 1, Y = -2 ? ; no
| ?- X = 3, Y is X/(X-1)*2*X+1.       ⟹   X = 3, Y = 10.0 ? ; 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
```

[31] 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 `fakt` hívások ne keveredjenek egyéb számításokkal:

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

- Az „`F = fakt1(N)`” ⟹ `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, 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(0, 2).  ⟹  no
| ?- fakt(4, 24). ⟹  yes                   | ?- fakt(N, 1).  ⟹  N = 0 ? ;
| ?- fakt(0, F).  ⟹  F = 1 ? ; no                                ! Inst.  err...
```


Programfejlesztési beépített predikátumok

- `consult(File)`: A `File` állományban levő programot beolvassa és értelmezendő alakban eltárolja. (`File` = `user` ⇒ 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:
  - levél esetén a levélben tárolt egész
  - csomópont esetén a két részfa levélösszegének összege

```
% 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);
   }
}
```

```
% 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 struktúranév után tilos formázó elemet tenni (operátorok miatt);
  - prefix operátor (ld. később) és „(” között kötelező a formázó elem;
  - klózt lezáró pont (.␣): önmagában álló pont (előtte nem tapadó jel áll) 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

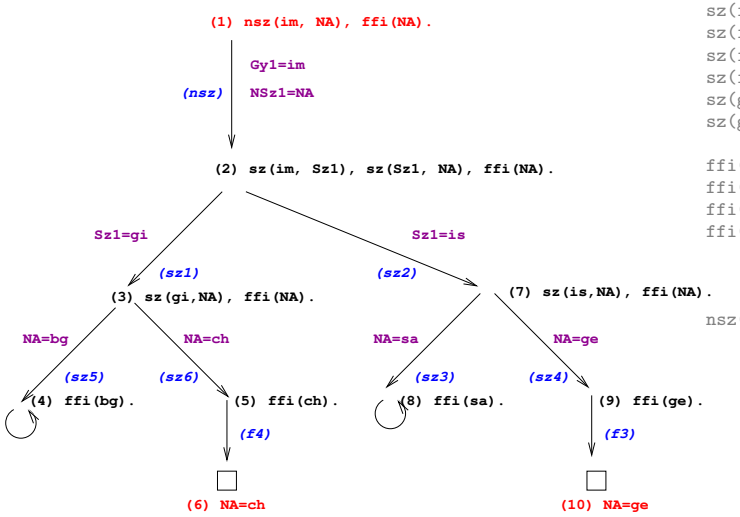
Elixir	**Prolog**
függvény, értéke tetsz. típusú	predikátum, azaz Boole-értékű függvény
arg. bemenő, a fv.érték kimenő	arg.-ok bemenők és kimenők is
egyetlen visszatérési érték	választási pontok, több megoldás lehet
külön ennes, lista típusok	a lista is összetett kifejezés
nincsenek felh. operátorok	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

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/V1-1`

A redukciós végrehajtás alapfogalmai (folyt.)

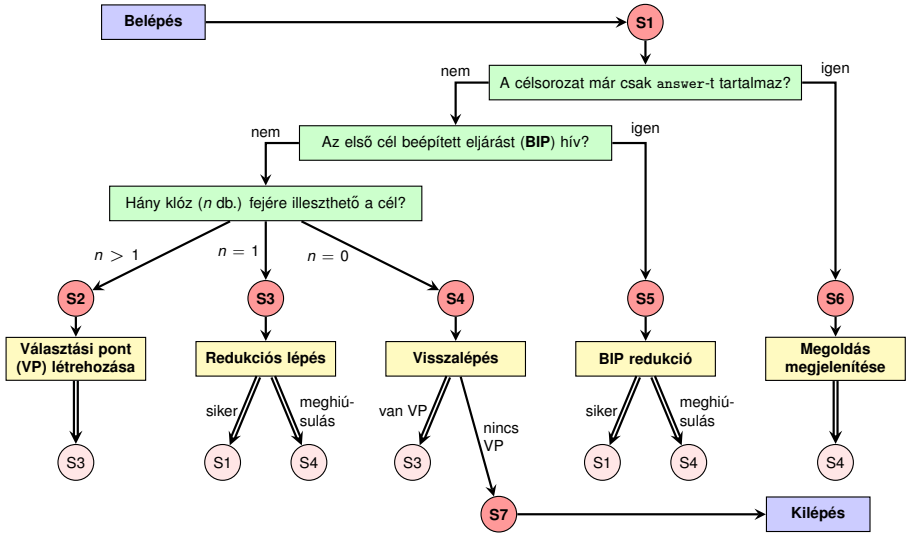
- A végrehajtás által használt (imperatív!) adatstruktúrák:
  - a jelenlegi célsorozatot tartalmazó változó (Goal)
  - a választási pontokat (VP) tartalmazó verem (Choice point stack)
- Például a `nsz(im, NA), ffi(NA), answer(NA)` célsorozat végrehajtásakor az alábbi VP verem jön létre:

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 ⇒ kimerítettük a keresési teret

A redukciós modell folyamatábrája



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

Megjegyzések a folyamatábrához

- Hétféle végrehajtási lépésünk van: **S1**–**S7**, ahol **S1** a kiindulási pont (de közbülső is), **S7** a végállapot.
- **S1** alapvető feladata az elágaztatás **S2**–**S6** 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).

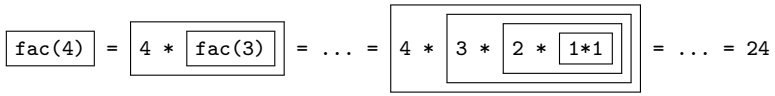
Tartalom

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 definicióját!

```
% F = N! (F az 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*.

