LISTÁK PROLOGBAN

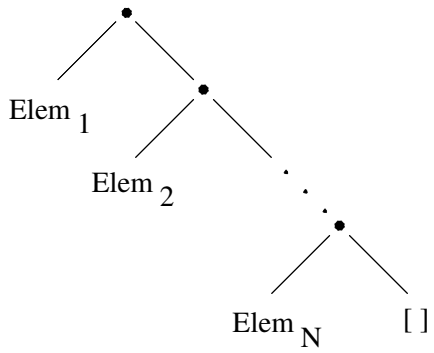
A Prolog lista-fogalma

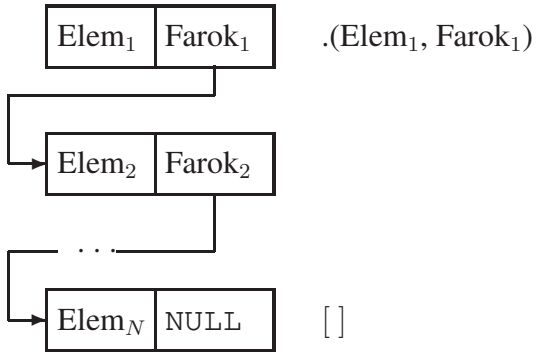
- A Prolog lista
  - Az üres lista a `[]` névkonstans. A nem-üres lista `'.'(Fej,Farok)` struktúra ahol
    - `Fej` a lista feje (első eleme), míg
    - `Farok` a lista farka, azaz a fennmaradó elemekből álló lista.
  - A listák írhatók egyszerűsített alakban („szintaktikus édesítés”).
  - Megvalósításuk optimalizált, időben és helyben is hatékonyabb, mint a „közönséges” struktúráké.
- Példa

```
számlista(.(E,L)) :-
    number(E), számlista(L).
számlista([]).

| ?- listing(számlista).
számlista([A|B]) :-
    number(A),
    számlista(B).
számlista([]).

| ?- számlista([1,2]).    %  [1,2] == .(1,.(2,[])) == [1|[2|[]]]
     yes
| ?- számlista([1,a,f(2)]).
     no
```

Listák írásmódjai

- Egy N elemű lista lehetséges írásmódjai:
  - alapstruktúra-alak: `.(Elem`$_1$`,.(Elem`$_2$`,...,.(Elem`$_N$`,[])...))`
  - ekvivalens lista-alak: `[Elem`$_1$`,Elem`$_2$`,...,Elem`$_N$`]`
  - kevésbe kényelmes ekvivalens alak: `[Elem`$_1$`|[Elem`$_2$`,...,[ Elem`$_N$`|[] ] ...]]`
- A listák fastruktúra alakja és megvalósítása





Listák jelölése — szintaktikus édesítőszerek

- az alapvető édesítés: `[Fej|Farok]` ≡ `.(Fej, Farok)`
- N-szeri alkalmazás kevesebb zárójellel:
  `[Elem`$_1$`,Elem`$_2$`,...,Elem`$_N$`|Farok]` ≡ `[Elem`$_1$`|[Elem`$_2$`,...,Elem`$_N$`|Farok]]`
- Ha a farok `[]`: `[Elem`$_1$`,Elem`$_2$`,...,Elem`$_N$`]` ≡ `[Elem`$_1$`,Elem`$_2$`,...,Elem`$_N$`|[]]`

```
| ?- [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] ?
| ?- [X|[3-Y/X|Y]]= .(A, [A-B,6]).  ⇒ A=3, B=[6]/3, X=3, Y=[6] ?
```

Tömör és minta-kifejezések, lista-minták, nyílt végű listák

- (Ismétlés:) Tömör (ground) kifejezés: változót nem tartalmazó kifejezés
- Minta: egy általában nem nem tömör kifejezés, mindazon kifejezéseket „képviseli”, amelyek belőle változó-behelyettesítéssel előállnak.
- Lista-minta: listát (is) képviselő minta.
- Nyílt végű lista: olyan lista-minta, amely bármilyen hosszú listát is képvisel.
- Zárt végű lista: olyan lista(-minta), amely egyféle hosszú listát képvisel.

Zárt végű	Milyen listákat képvisel
`[X]`	egyelemű
`[X,Y]`	kételemű
`[X,X]`	két egyforma elemből álló
`[X,1,Y]`	3 elemből áll, 2. eleme `1`

Nyílt végű	Milyen listákat képvisel
`X`	tetszőleges
`[X|Y]`	nem üres (legalább 1 elemű)
`[X,Y|Z]`	legalább 2 elemű
`[a,b|Z]`	legalább 2 elemű, elemei: `a, b, ...`

A logikai változó

- A logikai változó fogalma:
  - kifejezésként, kifejezésben egyaránt előfordulhat, vö. a változókat a (lista) mintákban.
  - két változó azonossá tehető (azaz egyesíthető): pl. két azonos változó egy kifejezésben.
  - a változó „teljes jogú” állampolgár a (rész)kifejezések világában
- SML-ben is van mintaillesztés, de a minta csak szétszedésre használható, összerakásra nem; a mintabeli változók mindig (tömör) értéket kapnak.
- (Egyes újabb funkcionális nyelvek, pl. az Oz nyelv, támogatják a logikai változókat.)
- Példa: Az alábbi célsorozat egy két **azonos** elemből álló listát épít fel az `L` változóban. Az elemek értéke **azonos** lesz a célsorozatbeli `X` változóval:

```
első_eleme([E|_], E).
második_eleme([_,E|_], E).
```

```
| ?- első_eleme(L, X), második_eleme(L, X). ⟹ L = [X,X|_A] ? ; no
```

- Ha az egyesített változók bármelyike értéket kap, a többi is erre az értékre helyettesítődik:

```
| ?- első_eleme(L, X), második_eleme(L, X), X = alma.
                                ⟹ X = alma, L = [alma,alma|_A] ? ; no
| ?- első_eleme(L, X), második_eleme(L, X), második_eleme(L, bor)
                                ⟹ X = bor, L = [bor,bor|_A] ? ; no
```

Listák összefűzése: az `append/3` eljárás

- `append(L1, L2, L3)`: Az `L3` lista az `L1` és `L2` listák elemeinek egymás után fűzésével áll elő (jelöljük: `L3` = `L1`⊕`L2`) — két megoldás:

```
append0([], L2, L) :- L = L2.
append0([X|L1], L2, L) :-
    append0(L1, L2, L3), L = [X|L3].
```

