

Deklaratív Programozás

Hanák Péter

hanak@inf.bme.hu

Irányítástechnika és Informatika Tanszék

OM Kutatás-Fejlesztési Helyettes Államtitkárság

Szeregi Péter, Benkő Tamás

{szeregi, benko}@iqsoft.hu

Számítástudományi és Információelméleti Tanszék

IQSOFT Intelligens Software Rt.

KÖVETELMÉNYEK — TUDNIVALÓK

Deklaratív programozás: tudnivalók

Honlap, levelezési lista

- Honlap: `<http://www.inf.bme.hu/~dp>`
- Levlista: `<http://www.inf.bme.hu/mailman/listinfo/dp-l>`. Moderált.
Csak a feliratkozottak küldhetnek levelet a `<dp-l@inf.bme.hu>` címre.

Jegyzet

- Szeredi Péter, Benkő Tamás: Deklaratív programozás. Bevezetés a logikai programozásba.
- Hanák D. Péter: Deklaratív programozás. Bevezetés a funkcionális programozásba.
- Új, bővített kiadások, kötetenként 600-800 Ft, terjedelemtől függően
- Előző kiadások a honlapon (ps, pdf)
- Jegyzetrendelés: a honlapon megadandó módon

Deklaratív programozás: tudnivalók (folyt.)

Fordító- és értelmezőprogramok

- SICStus Prolog (3.8.5, licenszköteles, aláírás ellenében jelszót adunk)
- Moscow SML (2.0, szabad szoftver)
- Mindkettő telepítve van a `<kempelen.inf.bme.hu>-n`
- Mindkettő letölthető a honlapról (linux, Win95/98/NT)
- Webes gyakorló felület készül (ld. honlap)
- Kézikönyvek HTML-változatban (MOSML pdf is)
- Más programok: swiProlog, gnuProlog smlnj
- emacs-szövegszerkesztő SML-, ill. Prolog-módban (linux, Win95/98/NT)

Deklaratív programozás: félévközi követelmények

Nagy házi feladat (NHF)

- Programozás mindkét nyelven (Prolog, SML)
- Mindenkinek önállóan kell kódolnia (programoznia)!
- Hatékony (időlimit!), jól dokumentált („kommentezett”) programok
- A két programhoz közös, 8-10 oldalas fejlesztői dokumentáció (TXT, TeX/LaTeX, HTML, PDF, PS; de nem DOC vagy RTF)
- Kiadás a 4.-5. héten, a honlapon, letölthető keretprogrammal
- Beadás a 13. héten (létraversenyzhez), legkésőbb a vizsgaidőszak első hetében; elektronikus levélben (ld. honlap)
- A beadáskor és a pontozáskor külön-külön teszt sorozatot használunk (nehézségben hasonlókat, de nem azonosakat)
- A minden teszt esetet hibátlanul megoldó programok *létraversengen* vesznek részt (hatékonyság, gyorsaság plusz pontokért)

Deklaratív programozás: félévközi követelmények (folyt.)

Nagy házi feladat (folyt.)

- Nem kötelező, de *nagyon* ajánlott!
- 40%-os szabály (nyelvenként a max. részpontszám 40%-a kell az eredményességhez)
- Súlya az osztályzatban: 15%

Deklaratív programozás: félévközi követelmények (folyt.)

Kis házi feladatok (KHF)

- 2-3 feladat Prologból is, SML-ből is
- Beadás elektronikus levélben (ld. honlap)
- Nem kötelező, de nagyon ajánlott
- Minden feladat jó megoldásáért 1-1 jutalompont

Gyakorló feladatok

- Kötelezők!
- Gyakorlás a honlapon keresztül
- A gyakorlatok megoldását nyilvántartjuk
- Pontot nem adunk
- Pótlási lehetőség a vizsgaidőszak első hetében

Deklaratív programozás: félévközi követelmények (folyt.)