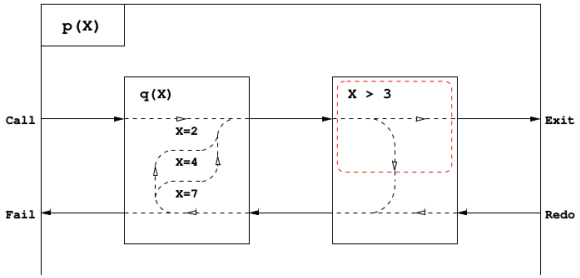
Prolog nyomkövetés eljárás-doboz modellel

- 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 ⇒ objektum):
  - eljárás meghívása (hívás kapu): objektum létrehozása
  - sikeres lefutás (kilépés kapu): változóbehelyettesítésekvisszaadása
  - sikertelen lefutás (meghiúsulás kapu): meghiúsulás jelzése
  - új megoldás kérése (újra kapu): következő 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.
```

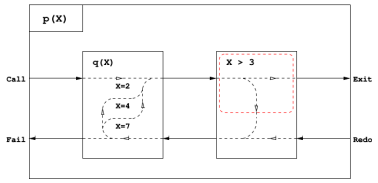


Előre: `Call p(X); Call q(X); Exit q(2); Call 2>3; Fail 2>3`
Vissza: `Redo q(2);`
Előre: `Exit q(4); Call 4>3; Exit 4>3; Exit p(4);` siker `X = 4 ? ;`
Vissza: `Redo p(4); Redo 4>3; Fail 4>3; Redo q(4);`
Előre: `Exit q(7); Call 7>3; Exit 7>3; Exit p(7);` siker `X = 7 ? ;`
Vissza: `Redo p(7); Redo 7>3; 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)

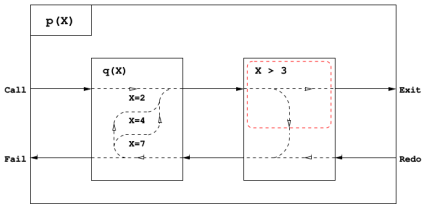
```
q(2).
q(4).
q(7).

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.)

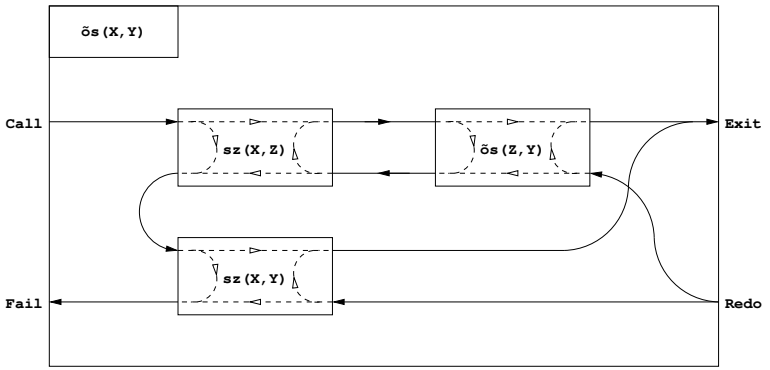
```
q(2).
q(4).
q(7).

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).` ⇒ `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, ∀ 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 ⇒ 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 (220. 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

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 kiküszöbölhetőek, ha `Vált` minden 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|_].        ⟹    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).  % app1/3 ≡ append/3 a fejlett Prologokban
```

- Egy `app1/3` eljáráshívás redukciós lépései:

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

Be- és kimenő argumentumok

- Az `append/3` predikátum az `append/2` Elixir függvény átírásával állt elő
- Ez a predikátum azonban használható más módon is, pl:

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

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

  `(+,+,+)`: ellenőrzés, pl. `append([1], [2], [1,2])` ⇒ `yes`

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

  `(+,-,+)`: adott prefixum ellenőrzése, pl. `append([1], L2, [1,2])` ⇒ `L2 = [2]`

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

  ...

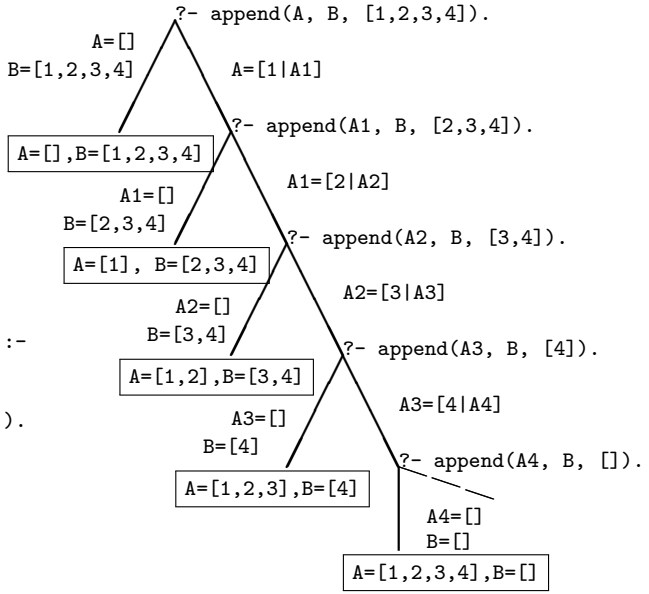
  `(-,-,+)`: lista szétszedése, pl. `append(L1, L2, [1,2])` ⇒ lásd fent

  `(-,-,-)`: ∞ keresés:, pl. `append(L1, L2, L3)` ⇒ `L1 = [], L3 = L2? ;`

  `L1 = [A], L3 = [A|L2]? ; L1 = [A,B], L3 = [A,B|L2]? ;...`

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).` ⟹ `L3 = [1,2|L2] ? ; no`
- A 2. arg.-ot nem bontjuk szét ⟹ 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 ∞ sok Prolog kifejezéssel fedhető le, pl.