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

```
    > append0([1,2,3],[4],A)
(2) > append0([2,3],[4],B), A=[1|B]
(2) > append0([3],[4],C), B=[2|C], A=[1|B]
(2) > append0([],[4],D),C=[3|D],B=[2|C],A=[1|B]
(1) > D=[4], C=[3|D], B=[2|C], A=[1|B]
BIP > C=[3,4], B=[2|C], A=[1|B]
BIP > B=[2,3,4], A=[1|B]
BIP > A=[1,2,3,4]
BIP > []
L = [1,2,3,4] ?
```

```
    > append([1,2,3],[4],A), write(A)
(2) > append([2,3],[4],B), write([1|B])
(2) > append([3],[4],C), write([1,2|C])
(2) > append([],[4],D), write([1,2,3|D])
(1) > write([1,2,3,4])
[1,2,3,4]
BIP > []
L = [1,2,3,4] ?
```

- Az `append0/append(L1, ...)` komplexitása: futási ideje arányos `L1` hosszával.
- Miért jobb az `append/3` mint az `append0/3`?
  - `append/3` **jobbrekurzív**, ciklussal ekvivalens (nem fogyaszt vermet)
  - `append([1,...,1000],[0],[2,...])` azonnal, `append0(...)` 1000 lépésben hiúsul meg
  - `append/3` használható szétszedésre is (lásd később), míg `append0/3` nem.

Lista építése *elölről* — nyílt végű listákkal

- Az append eljárás már az első redukciónál felépíti az eredmény fejét!
  (az eredményparaméter egy lista-minta lesz, a farok még ismeretlen, vö. logikai változó)

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-     append(L1, L2, L3).
| ?- append([1,2,3], [4], Ered) ⟹  Ered = [1|A], append([2,3], [4], A)
```

- Haladó nyomkövetési lehetőségek ennek demonstrálására
  - `library(debugger_examples)` — példák a nyomkövető programozására, új parancsokra
  - új parancs: ‘`N` ⟨ név ⟩’ — fókuszált argumentum elnevezése
  - szabványos parancs: ‘`^` ⟨ argszám ⟩’ — adott argumentumra fókuszálás
  - új parancs: ‘`P` [⟨ név ⟩]’ — adott nevű (ill összes) kifejezés kiiratása

```
| ?- use_module(library(debugger_examples)).
| ?- trace, append([1,2,3],[4,5,6],A).
        1      1 Call: append([1,2,3],[4,5,6],_543) ? ^ 3
        1      1 Call: ^3 _543 ? N Ered
        1      1 Call: ^3 _543 ? P                      ⇒  Ered = _543
        2      2 Call: append([2,3],[4,5,6],_2700) ? P  ⇒  Ered = [1|_2700]
        3      3 Call: append([3],[4,5,6],_3625) ? P    ⇒  Ered = [1,2|_3625]
        4      4 Call: append([],[4,5,6],_4550) ? P     ⇒  Ered = [1,2,3|_4550]
        4      4 Exit: append([],[4,5,6],[4,5,6]) ? P   ⇒  Ered = [1,2,3,4,5,6]
        3      3 Exit: append([3],[4,5,6],[3,4,5,6]) ?
        2      2 Exit: append([2,3],[4,5,6],[2,3,4,5,6]) ?
        1      1 Exit: append([1,2,3],[4,5,6],[1,2,3,4,5,6]) ?
⇒  A = [1,2,3,4,5,6] ? ; no
```

Listák megfordítása

- Naív (négyzetes lépésszámú) megoldás

```
% nrev(L, R): Az R lista az L megfordítása.
nrev([], []).
nrev([X|L], R) :-
    nrev(L, RL),
    append(RL, [X], R).
```

- Lineáris lépésszámú megoldás

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

% revapp(L1, L2, R): L1 megfordítását L2 elé fűzve kapjuk R-t.
revapp([], R, R).
revapp([X|L1], L2, R) :-
    revapp(L1, [X|L2], R).
```

- A `lists` könyvtár tartalmazza az `append/3` és `reverse/2` eljárások definícióját.
- A könyvtár betöltése:

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

append és revapp — listák gyűjtési iránya

- Prolog megvalósítás

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

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

- C++ megvalósítás

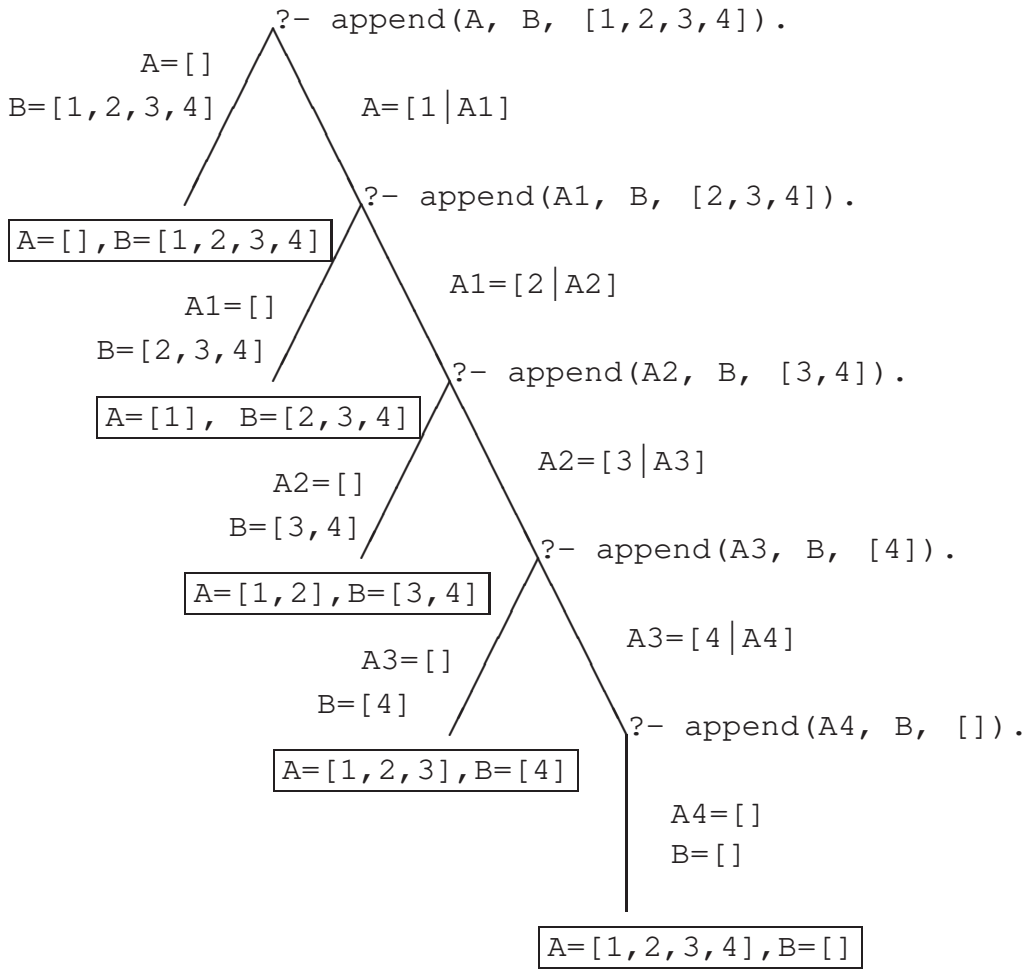
```
struct link   { link *next;
                char elem;
                link(char e): elem(e) {}
              };
typedef link *list;
```

```
list append(list list1, list list2)
{ list list3, *lp = &list3;
  for (list p=list1; p; p=p->next)
  { list newl = new link(p->elem);
    *lp = newl; lp = &newl->next;
  }
  *lp = list2;
  return list3;
}
```

```
list revapp(list list1, list list2)
{ list l = list2;
  for (list p=list1; p; p=p->next)
  { list newl = new link(p->elem);
    newl->next = l; l = newl;
  }

  return l;
}
```