Nagyvárthelyi, pótvárthelyi (NZH, PZH)

- NZH a 13. oktatási héten
- Kötelező!
- Semmilyen jegyzet, segédlet nem használható
- A megtanulandó könyvtári függvények, ill. eljárások listáját előre megadjuk
- 40%-os szabály (nyelvenként a max. részpontszám 40%-a kell az eredményességhez)
- PZH a vizsgaidőszak első hetében
- Súlya az osztályzatban: 15%

Deklaratív programozás: vizsga

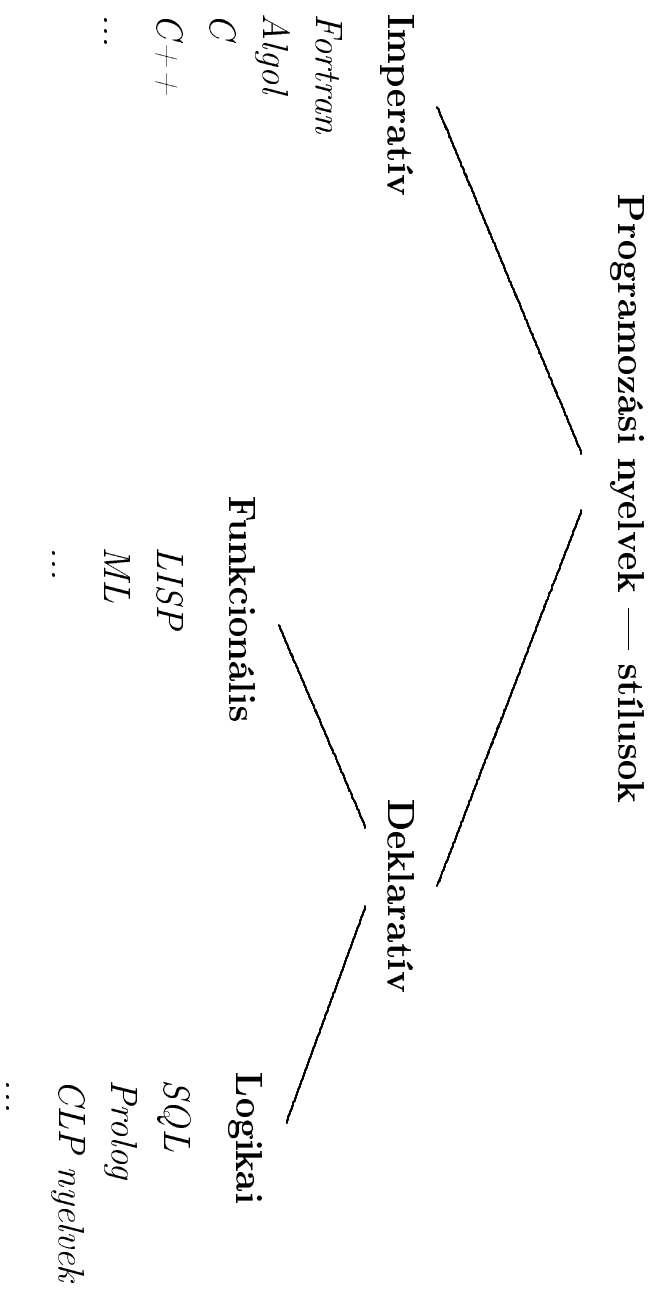
Vizsga

- Szóbeli, felkészülés írásban
- Prolog, SML: több kisebb feladat, kétszer 35 pontért (programírás, -elemzés)
- Semmilyen jegyzet, segédlet nem használható
- A megtanulandó könyvtári függvények, ill. eljárások listáját előre megadjuk
- Ellenőrizzük a nagy házi feladat és a zárthelyi „hitelességét”
- 40%-os szabály (nyelvenként a max. részpontszám 40%-a kell az eredményességhez)
- Korábbi vizsgakérdések a honlapon találhatóak

DEKLARATÍV ÉS IMPERATÍV PROGRAMOZÁS



Programozási nyelvek osztályozása



imperatív és deklaratív programozási nyelvek

imperatív program

- felszólító módú, utasításokból áll
- változó: változtatható értékű memóriahely
- Példa: `int fakt(int n) {int f=1; while (n>1) f*=n-; return f;}`

Deklaratív program

- kijelentő módú, egyenletekből, állításokból áll
- változó: egy ismeretlen, de (előbb–utóbb) rögzített értékű mennyiség
- SML példa: `fun fakt 0 = 1 | fakt n = n * fakt (n-1);`
- C példa: `int fakt(int n) {if (n<=1) return 1; else return n*fakt(n-1);}`

Deklaratív nyelvek jelszavai

- MIT és nem HOGYAN (WHAT rather than HOW): a *megoldás módja* helyett inkább a megoldandó *feladat leírását* kell megadni
- Egyszeres értékadás (single assignment) — párhuzamos végrehajthatóság

Példa — családi kapcsolatok

Adatok

Egy gyerek–szülő kapcsolat, pl.

gyerek	szülő
Imre	István
Imre	Gizella
István	Géza
István	Sarolta
Gizella	Civakodó Henrik
Gizella	Burgundi Gizella

A feladat:

Definiálандó az unoka–nagy­szülő kapcsolat, pl. keressük egy adott személy nagy­szüleit.