  `_` ⊕ `[1] = L` (≡`L` utolsó eleme `1`): `L = [1]; [_,1]; [_,_,1]; ...`
- `app0` szétszedésre nem jó, mert pl. `app0(L, [1], [])` ⟹ ∞ ciklus, hiszen redukálva a 2. klózzal ⟹ `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)`: (`L1` ⊕ `L2`) ⊕ `L3` = `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).` ⟹ `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, Ret) :- Ret = R0.    % def revapp([], r0)     do r0 end
revapp([X|L0], R0, Ret) :-          % def revapp([x|l0], r0) do
    revapp(L0, [X|R0], Ret).        %         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 `lists` könyvtárban a fenti módon definiált select/3 eljárás keresési tere véges, ha vagy a 2., vagy a 3. argumentuma zárt végű lista.

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

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
```

    „Sz Imre szülője” 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 változók felsorolását jelöli.

```
| ?- 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 (‘;’ jobbról-balra zárójeleződik).
- Az `egyébként` rész elhagyható, alapértelmezése: `fail`.
- `\+ felt` átírható felt. 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 azonos a sima diszjunkciós alakkal (lásd komment), de annál hatékonyabb, mert nem hagy maga után választási pontot.
- Szám előjele

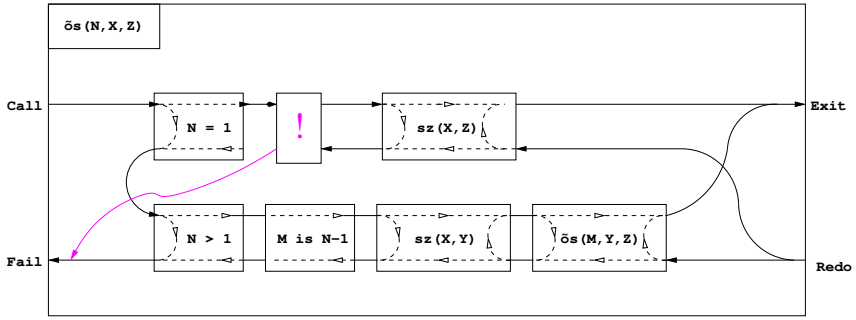
```
% Sign = sign(Num)
sign(Num, Sign) :-
        (   Num > 0 -> Sign = 1
        ;   Num < 0 -> Sign = -1
        ;   Sign = 0
        ).
```

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. a `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

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` op_1 `Y` op_2 `Z` zárójelezése, ahol op_1 és op_2 prioritása n_1 és n_2:
  - ha $n_1 > n_2$ akkor `X` op_1 `(Y` op_2 `Z)`;
  - ha $n_1 < n_2$ akkor `(X` op_1 `Y)` op_2 `Z`; (kisebb prio. ⇒ erősebb kötés)
  - ha $n_1 = n_2$ és op_1 jobb-asszociatív (`xfy`), akkor `X` op_1 `(Y` op_2 `Z)`;
  - egyébként, ha $n_1 = n_2$ és op_2 bal-assz. (`yfx`), akkor `(X` op_1 `Y)` op_2 `Z`;
  - egyébként szintaktikus hiba
- Érdekes példa: `:- op(500, xfy, +^).  % :- op(500, yfx, +).`

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

- A „vessző” jel többféle helyzetben is használható:
  - struktúra-argumentumokat, ill. listaelemeket határol el egymástól
  - 1000 prioritású `xfy` op. pl.: `(p:-a,b,c)`≡`:-(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
```

- Ugyanaz a névkonstans használható infix és prefix operátorként is, lásd pl. a beépített '`-`' operátort.

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!

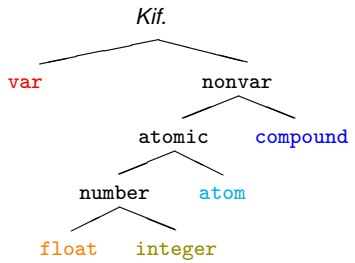
```
% deriv(Kif, D):  Kif-nek az x szerinti deriváltja D.
deriv(x, D) :-                  D = 1.
deriv(C, D) :-                  number(C), D = 0.
deriv(U+V, DU+DV) :-            deriv(U, DU), deriv(V, DV).
deriv(U-V, DU-DV) :-            deriv(U, DU), deriv(V, DV).
deriv(U*V, DU*V + U*DV) :-      deriv(U,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


Bev. példa újra – adott értékű kifejezések (kieg. anyag)

Idézzük fel a kurzus első előadásán ismertetett példát!

- Adott számokból megadott műveletek (pl. +, -, *, /) segítségével építsünk egy megadott értékű aritmetikai kifejezést!
  (Feltételezhető, hogy az adott számok mind különböznek.)
  - 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: $1 + 6 * (3 + 4)$, $(1 + 3)/4 + 6$
- Viszonylag nehéz megtalálni egy olyan aritmetikai kifejezést, amely az 1, 3, 4, 6 számokból áll, a négy alapműveletet használja és értéke 24

Adott értékű kifejezések – megoldás univ-val (kieg. anyag)