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
```



Variációk `appendre` 1. — Három lista összefűzése

- Az `append/3` keresési tere **véges**, ha első és harmadik argumentuma közül legalább az egyik zárt végű lista.
- `append(L1,L2,L3,L123)`: `L1` ⊕ `L2` ⊕ `L3` = `L123`

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

  - Nem hatékony, 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).
```

  - Az első `append/3` hívás nyílt végű listát állít elő:

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

Mintakeresés append/3-mal

- Párban előforduló elemek

```
% párban(Lista, Elem): A Lista számlistának Elem olyan
% eleme, amelyet egy ugyanilyen elem követ.
párban(L, E) :-
    append(_, [E,E|_], L).

| ?- párban([1,8,8,3,4,4], E).
        E = 8 ? ; E = 4 ? ; no
```

- Dadogó részek

```
% dadogó(L, D): D olyan nem üres részlistája L-nek,
% amelyet egy vele megegyező részlista követ.
dadogó(L, D) :-
        append(_, Farok, L),
        D = [_|_],
        append(D, Vég, Farok),
        append(D, _, Vég).

| ?- dadogó([2,2,1,2,2,1], D).
        D = [2] ? ; D = [2,2,1] ? ; D = [2] ? ; no
```

Keresés listában

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

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

```
member(Elem, [Fej|Farok]) :-
        (   Elem = Fej
        ;   member(Elem, Farok)
        ).
```

- A `member/2` felhasználási lehetőségei
  - Eldöntendő (igen-nem) kérdés:

    `| ?- member(2, [1,2,3]).` ⇒ `yes`
  - 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 – mindkét fenti hívásmintát használja:

    `| ?- 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 második argumentuma zárt végű lista.

member/2 általánosítása: select/3

- select(Elem, Lista, Marad): Elemet a Listaból elhagyva marad Marad.

```
select(Elem, [Elem|Marad], Marad).      % Elhagyjuk a fejet, marad a farok.
select(Elem, [X|Farok], [X|Marad0]) :-  % Marad a fej,
        select(Elem, Farok, Marad0).    %    a farokból hagyunk el elemet.
```

- Felhasználási lehetőségek:

```
| ?- select(1, [2,1,3], L).             % Adott elem elhagyása
        L = [2,3] ? ; 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
        no                              % úgy, hogy [2,...]-t kapjunk?
| ?- select(1, [X,2,X,3], L).
        L = [2,1,3], X = 1 ? ; L = [1,2,3], X = 1 ? ; no
```

- A lists könyvtár tartalmazza a member/2 és select/3 eljárások definícióját is.
- A select/3 keresési tere **véges**, ha 2. és 3. argumentuma közül legalább az egyik zárt végű.

Listák permutációja

- `permutation(Lista, Perm)`: `Lista` permutációja a `Perm` lista.
  (Az alábbi definíció a `library(lists)` könyvtárból származik:)

```
permutation([], []).
permutation(Lista, [Elso|Perm]) :-
        select(Elso, Lista, Maradek),
        permutation(Maradek, Perm).
```

- Felhasználási példák:

```
| ?- permutation([1,2], L).
        L = [1,2] ? ; L = [2,1] ? ; no
| ?- permutation([a,b,c], L).
        L = [a,b,c] ? ; L = [a,c,b] ? ; L = [b,a,c] ? ;
        L = [b,c,a] ? ; L = [c,a,b] ? ; L = [c,b,a] ? ;
        no
| ?- permutation(L, [1,2]).
        L = [1,2] ? ;
        végtelen keresési tér
```

- Ha `permutation/2`-ben az első argumentum ismeretlen, akkor a `select` hívás keresési tere végtelen!

TÍPUSOK PROLOGBAN

Példa: Bináris fák

- Az egészekből álló bináris fa különböző meghatározásai:
  - Szöveges definícióként (ismétlés):
    - vagy egy levél (`leaf(V)`), ahol `V` egy egész szám
    - vagy egy csomópont (`node(L,R)`), ahol `L` és `R` egészekből álló bináris fák
  - Matematikai jelöléssel:

    $\texttt{itree} \equiv \{\texttt{leaf}(i) \mid i \in \texttt{int}\} \cup \{\texttt{node}(l,r) \mid l,r \in \texttt{itree}\}$
  - A bevezetendő típus-jelölésekkel (két ekvivalens megfogalmazás):

```
:- type itree ==  {node(itree, itree)} \/ {leaf(int)}.
:- type itree --->  node(itree, itree) | leaf(int).
```

  - Egy **ellenőrző** Prolog predikátumként:

```
itree(leaf(V)) :-
        integer(V).
itree(node(L,R)) :-
        itree(L), itree(R).
```

- Az ilyen adattípust **megkülönböztetett unió**nak nevezzük, mert az unióban szereplő halmazokat az elemeik funktora megkülönbözteti (`leaf/1`, `node/2`)

Típusok leírása Prologban

- Típusleírás: (tömör) Prolog kifejezések egy halmazának megadása
- Alaptípusok leírása: `int, float, number, atom, any`
- Új típusok felépítése:

  `{ str(T`$_1$`, ..., T`$_n$`) }` $\equiv$ `{ str(`e_1`, ..., `e_n`) | `$e_1 \in$`T`$_1$`, ..., `$e_n \in$`T`$_n$` }`, $n \geq 0$

  Példa: `{személy(atom,atom,int)}` az olyan `személy/3` funktorú struktúrák halmaza, amelyben az első két argumentum atom, a harmadik egész.
- Típusok, mint halmazok úniója képezhető a `\/` operátorral.