A nagyszülő feladat — C nyelvű megoldás

```
/* Az adatbázis */
struct gysz {
    char *gyerek, *szulo;
} szulok[] = {
    "Imre",    "István",
    "Imre",    "Gizella",
    "István",  "Géza",
    "István",  "Sarolt",
    "Gizella", "Civakodó Henrik",
    "Gizella", "Burgundi Gizella",
    NULL,     NULL
};

/* unoka nagyszüleinek kiírása */
void nagyszuloi(char *unoka)
{
    struct gysz *mgysz = szulok;
    for (; mgysz->gyerek; ++mgysz)
        if(!strcmp(unoka, mgysz->gyerek))
        {
            struct gysz *mszn = szulok;
            for (; mszn->gyerek; ++mszn)
                if (!strcmp(mgysz->szulo,
                           mszn->gyerek))
                    puts(mszn->szulo);
        }
}
```

A nagyszülő feladat — SML megoldás

```
* szulei x = az x személy szüleinek listája *)
un szulei "Imre" = ["István", "Gizella"]
| szulei "István" = ["Géza", "Sarolt"]
| szulei "Gizella" = ["Civakodó Henrik", "Burgundi Gizella"]
| szulei _ = [] (* senki másnak nincs szülője *)
> val szulei = fn : string -> string list

* nagyszulei g = g nagyszüleinek listája*)
un nagyszulei g = List.concat (map szulei (szulei g));
> val nagyszulei = fn : string -> string list
```

A függvény futtatása

```
- nagyszulei "Imre";
> val it = ["Géza", "Sarolt", "Civakodó Henrik", "Burgundi Gizella"]
      : string list
```

A nagyszülő feladat — SQL megoldás

```
SQL> create table szulok (gyerek char(30), szulo char(30));  
...)
```

```
SQL> create view nagyszulok as select fiatal.gyerek, oreg.szulo  
2   from szulok fiatal, szulok oreg  
3   where fiatal.szulo = oreg.gyerek;  
view created.
```

```
SQL> select * from nagyszulok;
```

```
GYEREK          SZULO
```

mre	Civakodó Henrik
mre	Burgundi Gizella
mre	Géza
mre	Sarolt

```
SQL>
```


A nagyszülő feladat — Prolog megoldás

```
% szuloje(Gy, Sz): Gy szülője Sz.
szuloje('Imre', 'István').
szuloje('Imre', 'Gizella').
szuloje('István', 'Géza').
szuloje('István', 'Sarolt').
szuloje('Gizella',
        'Civakodó Henrik').
szuloje('Gizella',
        'Burgundi Gizella').

% Gyerek nagyszülője Nagyszulo.
nagyszuloje(Gyerek, Nagyszulo) :-
    szuloje(Gyerek, Szulo),
    szuloje(Szulo, Nagyszulo).
```

```
% Kik Imre nagyszülei?
| ?- nagyszuloje('Imre', NSz).
NSz = 'Géza' ? ;
NSz = 'Sarolt' ? ;
NSz = 'Civakodó Henrik' ? ;
NSz = 'Burgundi Gizella' ? ;
no

% Kik Géza unokái?
| ?- nagyszuloje(U, 'Géza').
U = 'Imre' ? ;
no
```

A deklaratív és imperatív megoldások összehasonlítása

A keresési feladat megoldása

- C nyelven: ciklussal
- SQL-ben: beépített adatbázis-kereséssel
- SML-ben: magasabbrendű függvénybe rejtett rekurzíóval
- Prologban: beépített mintaillesztéses eljáráshívással

Az összetett feltételek kezelése

- C nyelven: skatulyázott ciklussal
- SML-ben: leképezések komponálásával
- SQL-ben, Prologban: relációk konjunkciójának képzésével

A funkcionális és logikai megoldásokról

- az SML megoldás rendkívül tömör (magasabbrendű függvények)
- a Prolog megoldás többirányú (több függvénykapcsolatnak felel meg)

Egy „számológ” program: faktoriális

```

, Prolog megoldás: fakt(N, F) : F = N!.
fakt(0, 1).
          % 0! = 1.
fakt(N, F) :-
          % N! = F, ha
          N > 0,
          % N > 0 és
          N1 is N-1,
          % N1 = N-1 és
          fakt(N1, F1),
          % N1! = F1 és
          F is F1*N.
          % F = N*F1.