Írjunk egy, az alábbi fejkommentnek megfelelő eljárást:

```
% kif(+L, +MuvL, +Ertek, ?Kif): Kif egy olyan kifejezés, amely az L számlista
% elemeiből a MuvL listabeli műveletekkel épül fel, és amelynek értéke Ertek.
kif(L, MuvL, Ertek, Kif) :-
        permutation(L, PL),                  % lists könyvtárbeli eljárás
        levelek_muv_kif(PL, MuvL, Kif),
        catch(Kif =:= Ertek, _, fail).       % A 0-val való osztás kivédése

% A catch(+Cél,?Kiv,+KCél) beép. elj.: lefuttatja a Cél hívást. Ha a futás
% kivételt dob, akkor Kiv-et egyesíti ezzel a kivétellel, és KCél-t futtatja.

% levelek_muv_kif(+L, +MuvL, ?Kif): A MuvL listabeli műveletekkel felépített
%  Kif kifejezés leveleiben levő számok listája L.
levelek_muv_kif(L, _MuvL, Kif) :-
        L = [Kif], number(Kif).
levelek_muv_kif(L, MuvL, Kif) :-
        append(L1, L2, L), L1 \= [], L2 \= [],
        levelek_muv_kif(L1, MuvL, K1),
        levelek_muv_kif(L2, MuvL, K2),
        member(M, MuvL),
        Kif =.. [M,K1,K2].
```


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 ?
```

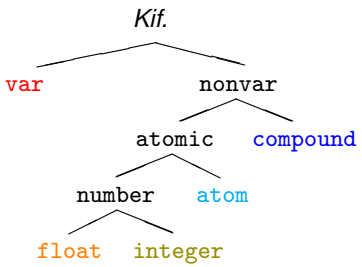
Számok szétszedése és összerakása

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

- A `number_codes(Szám, KódLista)` beépített eljárás végrehajtása:
  - Ha `Szám` adott (bemenő), és a $c_1c_2...c_n$ karakterekből áll, akkor `KódLista`-t egyesíti a $[k_1, k_2, ..., k_n]$ 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:

`var` $\prec$ `float` $\prec$ `integer` $\prec$ `atom` $\prec$ `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 (`abc`$\prec$`abcd`, `abcv`$\prec$`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. `p(x,s(u,v,w))` $\prec$ `a(b,c,d)`, vagy
  2. $n = m$, és $\texttt{név}_a \prec \texttt{név}_b$ (lexikografikusan), pl. `a(x,y)` $\prec$ `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. `r(1,u+v,3,x)` $\prec$ `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.

```
| ?- X = 1+2.     ⟹  X = 1+2
| ?- 3 = 1+2.     ⟹  no
```

- U `==` V: U azonos V-vel. Soha sem jelez hibát és soha sem helyettesít be.

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

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

```
| ?- X =:= 1+2.   ⟹  hiba
| ?- 1+2 =:= X.   ⟹  hiba
| ?- 2+1 =:= 1+2. ⟹  yes
| ?- 2.0 =:= 1+1. ⟹  yes
```

- U `is` V: U egyesítendő a V aritmetikai kifejezés értékével. Hiba, ha V nem (tömör) aritmetikai kifejezés.

```
| ?- 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
```

- (U `=..` V: U „szétszedettje” a V lista)

```
| ?- 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.

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

- 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` | `Cél` igaz }, `Lista` ≠ `[]`.

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` | (∃ `V1, ..., Vn`)`Cél` igaz } ≠ `[]`.

```
| ?- 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),`
  - emlékeztető: `sort(L, RL)` egy univerzális rendező eljárás, amely az `L` listát `@<` szerint rendezi az az azonos elemek kiszűrésével, 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

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.
négyzetei(L, Nk) :-
    findall(Y, (member(X, L), négyzete(X, Y)), Nk).
négyzete(X, Y) :- Y is X*X.
| ?- négyzetei([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

Részlegesen paraméterezett eljáráshívások – segédeszközök

- A `négyzetei/2` eljárás az Elixirből ismert `map/2` speciális esete.
- Prologban ennek a `maplist/3` eljárás felel meg:

```
négyzetei(Xs, Ys) :- maplist0(négyzete, Xs, Ys).
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` eljárások SICStus 4-ben már beépítettek, SICStus 3-ban még definiálni kellett ezeket, pl. így:

```
% Pr kiegészítve egy A utolsó argumentummal igaz.
call(Pr, A) :-
    Pr =.. PAs0, append(PAs0, [A], PAs1), Pr1 =.. PAs1, call(Pr1).
% Pr kiegészítve az A és B utolsó argumentumokkal igaz.
call(Pr, A, B) :-
    Pr =.. PAs0, append(PAs0, [A,B], PAs2), Pr2 =.. PAs2, call(Pr2).
...
```


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`.
- SWI Prologban a `maplist` hívásokból segédeljárás generálódik.