  `{személy(atom,atom,int)} \/ {atom-atom} \/ atom`
- Egy típusleírás elnevezhető (kommentben): `:- type` *`tnév`* `==` *`tleírás`*`.`

  ```
  :- type t1 == {atom-atom} \/ atom.,
  :- type ember == {ember-atom} \/ {semmi}.
  ```
- Megkülönböztetett únió: csupa különböző funktorú összetett típus úniója. Ha S_1,..., S_n mind különböző funktorú, alkalmazható az egyszerűsített (Mercury) jelölés:

  `:- type` T `== {` S_1 `} \/ ...\/ {` S_n `}.` $\Rightarrow$ `:- type` T `---> `S_1` ; ...; `S_n`.` Példák:

  ```
  :- type ember ---> ember-atom; semmi.
  :- type fa ---> leaf(int) ; node(fa,fa).
  ```

Típusok leírása Prologban — folytatás

- Paraméteres típusok — példák

```
:- type pair(T1, T2) ---> T1 - T2.      % egy  '-' nevű kétarg.-ú struktúra,
                                        % első arg. T1, a második T2 típusú.
:- type tree(T) ---> leaf(T)            % T típusú elemekből álló
          ; node(tree(T),tree(T)).      % bináris fa
:- type assoc_tree(KeyT, ValueT)        % KeyT és ValueT típusú
          == tree(pair(KeyT, ValueT)). % párokból álló fa
:- type szótár == assoc_list(szó, szó).
:- type szó == atom.
```

- Típusdeklarációk szintaxisa

⟨ típusdeklaráció ⟩	::=	⟨ típuselnevezés ⟩ | ⟨ típuskonstrukció ⟩
⟨ típuselnevezés ⟩	::=	`:- type` ⟨ típusazonosító ⟩ `==` ⟨ típusleírás ⟩ .
⟨ típuskonstrukció ⟩	::=	`:- type` ⟨ típusazonosító ⟩ `--->` ⟨ megkülönb. únió ⟩ .
⟨ megkülönb. únió ⟩	::=	⟨ konstruktor ⟩ `;` ...
⟨ konstruktor ⟩	::=	⟨ névkonstans ⟩ | ⟨ struktúranév ⟩ `(` ⟨ típusleírás ⟩, ... `)`
⟨ típusleírás ⟩	::=	⟨ típusazonosító ⟩ | ⟨ típusváltozó ⟩ | `{` ⟨ konstruktor ⟩ `}` |
		⟨ típusleírás ⟩ `\/` ⟨ típusleírás ⟩
⟨ típusazonosító ⟩	::=	⟨ típusnév ⟩ | ⟨ típusnév ⟩ `(` ⟨ típusváltozó ⟩, ... `)`
⟨ típusnév ⟩	::=	⟨ névkonstans ⟩
⟨ típusváltozó ⟩	::=	⟨ változó ⟩

Predikátumtípus-deklarációk

- Predikátumtípus-deklaráció
  `:- pred` ⟨eljárásnév⟩`(`⟨típusazonosító⟩, ...`)`
- Példa:

```
:- pred sum_tree(tree(int), int).
```

- Predikátummód-deklaráció (Nem kötelező, több is megadható.)
  `:- mode` ⟨eljárásnév⟩`(`⟨módazonosító⟩, ...`)` ahol ⟨módazonosító⟩ ::= `in` | `out` | `inout`.

  (Mercury-ban az `inout` módazonosító nem megengedett.)
- Példák:

```
:- mode sum_tree(in, in).    % ellenőrzés
:- mode sum_tree(in, out).   % fa-összeg előállítása
:- mode sum_tree(out,in).    % adott összegű fa építése
```

- Vegyes típus- és móddeklaráció
  `:- pred` ⟨eljárásnév⟩`(`⟨típusazonosító⟩`::`⟨módazonosító⟩, ...`)`
- Példa:

```
:- pred between(int::in, int::in, int::out).
```

Móddeklaráció: a SICStus kézikönyv által használt alak

- A SICStus kézikönyv egy másik jelölést használ a bemenő/kimenő argumentumok jelzésére, pl.

```
sum_tree(+T, ?Sum).
```

- Mód-jelölő karakterek:
  - `+` bemenő argumentum (behelyettesített)
  - `-` kimenő argumentum (behelyettesítetlen)
  - `:` eljárás-paraméter (meta-eljárásokban)
  - `?` tetszőleges

A PROLOG SZINTAXIS

A Prolog szintaxis összefoglalása

- A Prolog szintaxis alapelvei
  - Minden programelem kifejezés!
  - A szükséges összekötő jelek (`',', ;, :- -->`): szabványos operátorok.
  - A beolvasott kifejezést funktora alapján osztályozzuk:
    - *kérdés:* `?- Cél.`
      `Cél`t lefuttatja, és a változó-behelyettesítéseket kiírja (ez az alapértelmezés az ún. top-level interaktív felületen).
    - *parancs:* `:- Cél.`
      A `Cél`t csendben lefuttatja. Pl. deklaráció (operátor, ...) elhelyezésére.
    - *szabály:* `Fej :- Törzs.`
      A szabályt felveszi a programba.
    - *nyelvtani szabály:* `Fej --> Törzs.`
      Prolog szabállyá alakítja és felveszi (lásd a DCG nyelvtan).
    - *tényállítás:* `Minden egyéb kifejezés.`
      Üres törzsű szabályként felveszi a programba.

A Prolog nyelv-változatok

- A SICStus rendszer két üzemmódja
  - `iso` Az ISO Prolog szabványnak megfelelő.
  - `sicstus` Korábbi változatokkal kompatibilis.
  - Állítása: `set_prolog_flag(language, Mód).`
  - Különbségek:
    - szintaxis-részletek, pl. a `0x1ff` szám-alak csak ISO módban,
    - beépített eljárások viselkedésének kisebb eltérései.
  - az eddig ismertetett eljárások hatása lényegében nem változik.