```

```

* SML megoldás: fakt n = n! *)
un fakt 0 = 1
| fakt n = n * fakt (n-1)

```

Tanulságok, érdekességek

- SML skatulyázott függvények \Leftrightarrow Prolog egymás mellé rendelt relációk
- Negatív argumentumra a Prolog kód meghiusul, az SML kód végtelen ciklusba esik

Egy összetettebb példa: N-edik generációs ősök

SQL megoldás

```
* szulei_1 lista = a lista-ban szereplő emberek szüleinek listája*)
un szulei_1 ls = List.concat (map szulei ls)
> val szulei_1 = fn : string list -> string list

un nagyszulei1 gy = szulei_1 (szulei gy)
un nagyszulei2 gy = szulei_1 (szulei_1 [gy])

un osei (0, gy) = [gy]
  | osei (n, gy) = szulei_1 (osei (n-1, gy))
> val osei = fn : int * string -> string list

- osei (2, "Imre");
> val it = ["Géza", "Sárolt", "Ciuakodó Henrik", "Burgundi Gizella"]
      : string list
```

N-edik generációs ősök — Prolog megoldás

```

ose(N, E0, E) : E0-nak N-edik generációs őse az E.
**** N adott szám. ****
ose(0, E, E).
ose(N, E0, E) :-
    N > 0, N1 is N-1,
    szuloje(E0, Sz),
    ose(N1, Sz, E).

```

Utása

?- ose(2, 'Imre', Os).		?- ose(2, Utod, 'Burgundi Gizella').
Os = 'Géza' ? ;		Utod = 'Imre' ? ;
Os = 'Sarolt' ? ;		no
Os = 'Civakodó Henrik' ? ;		?- ose(N, 'Imre', Os).
Os = 'Burgundi Gizella' ? ;		N = 0, Os = 'Imre' ? ;
no		{INSTANTIATION ERROR: _157>0 - arg 1}

Nedik generációs ösök — általánosabb Prolog megoldás

$\text{osel}(N, E0, E) : E0\text{-nak } N\text{-edik generációs } \text{őse az } E.$

$$\text{set}(0, E, E).$$
$$\text{set}(N, E_0, E) :-$$

szuloje(E0, Sz), osel(N1, Sz, E), N is N1+1.

Put'asa

```

?- osel(N, 'Imre', Os), N < 2.

```

$$= 0, \text{ Os} = \text{Imre}'; \dots$$

$= 1$, $0s =$ 'István' ? ;

= 1, Os = 'Gizella' ? ;

Or

```
?- osel(I, Utd, 'Burgundi Gizella').
```

= 0, Utod = 'Burgundi Gizella' ? ; ;

$= 2$, $U_{\text{tod}} = \text{'Imre'}$; ? ;

= 1, Utod = 'Gizella' ? ;

01

A funkcionális programozásról dióhéjban

Alapeszme

- a program elemei értékek, speciálisan függvények
 - egy függvény egy kiszámítási szabályt ad meg
 - a program futása: kiértékelés (egyszerűsítés, redukció)

A funkcionális programozás első megvalósítása: LISP

- alapötlet: listák könnyű/hatékony feldolgozása

A funkcionális programozás egy modern megvalósítása: SML

- a függvények „teljes jogú” értékek
- erős típusfogalom, típusok automatikus levezetése

ΣML — előnyök és hátrányok

Miért jó?

- nagyon tömör kód
- függvények is értékek: futási időben létrehozhatók
- mintaillesztés: adatstruktúrák könnyen, áttekinthetően kezelhetők
- erős típusrendszer

Mik a hátrányai?

- megszokottól eltérő programozói stílus

Hogyan tovább?

- lusta kiértékelés (Haskell, Clean)
- párhuzamos végrehajtás (Parallel Haskell, CAML — Concurrent ML)
- típusrendszer bővítése öröklődéssel (Haskell, Clean, Objective CAML)

A logikai programozásról dióhéjban

Alapeszme

- A program elemei logikai állításoknak felelnek meg, pl.:
nagyszuloje(U, N) :- szuloje(U, Sz), szuloje(Sz, N).
matematikai formája:
 $UNNSz(\text{nagyszuloje}(U, N) \leftarrow \text{szuloje}(U, Sz) \wedge \text{szuloje}(Sz, N))$
- A program futása: dedukció (tételbizonyítási folyamat)

A logikai programozás első megvalósítása: a Prolog nyelv

- A logikai állítások egyszerűek, tekinthetők eljárásdefíciónak is
- A tételbizonyítási folyamat értelmezhető mint:
mintaillesztéses eljáráshívás + visszalépéses keresés
- Prolog = RDBMS + rekurzió + adatstruktúrák

Prolog — előnyök és hátrányok

Miért jó?