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
```


IV. rész

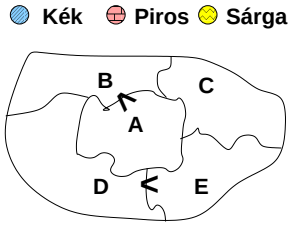
Haladó Prolog

Tartalom

Háttér: CSP (Constraint Satisfaction Problems)

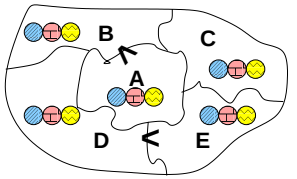
Példafeladat

Egy térkép kiszínezése kék, piros és sárga színekkel úgy, hogy a szomszédos országok különböző színűek legyenek, és ha két ország határán a < jel van, akkor a két szín ábécé-rendben a megadott módon kövesse egymást.



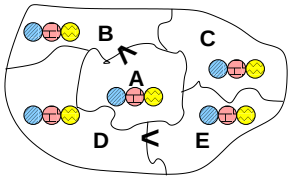
Egy lehetséges megoldási folyamat (*zárójelben a CSP elnevezések*)

1. Minden mezőben elhelyezzük a három lehetséges színt (*változók és tartományaik felvétele*).

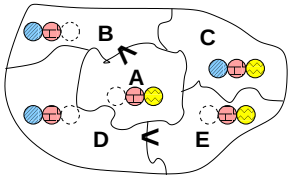


CSP példafeladat, folyt.

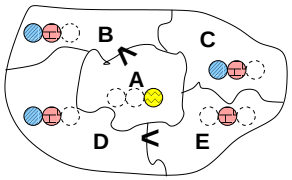
1. Minden mezőben elhelyezzük a három lehetséges színt (*változók és tartományaik felvétele*).



2. Az „A” mező nem lehet kék, mert annál „B” nem lehetne kisebb. A „B” nem lehet sárga, mert annál „A” nem lehetne nagyobb. Az „E” és „D” mezők hasonlóan szűkíthetők (*szűkítés, él-konzisztencia biztosítása*).

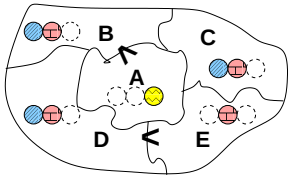


3. Ha az „A” mező piros lenne, akkor mind „B”, mind „D” kék lenne, ami ellentmondás (*globális korlát, ill. borotválási technika*). Tehát „A” sárga. Emiatt a vele szomszédos „C” és „E” nem lehet sárga (*él-konzitens szűkítés*).

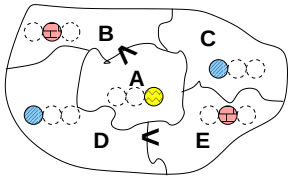


CSP példafeladat, folyt. 2

3. . . . Emiatt a vele szomszédos „C” és „E” nem lehet sárga (*él-konzitens szűkítés*)..



4. Mivel „E” piros, „C” és „D” nem lehet az, tehát mindkettő kék. Mivel „D” kék, „B” csak piros lehet (*él-konzitens szűkítés*).



Tehát az egyetlen megoldás:
A = sárga, B = piros, C = kék, D = kék, E = piros.

A CSP fogalma

- CSP = (X, D, C)
  - $X = \langle x_1, \ldots, x_n \rangle$ — változók
  - $D = \langle D_1, \ldots, D_n \rangle$ — tartományok, azaz nem üres halmazok
  - x_i változó a D_i véges halmazból (x_i tartománya) vehet fel értéket
  - C a problémában szereplő korlátok (atomi relációk) halmaza, argumentumaik X változói (például $C \ni c = r(x_1, x_3)$, $r \subseteq D_1 \times D_3$)
- A CSP feladat megoldása: minden x_i változóhoz egy $v_i \in D_i$ értéket kell rendelni úgy, hogy minden $c \in C$ korlátot egyidejűleg kielégítsünk.
- A példafeladatban:
  - Változók: A, B, C, D, E.
  - Kiindulási tartományok: mind az 5 változó tartománya {k, p, s}
  - Végső tartományok: A = {s}, B = {p}, C = {k}, D = {k}, E = {p}.
  - Korlátok (constraints, kényszerek):
    B < A, D < E, D $\neq$ B, B $\neq$ C, C $\neq$ E, D $\neq$ A, C $\neq$ A.
    (B < A írható így is: kisebb(B, A))

A CSP fogalma, folyt.

- **Definíció:** egy c korlát egy x_i változójának d_i értéke *felesleges,* ha nincs a c többi változójának olyan értékrendszere, amely d_i-vel együtt kielégíti c-t.
- Pl: legyen A < B, A tartománya {2,3,4,5}, B tartománya {1,2,3,4}
  - A-nak felesleges a 4 és 5 értéke,
  - B-nek felesleges az 1 és 2 értéke,
- **Állítás:** *felesleges érték elhagyásával (szűkítés) ekvivalens CSP-t kapunk.*
- **Definíció:** egy korlát *él-konzisztens* (arc consistent), ha egyik változójának tartományában sincs felesleges érték. A CSP *él-konzisztens*, ha minden korlátja él-konzisztens. Az él-konzisztencia szűkítéssel biztosítható.
- Ha minden reláció bináris, a CSP probléma gráffal ábrázolható (változó $\Rightarrow$ csomópont, reláció $\Rightarrow$ él). Az *él*-konzisztencia elnevezés ebből fakad.

A CSP megoldás folyamata

- felvesszük a változók tartományait;
- felvesszük a korlátokat mint démonokat, amelyek szűkítéssel él-konzisztenciát biztosítanak;
- többértelműség esetén címkézést (labeling) végzünk:
  - kiválasztunk egy változót (pl. a legkisebb tartományút),
  - a tartományt két vagy több részre osztjuk (választási pont),
  - az egyes választásokat visszalépéses kereséssel bejárjuk(egy tartomány üresre szűkülése váltja ki a visszalépést).

A térképszínezés mint CSP feladat

Modellezés (leképezés CSP-re)

- változók meghatározása: országonként egy változó, amely az ország színét jelenti;
- változóértékek kódolása: kék $\rightarrow$ 1, piros $\rightarrow$ 2, sárga $\rightarrow$ 3 (sok CSP megvalósítás kiköti, hogy a tartományok elemei pl. nem-negatív egészek);
- korlátok meghatározása:
  - az előírt < relációk teljesülnek,
  - a többi szomszédos ország-pár különböző színű.

CSP és Prolog

- A CSP logikai programozáson alapuló megvalósítását legtöbbször CLPFD (Constraint Logic Programming on Finite Domains) néven illetik
- Sok Prolog rendszer kínál CSP megoldó könyvtárat, `clpfd` néven