Szintaktikus édesítőszerek — összefoglalás, gyakorlati tanácsok

- Operátoros kifejezések alapstruktúra alakra hozása
  - Zárójelezzük be a kifejezést, az operátorok prioritása és fajtája alapján, például `-a+b*2` ⇒ `((-a)+(b*2)).`
  - Hozzuk az operátoros kifejezéseket alapstruktúra alakra:
    `(A Inf B)` ⇒ `Inf(A,B)`, `(Pref A)` ⇒ `Pref(A)`, `(A Postf)` ⇒ `Postf(A)`
    Példa: `((-a)+(b*2))` ⇒ `(-(a)+ *(b,2))` ⇒ `+(-(a),*(b,2)).`
  - Trükkös esetek:
    - A vesszőt névként idézni kell: pl. `(pp,(qq;rr))` ⇒ `','(pp,;(qq,rr)).`
    - `-` *`Szám`* ⇒ negatív számkonstans, de `-` *`Egyéb`* ⇒ prefix alak.
      Példa. `-1+2` ⇒ `+(-1,2)`, de `-a+b` ⇒ `+(-(a),b).`
    - *`Név`*`(...)` ⇒ struktúrakifejezés;
      `Név` `(...)` ⇒ prefix operátoros kifejezés. Példák:
      `-(1,2)` ⇒ `-(1,2)` (változatlan), de
      `- (1,2)` ⇒ `-(','(1,2)).`

Szintaktikus édesítőszerek — listák, egyebek

- Listák alapstruktúra alakra hozása
  - Farok-megadás betoldása.
    `[1,2]` ⇒ `[1,2|[]]`. `[[X|Y]]` ⇒ `[[X|Y]|[]]`
  - Vessző (ismételt) kiküszöbölése `[Elem1,Elem2...]` ⇒ `[Elem1|[Elem2...]]`.
    `[1,2|[]]` ⇒ `[1|[2|[]]]`
    `[1,2,3|[]]` ⇒ `[1|[2,3|[]]]` ⇒ `[1|[2|[3|[]]]]`
  - Struktúrakifejezéssé alakítás: `[Fej|Farok]` ⇒ `.(Fej,Farok)`.
    `[1|[2|[]]]` ⇒ `.(1,.(2,[]))`, `[[X|Y]|[]]` ⇒ `.(.(X,Y),[])`
- Egyéb szintaktikus édesítőszerek:
  - Karakterkód-jelölés: `0'Kar`.
    `0'a` ⇒ `97`, `0'b` ⇒ `98`, `0'c` ⇒ `99`, `0'd` ⇒ `100`, `0'e` ⇒ `101`
  - Füzér (string): `"xyz..."` ⇒ az `xyz...` karakterek kódját tartalmazó lista
    `"abc"` ⇒ `[97,98,99]`, `""` ⇒ `[]`, `"e"` ⇒ `[101]`
  - Kapcsos zárójelezés: `{Kif}` ⇒ `{}(Kif)` (egy `{}` nevű, egyargumentumú struktúra — a `{}` jelpár egy önálló lexikai elem, egy névkonstans).
  - Bináris, hexa stb. alak (csak `iso` módban), pl. `0b101010`, `0x1a`.

Kifejezések szintaxisa — kétszintű nyelvtanok

- Egy részlet egy „hagyományos” nyelv kifejezés-szintaxisából:

⟨ kifejezés ⟩ ::= ⟨ tag ⟩
| ⟨ kifejezés ⟩ ⟨ additív művelet ⟩ ⟨ tag ⟩

⟨ tag ⟩ ::= ⟨ tényező ⟩
| ⟨ tag ⟩ ⟨ multiplikatív művelet ⟩ ⟨ tényező ⟩

⟨ tényező ⟩ ::= ⟨ szám ⟩ | ⟨ azonosító ⟩ | (⟨ kifejezés ⟩)

- Ugyanez kétszintű nyelvtannal:

⟨ kifejezés ⟩ ::= ⟨ kif *2* ⟩

⟨ kif *N* ⟩ ::= ⟨ kif *N-1* ⟩
| ⟨ kif *N* ⟩ ⟨ *N* prioritású művelet ⟩ ⟨ kif *N-1* ⟩

⟨ kif *0* ⟩ ::= ⟨ szám ⟩ | ⟨ azonosító ⟩ | (⟨ kif *2* ⟩)

{az additív ill. multiplikatív műveletek prioritása *2* ill. *1* }

Kifejezések szintaxisa

⟨ programelem ⟩ ::= ⟨ kifejezés 1200 ⟩ ⟨ záró-pont ⟩

⟨ kifejezés *N* ⟩ ::=
- ⟨ op *N* fx ⟩ ⟨ köz ⟩ ⟨ kifejezés *N-1* ⟩
- | ⟨ op *N* fy ⟩ ⟨ köz ⟩ ⟨ kifejezés *N* ⟩
- | ⟨ kifejezés *N-1* ⟩ ⟨ op *N* xfx ⟩ ⟨ kifejezés *N-1* ⟩
- | ⟨ kifejezés *N-1* ⟩ ⟨ op *N* xfy ⟩ ⟨ kifejezés *N* ⟩
- | ⟨ kifejezés *N* ⟩ ⟨ op *N* yfx ⟩ ⟨ kifejezés *N-1* ⟩
- | ⟨ kifejezés *N-1* ⟩ ⟨ op *N* xf ⟩
- | ⟨ kifejezés *N* ⟩ ⟨ op *N* yf ⟩
- | ⟨ kifejezés *N-1* ⟩

⟨ kifejezés 1000 ⟩ ::= ⟨ kifejezés 999 ⟩ , ⟨ kifejezés 1000 ⟩

⟨ kifejezés 0 ⟩ ::=
- ⟨ név ⟩ (⟨ argumentumok ⟩)
  - { A ⟨ név ⟩ és a (közvetlenül egymás után áll! }
- | (⟨ kifejezés 1200 ⟩) | { ⟨ kifejezés 1200 ⟩ }
- | ⟨ lista ⟩ | ⟨ füzér ⟩
- | ⟨ név ⟩ | ⟨ szám ⟩ | ⟨ változó ⟩

Kifejezések szintaxisa — folytatás

⟨ op *N T* ⟩ ::=		⟨ név ⟩ {feltéve, hogy ⟨ név ⟩ *N* prioritású és *T* típusú operátornak lett deklarálva}
⟨ argumentumok ⟩ ::=		⟨ kifejezés 999 ⟩
	|	⟨ kifejezés 999 ⟩ `,` ⟨ argumentumok ⟩
⟨ lista ⟩ ::=		`[]`
	|	`[` ⟨ listakif ⟩ `]`
⟨ listakif ⟩ ::=		⟨ kifejezés 999 ⟩
	|	⟨ kifejezés 999 ⟩ `,` ⟨ listakif ⟩
	|	⟨ kifejezés 999 ⟩ | ⟨ kifejezés 999 ⟩
⟨ szám ⟩ ::=		⟨ előjeltelen szám ⟩
	|	+ ⟨ előjeltelen szám ⟩
	|	− ⟨ előjeltelen szám ⟩
⟨ előjeltelen szám ⟩ ::=		⟨ természetes szám ⟩
	|	⟨ lebegőpontos szám ⟩