- tömör kód, többirányú eljárások
- „automatikus” visszalépéses keresés, ciklusok kiváltása
- „logikai” változó — meghatározatlan adatok kezelése

Mik a hátrányai?

- nehéz megtanulni (különösen „tapasztalt” programozóknak)
- rögzített, rugalmatlan vezérlési mechanizmus
- gyenge következtetési képesség

Hogyan tovább?

- CLP — korlát logikai programozás (constraint logic programming)
- annotációk, típusok — Mercury
- rugalmasabb vezérlés, párhuzamos végrehajtás — Aurora, Andorra, Oz

Deklaratív programozás — miért tanítjuk?

Új, magasszintű programozási elemek

- rekurzió
- mintaillesztés
- visszalépéses keresés

Új gondolkodási stílus

- a programrészek (relációk, függvények) önálló jelentéssel bírnak
 - a kód és a jelentés összevethető: program-verifikáció
- Új alkalmazási területek
- szimbolikus alkalmazások
 - következtetési módszerekre épülő megoldások
 - nagyfokú megbízhatóságot igénylő rendszerek

Egy példa: párbeszéd egy 50 soros Prolog programmal

?:- párbeszéd.
/: Magyar legény vagyok én.
Felfogtam.
/: Ki vagyok én?
Magyar legény
/: Péter kicsoda?
Nem tudom.
/: Péter tanuló.
Felfogtam.
/: Péter jó tanuló.
Felfogtam.
/: Péter kicsoda?
Tanuló
jó tanuló
/: Boldog vagyok.
Felfogtam.
/: Te egy Prolog program vagy.
Felfogtam.
/: Ki vagyok én?
Magyar legény
Boldog
/: Okos vagy.
Felfogtam.
/: Te vagy a világ közepe.
Felfogtam.
/: Ki vagy te?
egy Prolog program
Okos
a világ közepe
/: Valóban?
Nem értem.
/: Unlak.
Én is.

BEVEZETÉS A LOGIKAI PROGRAMOZÁSBA



Beevezetés a Logikai Programozásba

Az előadássorozat áttekintése

- Beevezetés
- A Prolog nyelv alapjai
- Prolog programozási módszerek
- A legfontosabb beépített eljárások
- Fejlettebb nyelvi és rendszerelemek
- Prolog programozási példa
- Új irányzatok a logikai programozásban

A Prolog/LP rövid történeti áttekintése

1960-as évek	Tételbizonyító programok
1970-72	A logikai programozás elméleti alapjai (R A Kowalski)
1972	Az első Prolog interpreter (A Colmerauer)
1975	A második Prolog interpreter (Szeredi P)
1977	Az első Prolog fordítóprogram (D H D Warren)
1977-79	Számos kísérleti Prolog alkalmazás Magyarországon
1981	A japán 5. generációs projekt a logikai programozást választja
1982	A magyar MProlog az egyik első kereskedelmi forgalomba kerülő Prolog megvalósítás
1983	Egy új fordítási modell és absztrakt Prolog gép (WAM) megjelenése (D H D Warren)
1986	Prolog szabványosítás kezdete
1987-89	Új logikai programozási nyelvek (CLP, Gödel, stb.)
1990-...	Prolog megjelenése párhuzamos számítógépeken
	Nagyhatékonyságú Prolog fordítóprogramok

Információk a logikai programozásról

Prolog megvalósítások:

- SWI Prolog: <http://www.swi.psy.uva.nl/projects/SWI-Prolog/>
- SICStus Prolog: http://www.sics.se/ps/sicstus/sicstus_toc.html
- GNU Prolog: <http://pauillac.inria.fr/~diaz/gnu-prolog/>

Hálózati információforrások:

The WWW Virtual Library: Logic Programming:

<http://www.comlab.ox.ac.uk/archive/logic-prog.html>

CMU Prolog Repository:

<http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/lang/prolog/0.html>

Prolog FAQ:

<http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/faq/prolog.faq>

Prolog Resource Guide:

[http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/faq/prg_\[12\].faq](http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/faq/prg_[12].faq)

Magyar nyelvű Prolog irodalom

Tarkas Zsuzsa, Futó Iván, Langer Tamás, Szeredi Péter:

Az MPProlog programozási nyelv.