```
| ?- use_module(library(clpfd)).
...
| ?- domain([A,B,C,D,E], 1, 3),
     A #> B, A #\= C, A #\= D, A #\= E, B #\= C, B #\= D, C #\= E, D #< E.
            A in 2..3, B in 1..2, C in 1..3, D in 1..2, E in 2..3 ? ; no

| ?- domain([A,B,C,D,E], 1, 3),
     A #> B, A #\= C, A #\= D, A #\= E, B #\= C, B #\= D, C #\= E, D #< E,
     labeling([], [A,B,C,D,E]).
            A = 3, B = 2, C = 1, D = 1, E = 2 ? ; no

| ?- domain([A,B,C,D,E], 1, 3),
     A #> B, A #\= D, B #\= C, B #\= D, D #< E,
%    A #\= C, A #\= E, C #\= E helyett:
     all_distinct([A,C,E]).      % Az "A, C, E különbözőek" korlát okos megvalósítása,
                                 % globális kombinatorikai korláttal
            A = 3, B = 2, C = 1, D = 1, E = 2 ? ; no
```


Hatékony programozás Prologban

- A keresési tér szűkítése
- Determinizmus és indexelés
- Jobbrekurzió, akkumulátorok
- Imperatív programok átírása Prologba

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. Erlang 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 ⇒ 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+Ln+Ln−1+...+L1).
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+L1+L2+...+Ln).
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 az imperatív (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 változó kezdőértéke;
    - `S` az összeg a `sum/3` lefutása után: a 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 */
}
```


V. rész

Haladó Elixir

Tartalom

A mohó kiértékelés lépései

Egy összetett kifejezést az Elixir, mint láttuk, balról jobbra haladva az alábbi lépésekben értékel ki mohó (eager, strict), más néven applikatív sorrendű (applicative order)) kiértékeléssel, az alábbi rekurzív kiértékelési szabály szerint:

1. először kiértékeli – balról jobbra haladva – az operátor*kifejezést* (más néven függvénykifejezést) és az aktuális paramétereit (más néven argumentumait),
2. ezután az operátor (függvény) törzsében a formális paraméterek összes előfordulását lecseréli az aktuális paraméterekre,
3. végül kiértékeli az operátor (függvény) törzsét.

Kivétel: a lusta kiértékelésű `and` és `&&`, `or` és `||` műveletek.

Függvényalkalmazás mohó kiértékelése

Nézzük például a következő egyszerű függvényeket:

```
defp sq(x), do: x * x
def sumsq(x, y), do: sq(x) + sq(y)
def f(a), do: sumsq(a+1, a*2)
```

Mohó kiértékelés esetén minden lépésben egy részkifejezést egy vele egyenértékű kifejezéssel helyettesítünk.

Pl. az `f(5)` mohó kiértékelésének ezek a lépései:[32]

`f(5)` → `sumsq(5+1, 5*2)` → `sumsq(6, 5*2)` → `sumsq(6, 10)` → `sq(6) + sq(10)` → `6*6 + sq(10)` → `36 + sq(10)` → `36 + 10*10` → `36 + 100` → `136`

- A függvényalkalmazás itt bemutatott *helyettesítési modellje*, az *egyenlők helyettesítése egyenlőkkel* (equals replaced by equals) segíti a függvényalkalmazás *jelentésének* megértését.
- Olyan esetekben alkalmazható, amikor egy függvény *jelentése független* a környezetétől (pl. ha minden mellékhatás ki van zárva).

[32] → a helyettesítést jelöli.

Függvényalkalmazás lusta kiértékelése

- Egy összetett kifejezés kiértékelésére más lehetőség is van:

  a kiértékelést addig halogatjuk, ameddig csak lehetséges.

  Ezt a kiértékelést *lusta* (lazy), más néven *szükség szerinti* (by need) vagy *normál sorrendű* (normal order) kiértékelésnek nevezzük
  Nézzük az előző függvénydefiníciókat:

  ```
  defp sq(x), do: x * x
  def sumsq(x, y), do: sq(x) + sq(y)
  def f(a), do: sumsq(a+1, a*2)
  ```

- Pl. az `f(5)` lusta kiértékelésének ezek a lépései:

  `f(5)` → `sumsq(5+1, 5*2)` → `sq(5+1) + sq(5*2)` → `(5+1)*(5+1) + (5*2)*(5*2)` → `6*(5+1) + (5*2)*(5*2)` → `6*6 + (5*2)*(5*2)` → `36 + (5*2)*(5*2)` → `36 + 10*(5*2)` → `36 + 10*10` → `36 + 100` → `136`

A mohó és a lusta kiértékelés összevetése

- Olyan esetekben, amelyekben az *egyenlők helyettesítése egyenlőkkel* modell alkalmazható, a kétféle kiértékelési sorrend azonos eredményt ad.
- Vegyük észre, hogy lusta (szükség szerinti) kiértékelésnél egyes részkifejezéseket néha többször is ki kell értékelni.
- A többszörös kiértékelést a lusta kiértékelésű nyelveknél (pl. Haskell, Alice) a jobb fordítóprogramok úgy kerülik el, hogy
  - az azonos részkifejezéseket megjelölik,
  - amikor egy részkifejezést először kiértékelnek, *az eredményét megjegyzik* (vö. memoization),
  - a többi előfordulásakor pedig ezt az eredményt veszik elő.

  E módszer kisebb hátránya a nyilvántartás szükségessége.

  Ma általában így működik a *lusta* kiértékelés.

Tartalom

Lusta farkú lista

Lusta kiértékelés

- Amikor egy függvényt definiálunk, még nem értékeljük ki.
- Ahhoz, hogy kiértékeljük, *meg kell hívni*.
- A függvény lusta kiértékelését használjuk fel a lusta lista létrehozásához.

Lusta (farkú) lista

- Idézzük föl a `lifo` típus definícióját, amivel valójában listát definiáltunk:
  @type lifo :: :empty | {lifo, any}
- A `lifo`-hoz hasonlóan definiálhatunk pl. `lazy_list` néven egy lusta listát:
  @type lazy_list :: nil | {any, (() -> lazy_list)}
- Ez a lista tehát azért lusta, mert a második tag (a farok) *kiértékelése* – a függvénydefiníció miatt – *késleltett* (vö. delayed evaluation).
- Ez a lista azonban csak *részben* lusta: a fej mindig kiértékelődik, a farok lusta. (Valóban lusta lista létrehozása még körülményesebb.)