Kifejezések szintaxisa — megjegyzések

- A ⟨kifejezés N⟩-ben ⟨köz⟩ csak akkor kell ha az őt követő kifejezés nyitó-zárójellel kezdődik.

```
| ?- op(500, fx, succ).
yes
| ?- write_canonical(succ (1,2)), nl, write_canonical(succ(1,2)).
succ(','(1,2))
succ(1,2)
```

- A `{` ⟨kifejezés⟩ `}` azonos a `{}(`⟨kifejezés⟩`)` struktúrával, ez pl. a DCG nyelvtanoknál hasznos.

```
| ?- write_canonical({a}).
{}(a)
```

- Egy ⟨füzér⟩ `"` jelek közé zárt karaktersorozat, általában a karakterek kódjainak listájával azonos.

```
| ?- write("baba").
[98,97,98,97]
```

A Prolog lexikai elemei 1. (ismétlés)

- ⟨ név ⟩
  - kisbetűvel kezdődő alfanumerikus jelsorozat (ebben megengedve kis- és nagybetűt, számjegyeket és aláhúzásjelet);
  - egy vagy több ún. speciális jelből (`+-*/\$^<>=~:.?@#&`) álló jelsorozat;
  - az önmagában álló `!` vagy `;` jel;
  - a `[]` `{}` jelpárok;
  - idézőjelek (`'`) közé zárt tetszőleges jelsorozat, amelyben \ jellel kezdődő escape-szekvenciákat is elhelyezhetünk.
- ⟨ változó ⟩
  - nagybetűvel vagy aláhúzással kezdődő alfanumerikus jelsorozat.
  - az azonos jelsorozattal jelölt változók egy klózon belül azonosaknak, különböző klózokban különbözőeknek tekintődnek;
  - kivétel: a semmis változók (_) minden előfordulása különböző.

A Prolog lexikai elemei 2.

- ⟨ természetes szám ⟩
  - (decimális) számjegysorozat;
  - 2, 8 ill. 16 alapú számrendszerben felírt szám, ilyenkor a számjegyeket rendre a `0b`, `0o`, `0x` karakterekkel kell prefixálni (csak `iso` módban)
  - karakterkód-konstans `0'` *c* alakban, ahol *c* egyetlen karakter
- ⟨ lebegőpontos szám ⟩
  - mindenképpen tartalmaz tizedespontot
  - mindkét oldalán legalább egy (decimális) számjeggyel
  - `e` vagy `E` betűvel jelzett esetleges exponens

Megjegyzések és formázó-karakterek

- Megjegyzések (comment)
  - A `%` százalékjeltől a sor végéig
  - A `/*` jelpártól a legközelebbi `*/` jelpárig.
- Formázó elemek
  - szóköz, újsor, tabulátor stb. (nem látható karakterek)
  - megjegyzés
- A programszöveg formázása
  - formázó elemek (szóköz, újsor stb.) szabadon elhelyezhetők;
  - kivétel: struktúrakifejezés neve után nem szabad formázó elemet tenni;
  - prefix operátor és `(` közé kötelező formázó elemet tenni;
  - ⟨záró-pont⟩: egy `.` karakter amit egy formázó elem követ.

PROLOG PÉLDÁK

A régi jegyzet bevezető példásora: útvonalkeresés

- A feladat:
  - Tekintsük (autóbusz)járatok egy halmazát.
  - Mindegyik járathoz a két végpont és az útvonal hossza van megadva.
  - Írjunk Prolog eljárást, amellyel megállapítható, hogy két pont összeköthető-e pontosan N csatlakozó járattal!
- Átfogalmazás: egy súlyozott irányítatlan gráfban két pont közötti utat keresünk. Élek:

```
% járat(A, B, H): Az A és B városok között van járat, és hossza H km.
járat('Budapest', 'Prága', 515).
járat('Budapest', 'Bécs', 245).
járat('Bécs', 'Berlin', 635).
járat('Bécs', 'Párizs', 1265).
```

- Irányított élek:

```
% útszakasz(A, B, H): A-ból B-be eljuthatunk egy H úthosszú járattal.
útszakasz(Kezdet, Cél, H) :-
     (   járat(Kezdet, Cél, H)
     ;   járat(Cél, Kezdet, H)
     ).
```

Az útvonalkeresési feladat — folytatás

- Adott lépésszámú útvonal (él-sorozat) és hossza:

```
% útvonal(N, A, B, H): A és B között van (pontosan)
% N szakaszból álló útvonal, amelynek összhossza H.
útvonal(0, Hová, Hová, 0).
útvonal(N, Honnan, Hová, H) :-
    N > 0,
    N1 is N-1,
    útszakasz(Honnan, Közben, H1),
    útvonal(N1, Közben, Hová, H2),
    H is H1+H2.
```

- Futási példa:

```
| ?- útvonal(2, 'Párizs', Hová, H).
        H = 1900, Hová = 'Berlin' ? ;
        H = 2530, Hová = 'Párizs' ? ;
        H = 1510, Hová = 'Budapest' ? ;
        no
```

Körmentes út keresése

- Könyvtár betöltése, adott funktorú eljárások importálásával:

```
:- use_module(library(lists), [member/2]).
```

- Segéd-argumentum: az érintett városok listája, fordított sorrendben

```
% útvonal_2(N, A, B, H): A és B között van (pontosan)
% N szakaszból álló körmentes útvonal, amelynek összhossza H.
útvonal_2(N, Honnan, Hová, H) :-
    útvonal_2(N, Honnan, Hová, [Honnan], H).

% útvonal_2(N, A, B, Kizártak, H): A és B között van pontosan
% N szakaszból álló körmentes, Kizártak elemein át nem menő H hosszú út.
útvonal_2(0, Hová, Hová, Kizártak, 0).
útvonal_2(N, Honnan, Hová, Kizártak, H) :-
    N > 0, N1 is N-1, útszakasz(Honnan, Közben, H1),
    \+ member(Közben, Kizártak),
    útvonal_2(N1, Közben, Hová, [Közben|Kizártak], H2), H is H1+H2.
```

- Példa-futás:

```
| ?- útvonal_2(2, 'Párizs', Hová, H).
        H = 1900, Hová = 'Berlin' ? ;
        H = 1510, Hová = 'Budapest' ? ; no
```