Lusta farkú lista feje, farka (fájl: `dp_lazy.ex`)

- Lusta farkú lista feje

  @spec head(xs::lazy_list) :: x::any
  # Az xs lusta farkú lista feje x
  ```
  def head({x, _xs}), do: x
  ```
- Lusta farkú lista farka

  @spec tail(ys::lazy_list) :: xs::lazy_list
  # Az xs lusta farkú lista farka ys
  ```
  def tail({_x, xs}), do: xs.()
  ```
- Mindkét függvény egyetlen klózból áll!
- Az üres lusta listának nincs se feje, se farka, ezért nem kell hozzá klóz.

Lusta farkú lista építése: sorozat 1 (fájl: `dp_lazy.ex`)

- Végtelen számsorozat

  @spec infseq(n::integer) :: ys::lazy_list
  # Az n-nel kezdődő, egyesével növő egész számok lusta farkú listája ys
  ```
  def infseq(n), do: {n, fn() -> infseq(n+1) end}
  ```
- Példák:

```
iex> l1s = Dp.Lazy.infseq(0)
{0, #Function<4.23268542/0 in Dp.Lazy.infseq/1>}
iex> Dp.Lazy.tail(l1s)
{1, #Function<4.23268542/0 in Dp.Lazy.infseq/1>}
iex> Dp.Lazy.tail(Dp.Lazy.tail(l1s))
{2, #Function<4.23268542/0 in Dp.Lazy.infseq/1>}
```

Lusta farkú lista építése: sorozat 2 (fájl: `dp_lazy.ex`)

- Véges számsorozat

```
@spec seq(m::integer, n::integer) :: ys::lazy_list
# Az m-től n-ig egyesével növő egész számok lusta farkú listája ys
def seq(m, n) when m <= n, do: {m, fn() -> seq(m+1, n) end}
def seq(_m, _n),           do: nil
```

- Példák:

```
iex> Dp.Lazy.seq(1,1)
{1, #Function<8.23268542/0 in Dp.Lazy.seq/2>}
iex> Dp.Lazy.head(Dp.Lazy.seq(1,1))
1
iex> Dp.Lazy.tail(Dp.Lazy.seq(1,1))
nil
```


Lista konvertálása 1 (fájl: `dp_lazy.ex`)

- Elixir-listából lusta farkú lista
- Nagyon gyors: egyetlen függvényhívás!

```
@spec cons(ls::[any]) :: ys::lazy_list
# Az ls lista elemeiből álló lusta farkú lista ys
def cons([]),     do: nil
def cons([x|xs]), do: {x, fn() -> cons(xs) end}
```

- Példák

```
iex> l2s = Dp.Lazy.cons([1,2])
{1, #Function<1.23268542/0 in Dp.Lazy.cons/1>}
iex> t2s = elem(l2s, 1)
#Function<1.23268542/0 in Dp.Lazy.cons/1>
iex> t2s.()
{2, #Function<1.23268542/0 in Dp.Lazy.cons/1>}
iex> Dp.Lazy.tail(t2s.())
nil
```


Lista konvertálása 2 (fájl: `dp_lazy.ex`)

- Lusta farkú listából Elixir-lista
- Csak az első `n` elemét vesszük ki (ne feledjük, hogy a lusta farkú lista is lehet véges!)

```
@spec take(xs::lazy_list, n::integer) :: ls::[any]
# Az xs lusta farkú lista első n eleméből álló lista ls
def take(_, 0),      do: []
def take(nil, _),    do: []
def take({x,_}, 1),  do: [x]  # Így nem kell kiértékelni a lista farkát[33]
def take({x,xs}, n), do: [x | take(xs.(), n-1)]
```

- Példák

```
iex> Dp.Lazy.take(Dp.Lazy.infseq(0), 5)
[0, 1, 2, 3, 4]
iex> Dp.Lazy.take(Dp.Lazy.seq(1,2), 5)
[1, 2]
```

[33] Optimalizációs lépés; ez a klóz elhagyható.

Függvények lusta listára adaptálva: `sum` (fájl: `dp_lazy.ex`)

- Lusta farkú lista elemeinek összeadása (csak véges lusta farkú listákra!)

```
@spec sum(xs::lazy_list) :: n::number
# Az xs véges lusta farkú számlista elemeinek összege n
sum(xs), do: sum(xs, 0)
@spec sum(xs::lazy_list, a::number) :: n::number
# Az xs véges lusta farkú számlista elemeinek és az a-nak az összege n
def sum(nil, a),     do: a
def sum({x, xs}, a), do: sum(xs.(), a+x)
```

Függvények lusta listára adaptálva: `map` (fájl: `dp_lazy.ex`)

- Motiváció:
  1. listajelölő (komprehenzió) nem alkalmazható;
  2. lusta farkú lista szintaxisa ijesztő, nehézkes, ezért elrejtendő.
- `map/2` lusta farkú listára

```
@spec map(f::(() -> any), ys::lazy_list) :: zs::lazy_list
# Az ys elemeiből az f transzformációval előálló lusta farkú lista zs
def map(_, nil),    do: nil
def map(f, {y,ys}), do: {f.(y), fn() -> map(f, ys.()) end}
```

- Példák

```
iex> Dp.Lazy.take(Dp.Lazy.infseq(12), 12)
[12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
iex> mz = Dp.Lazy.map(&(rem(&1,3) === 0), Dp.Lazy.infseq(12))
{true, #Function<5.23268542/0 in Dp.Lazy.map/2>}
iex> Dp.Lazy.take(mz, 8)
[true, false, false, true, false, false, true, false]
```