Továbbfejlesztés: körmentes út keresése, útvonal-gyűjtéssel

- Az alapötlet: a `Kizártak` listában gyűlik a (fordított) útvonal.
- A rekurzív eljárásban szükséges egy **új argumentum**, hogy az útvonalat kiadjuk!

```
:- use_module(library(lists), [member/2, reverse/2]).

% útvonal_3(N, A, B, Út, H): A és B között van (pontosan)
% N szakaszból álló körmentes Út útvonal, amelynek összhossza H.
útvonal_3(N, Honnan, Hová, Út, H) :-
    útvonal_3(N, Honnan, Hová, [Honnan], FÚt, H),
    reverse(FÚt, Út).

% útvonal_3(N, A, B, FÚt0, FÚt, H): A és B között van pontosan
% N szakaszból álló körmentes, FÚt0 elemein át nem menő H hosszú út.
% FÚt = (az A → B útvonal megfordítása) ⊕ FÚt0.
útvonal_3(0, Hová, Hová, FordÚt, FordÚt, 0).
útvonal_3(N, Honnan, Hová, FordÚt0, FordÚt, H) :-
    N > 0, N1 is N-1, útszakasz(Honnan, Közben, H1),
    \+ member(Közben, FordÚt0),
    útvonal_3(N1, Közben, Hová, [Közben|FordÚt0], FordÚt, H2), H is H1+H2.

| ?- útvonal_3(2, 'Párizs', _, Út, H).
        H = 1900, Út = ['Párizs','Bécs','Berlin'] ? ;
        H = 1510, Út = ['Párizs','Bécs','Budapest'] ? ; no
```

Súlyozott gráf ábrázolása éllistával

- A gráf ábrázolása
  - a gráf élek listája,
  - az él egy három-argumentumú struktúra,
  - argumentumai: a két végpont és a súly.
- Típus-definíció

```
% :- type él ---> él(pont, pont, súly).
% :- type pont == atom.
% :- type súly == int.
% :- type gráf == list(él).
```

- Példa

```
hálózat([él('Budapest','Bécs',245),
         él('Budapest','Prága',515),
         él('Bécs','Berlin',635),
         él('Bécs','Párizs',1265)]).
```

Ismétlődésmentes útvonal keresése listával ábrázolt gráfban

```
:- use_module(library(lists), [select/3]).

% útvonal_4(N, G, A, B, L, H): A G gráfban van egy  A-ból
% B-be menő N szakaszból álló L út, melynek összhossza H.
útvonal_4(0, _Gráf, Hová, Hová, [Hová], 0).
útvonal_4(N, Gráf, Honnan, Hová, [Honnan|Út], H) :-
        N >0, N1 is N-1,
        select(Él, Gráf, Gráf1),
        él_végpontok_hossz(Él, Honnan, Közben, H1),
        útvonal_4(N1, Gráf1, Közben, Hová, Út, H2),
        H is H1+H2.

% él_végpontok_hossz(Él, A, B, H): Az Él irányítatlan él
% végpontjai A és B, hossza H.
él_végpontok_hossz(él(A,B,H), A, B, H).
él_végpontok_hossz(él(A,B,H), B, A, H).

| ?- hálózat(_Gráf), útvonal_4(2, _Gráf, 'Budapest', _, Út, H).
        H = 880, Út = ['Budapest','Bécs','Berlin'] ? ;
        H = 1510, Út = ['Budapest','Bécs','Párizs'] ? ;
        no
```

Bináris fákra vonatkozó példasor — fa levele

- Ismétlés: egészekből álló bináris fa:

```
:- type itree ==  {node(itree, itree)} \/ {leaf(int)}.
:- type itree --->  node(itree, itree) | leaf(int).
```

- Írjunk egy predikátumot annak eldöntésére, hogy egy adott érték szerepel-e egy fa levelében (vö. `member/2`)!

```
% fa_levele(Fa, Ertek): A Fa bináris fa levelében szerepel az Ertek szám.
fa_levele(leaf(V), V).        % ha a fa egyetlen levélből áll és a levélbeli
                              % érték megegyezik a keresettel, akkor ``siker''
fa_levele(node(L,_), V) :-
        fa_levele(L, V).      % ha  a bal részfában van, akkor az egészben is
fa_levele(node(_,R), V) :-
        fa_levele(R, V).      % ha a jobb részfában van, akkor az egészben is
```

- Az aláhúzásjel egy ún. semmis (void) változó, ennek minden előfordulása különböző változó!
- Példák: ellenőrzés (1), adott fa leveleinek felsorolása (2), adott levelű fák felsorolása, (3) (∞ keresési tér).

```
| ?- fa_levele(node(node(leaf(1),leaf(2)),leaf(7)), 2). ⟹ yes          (1)
| ?- fa_levele(node(node(leaf(1),leaf(2)),leaf(7)), 3). ⟹ no           (1)
| ?- fa_levele(node(leaf(1),leaf(7)), E). ⟹ E = 1 ? ; E = 7 ? ; no     (2)
| ?- fa_levele(Fa, 3). ⟹ Fa = leaf(3) ? ; Fa = node(leaf(3),_A) ? ;  ... (3)
```

Összetett adatstruktúrák konjunktív és diszjunktív bejárása

- Prologban egy összetett adatstruktúrát kétféleképpen lehet bejárni:
  - konjunktívan: a részek bejárása ÉS kapcsolatban van, általában egy eredményt ad
    - pl. fa összegzése (`sum_tree`), fa ellenőrzése (`itree`), fa kiírása:

```
% faki(Fa): Fa kiírható (mindig teljesül :-). Mellékhatásként kiírja a Fa fát.
faki(leaf(V)) :-
        write(@), write(V).     % A write(X) beépített pred. kiírja az X kifejezést.
faki(node(L,R)) :-
        write('('), faki(L), write(' -- '), faki(R), write(')').

| ?- faki(node(node(leaf(1),leaf(8)),leaf(7))).     ⇒ ((@1 -- @8) -- @7)
yes
```

  - diszjunktívan: a részek bejárása VAGY kapcsolatban van, visszalépéskor új eredmény
    - pl. fa leveleinek felsorolása (`fa_levele`)
- A diszjunktív, felsoroló bejárás könnyen kiegészíthető további feltételekkel
  - Keressük egy fának az `(5,10)` intervallumba eső leveleit:

```
| ?- _Fa = node(node(leaf(1),leaf(8)),leaf(7)), fa_levele(_Fa, E), 5 < E, E < 10.
     ⟹ E = 8 ? ; E = 7 ? ; no
| ?- _Fa = (...),  fa_levele(_Fa, E), 5 < E, E < 10, write(E), write(' '), fail.
     ⇒ 8 7  ⟹ no
```

- A `fail` beépített predikátum mindig meghiúsul, pl. ún. visszalépéses ciklus szervezésére jó.