Függvények lusta listára adaptálva: `filter` (fájl: `dp_lazy.ex`)

- `filter/2` lusta farkú listára

```
@spec filter(p::(() -> boolean), ys::lazy_list) :: zs::lazy_list
# Az ys p-vel megszűrt elemeiből álló lusta farkú lista a zs
# Kicsit mohó, az eredménylista fejéig kiértékeli a listát
def filter(_, nil),    do: nil
def filter(p, {y, ys}) do
  case p.(y) do
    true  -> {y, fn() -> filter(p, ys.()) end}
    false -> filter(p, ys.()) # Megkeressük az eredmény fejét
  end
end
```

- Példák

```
iex> Dp.Lazy.take(Dp.Lazy.infseq(4), 12)
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
iex> mz = Dp.Lazy.filter(&(rem(&1,3)===0), Dp.Lazy.infseq(4))
{6, #Function<3.83713946/0 in Dp.Lazy.filter/2>}
iex> Dp.Lazy.take(mz, 8)
[6, 9, 12, 15, 18, 21, 24, 27]
```


Függvények lusta listára adaptálva: `append` (fájl: `dp_lazy.ex`)

- `append/2` lusta farkú listára

```
@spec append(xs::lazy_list, ys::lazy_list) :: zs::lazy_list
# Az xs lusta farkú lista ys (ugyancsak lusta farkú lista) elé
# fűzésével előálló lusta farkú lista zs. Ha xs nem véges, az
# összefűzés felesleges, az ys elemei elérhetetlenek lesznek
def append(nil, ys),    do: ys
def append({x,xs}, ys), do: {x, fn() -> append(xs.(),ys) end}
```

- Példák

```
iex> mz = Dp.Lazy.append(Dp.Lazy.seq(3,6), Dp.Lazy.infseq(9))
{3, #Function<0.83713946/0 in Dp.Lazy.append/2>}
iex> Dp.Lazy.take(mz, 4)
[3, 4, 5, 6]
iex> Dp.Lazy.take(mz, 12)
[3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 15, 16]
iex> mz = Dp.Lazy.append(Dp.Lazy.infseq(9), Dp.Lazy.seq(3,6))
{9, #Function<0.83713946/0 in Dp.Lazy.append/2>}
iex> Dp.Lazy.take(mz, 12)
[9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```


A lusta farkú lista és a mohó lista összehasonlítása

- Tárigénye csak a kiértékelt résznek van
- Lusta farkú lista *teljes* kiértékelése sokkal lassúbb lehet (késleltetés)
- Az időigénye azonban *lehet kisebb is, ha* nem kell teljesen kiértékelni

Nevezetes számsorozatok 1 (fájl: dp_lazy.ex)

- Fibonacci-sorozat

```
@spec fibs(curr::integer, next::integer) :: ys::lazy_list
# ys egy olyan általánosított Fibonacci-sorozat (lusta
# farkú lista), amelynek első két tagja curr és next
def fibs(curr, next), do:
  {curr, fn() -> fibs(next, curr + next) end}
```

- Az n-edik Fibonacci-szám

```
@spec fib(n::integer) :: f::integer
# Az n-edik Fibonacci-szám f
def fib(n), do: nth(fibs(0,1), n)
```

- Példák

```
iex> Dp.Lazy.take(Dp.Lazy.fibs(0, 1), 10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
iex> f = &Dp.Lazy.fib/1; {f.(5), f.(6), f.(7), f.(8), f.(9)}
{5, 8, 13, 21, 34}
iex> Dp.Lazy.fib 100_000
259740693472217241661550340212759154148804853865176 96...
```


Nevezetes számsorozatok 2 (fájl: dp_lazy.ex)

- Eratosztenészi szita

```
@spec sift(prime::integer, ys::lazy_list) :: zs::lazy_list
# Az ys lusta farkú lista azon elemeinek listája a zs lusta farkú
# lista, melyek nem oszthatók prime-mal
def sift(prime, ys), do:
  filter(fn(n) -> rem(n, prime) !== 0 end, ys)

@spec sieve(ys::lazy_list) :: zs::lazy_list
# A zs lusta farkú lista az ys nem véges lusta farkú
# lista "szitáltja" (üres listára kivételt dob)
def sieve({x, xs}), do: {x, fn() -> sieve(sift(x,xs.())) end}
```

- Példák

```
iex> Dp.Lazy.take(Dp.Lazy.sieve(Dp.Lazy.infseq 2), 10)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
iex> Dp.Lazy.take(Dp.Lazy.sieve(Dp.Lazy.infseq 2),5000) \
  |> Enum.drop(4995)
[48563, 48571, 48589, 48593, 48611]
```


Lusta append alkalmazása: lusta qsort (fájl: dp_lazy.ex)

- Gyorsrendezés

```
@spec qsort(ys::lazy_list) :: zs::lazy_list
# Az ys lusta farkú lista rendezett változata zs
# Az append alkalmazása: ha csak a lista elejére van
# szükség, csak a lista elejét rendezi
def qsort(nil),       do: nil
def qsort({pivot, ys}) do
  low  = fn(x) -> x < pivot end
  high = fn(x) -> x >= pivot end
  append(
    qsort(filter(low, ys.())),
    {pivot, fn() -> qsort(filter(high, ys.())) end} )
end
```

- Példák

```
iex> Dp.Lazy.take(Dp.Lazy.qsort(Dp.Lazy.cons([5,3,6,8,1,7])), 2)
[1, 3]
iex> Dp.Lazy.take(Dp.Lazy.qsort(Dp.Lazy.cons([5,3,6,8,1,7])), 5)
[1, 3, 5, 6, 7]
```