Levél elhagyása bináris fából

- Írjunk egy predikátumot annak eldöntésére, hogy egy adott érték szerepel-e egy összetett fa levelében! A predikátum adja vissza a levél elhagyása után fennmaradó fát!

```
% flm(Fa, Ertek, Marad): A Fa összetett bináris fa egy Ertek értékű
% levelének elhagyása után marad a Marad fa. (flm = fa_level_maradek)
flm(node(leaf(V),T), V, T).    % ha a bal részfa a keresett levél
                               % akkor a jobb részfa a maradék
flm(node(T,leaf(V)), V, T).    % ugyanez jobboldali levél esetére
flm(node(L0,R), V, node(L,R)) :-
        flm(L0, V, L).         % ha a bal részfából elhagyható a levél
                               % akkor ennek maradéka, kiegészítve
                               % a jobb részfával, lesz a teljes fa maradéka
flm(node(L,R0), V, node(L,R1)) :-
        flm(R0, V, R1).        % ugyanez jobb részfa esetére
```

- Az `flm/3` predikátum használható ellenőrzése, de fa szétbontására is:

```
| ?- flm(node(leaf(1),node(leaf(2),leaf(3))), 2, T).  ⟹
   T = node(leaf(1),leaf(3)) ? ; no
| ?- flm(node(leaf(1),node(leaf(2),leaf(3))), 7, T).  ⟹ no
| ?- flm(node(leaf(1),node(leaf(2),leaf(3))), X,  T). ⟹
   T = node(leaf(2),leaf(3)), X = 1 ? ;
   T = node(leaf(1),leaf(3)), X = 2 ? ;
   T = node(leaf(1),leaf(2)), X = 3 ? ; no
```

Levél beszúrása bináris fába

- Írjunk egy predikátumot arra, hogy egy adott értékű levelet egy fába minden lehetséges módon beszúrjon!
- Nem kell irnunk, már megírtuk! Az `flm` predikátum erre is jó:

```
% flm(Fa, Ertek, Marad): A Fa összetett bináris fa egy Ertek értékű
% levelének elhagyása után marad a Marad fa. Röviden: Fa - Ertek = Marad.

% flm(Fa, Ertek, Marad): A Fa (összetett) bináris fa úgy áll elő, hogy
% a Marad fába beszúrunk egy E értékű levelet. Fa = Marad + Ertek.
flm(node(leaf(V),T), V, T).    % Egy T fába beszúrhatunk egy levelet
(...)                          % úgy, hogy az egylevelű fát T elé tesszük
```

- Példák:

```
| ?- flm(Fa, 2, leaf(1)), faki(Fa), write(' '), fail.
(@2 -- @1) (@1 -- @2)                          ⟹ no
| ?- flm(Fa0, 2, leaf(1)), flm(Fa, 3, Fa0), faki(Fa), write(' '), fail.
(@3 -- (@2 -- @1)) ((@2 -- @1) -- @3) ((@3 -- @2) -- @1) ((@2 -- @3) -- @1)
(@2 -- (@3 -- @1)) (@2 -- (@1 -- @3)) (@3 -- (@1 -- @2)) ((@1 -- @2) -- @3)
((@3 -- @1) -- @2) ((@1 -- @3) -- @2) (@1 -- (@3 -- @2)) (@1 -- (@2 -- @3))  ⟹ no

negylevelu(X, Y, Z, U, Fa) :-  % Fa az X, Y, Z, U levelekből áll
     flm(Fa0, Y, leaf(X)), flm(Fa1, Z, Fa0), flm(Fa, U, Fa1).

| ?- findall(Fa, negylevelu(1,3,4,6,Fa), Fak), length(Fak,Db). ⟹  Db = 120, Fak = (...)
```

Példa: adott értékű kifejezés előállítása

- A feladat: írjunk Prolog programot a következő feladvány megoldására:
  - Az 1, 3, 4, 6 számokból a négy alapművelet felhasználásával állítsuk elő a 24 számértéket!
  - Mind a négy számot fel kell használni, tetszőleges sorrendben.
  - Tetszőleges alapműveletek használhatók, tetszőlegesen zárójelezéssel.
- Már van egy predikátumunk (`negylevelu/5`), amely adott számokból tetszőleges fát épít.
- Definiáljunk egy predikátumot, amely egy fának megfelelő aritmetikai kifejezéseket készít!

```
% fa_kif(Fa, Kif): Kif a Fa fával azonos alakú, azonos számokból álló
% aritmetikai kifejezés, amelyben a négy alapművelet fordulhat elő.
fa_kif(leaf(V), V).
fa_kif(node(L,R), Exp) :-
        fa_kif(L, E1),
        fa_kif(R, E2),
        alap4(E1, E2, Exp).

% alap4(X, Y, Kif): Kif 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).

| ?- fa_kif(node(leaf(1),node(leaf(2),leaf(3))), Kif).
Kif = 1+(2+3) ? ; Kif = 1-(2+3) ? ; Kif = 1*(2+3) ? ; Kif = 1/(2+3) ? ;
(...)
Kif = 1+2/3 ?   ; Kif = 1-2/3 ?   ; Kif = 1*(2/3) ? ; Kif = 1/(2/3) ? ; no
```

Példa: adott értékű kifejezés előállítása (folyt.)

- Korábban elkészített predikátumok:
  - adott számokból álló fákat felsoroló `negylevelu/5`
  - adott fával azonos szerkezetű aritmetikai kifejezéseket felsoroló `fa_kif/2`
- Ezekre építve könnyen megírható a feladvány megoldására használható predikátum:

```
% Kif egy a négy alapművelettel az X, Y, Z, U számokból
% felépített kifejezés, amelynek értéke Ertek.
negylevelu_erteke(X, Y, Z, U, Ertek, Kif) :-
        negylevelu(X, Y, Z, U, Fa),
        fa_kif(Fa, Kif),
        Kif =:= Ertek.

| ?- negylevelu_erteke(1,3,4,6,24,Kif).
...
```

- Megjegyzések
  - Az aritmetikai eljárásokban a változók nem csak számokra, hanem tömör aritmetikai kifejezésekre is be lehetnek helyettesítve.
  - A `negylevelu_erteke` eljárás utolsó hívása helyett **nem** lenne jó: `Ertek is Kif`. Miért?