Műszaki Könyvkiadó, 1989

Márkus Zsuzsa: Prologban programozni könnyű.

Novotrade, 1988

Futó Iván (szerk.): Mesterséges intelligencia. (9.2 fejezet, Szeredi Péter)

Aula Kiadó, 1999

A PROLOG NYELV KÖZELÍTŐ SZINTAXISA



Predikátumok, klózek

```
két klózból álló predikátum definíciója, funktora: ose/3
se(0, E, E).
                                %
se(N, E0, E) :-
                                %          fej |
                                % cél    | törzs | 2. klóz, szabály
                                ose(N1, Sz, E), N is N1+1.    % cél, cél | |
```

```
< Prolog program > ::= < predikátum > ...
< predikátum > ::= < klóz > ...          {azonos funktorú}
< klóz > ::= < tényállitás > .⊥ | < szabály > .⊥
< tényállitás > ::= < fej >
< szabály > ::= < fej > :- < törzs >
< törzs > ::= < cél >, ...
< cél > ::= < kifejezés >
< fej > ::= < kifejezés >
```

• Alternatív szóhasználat:

- predikátum — eljárás
- cél — hívás

Prolog kifejezések

```

ose(0, E, E)                % összetett kifejezés, funktora ose/3
| | | |
strukturánév | argumentum, változó
\ - argumentum, számkonstans

```

$\langle \text{kifejezés} \rangle$	$::=$	$\langle \text{változó} \rangle$	$\{\text{var}\}$
		$\langle \text{konstans} \rangle$	$\{\text{atomic}\}$
		$\langle \text{összetett kifejezés} \rangle$	$\{\text{compound}\}$
$\langle \text{konstans} \rangle$	$::=$	$\langle \text{névkonstans} \rangle$	$\{\text{atom}\}$
		$\langle \text{számkonstans} \rangle$	$\{\text{number}\}$
$\langle \text{összetett kifejezés} \rangle$	$::=$	$\langle \text{strukturánév} \rangle$ ($\langle \text{argumentum} \rangle, \dots$)	
$\langle \text{strukturánév} \rangle$	$::=$	$\langle \text{névkonstans} \rangle$	
$\langle \text{argumentum} \rangle$	$::=$	$\langle \text{kifejezés} \rangle$	

- összetett kifejezés funktora = strukturánév/argumentumszám, pl. ose/3
- konstans funktora = konstans/0, pl. 'István'/0
- változónak nincs funktora

Lexikai elemek

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

```
< változó >      < nagybetű > < alfanum > ... |
                 _ < alfanum > ... |
< névkonstans > ::= ' < névkar > ... ' |
                 < kisbetű > < alfanum > ... |
                 < tapadó jel > ... | ! | ; | [] | {}
< névkar >      ::= {tetszőleges nem ' és nem \ karakter} |
                 \ < escape szekvencia >
< alfanum >      ::= < kisbetű > | < nagybetű > | < számjegy > | _
< tapadó jel >   ::= + | - | * | / | \ | $ | ~ | < | > | = | ' | ' | ~ | : | . | ? | @ | # | &
< számkonstans > ::= {előjeles vagy előjeltelen számjegysorozat
                     esetleges tizedes résssel és exponenssel}
```

Szintaktikus édesítőszerek: operátorok

N is $N1+1$ ekvivalens az $is(N, +(N1,1))$ kifejezéssel

Operátor-deklaráció

- $:- op(\langle prioritás \rangle, \langle fajta \rangle, \langle operátornév \rangle).$
- $\langle operátornév \rangle$ tetszőleges névkonstans
- $\langle prioritás \rangle$ 0–1200 közötti egész
- $\langle fajta \rangle$
 - infix: $yfx, xfy, xfx; A \text{ op } B \equiv op(A, B)$
 - prefix: $fx, fy; op \ A \equiv op(A)$
 - postfix: $xf, yf; A \text{ op } \equiv op(A)$
- a $\langle fajta \rangle$ -ban x és y az asszociativitást határozzák meg:
 - x : az adott oldalon nem állhat azonos prioritású operátor zárójellezetlenül
 - y : az adott oldalon állhat azonos prioritású operátor zárójellezetlenül

Beépített operátorok

Szabványos operátorok

```

1200 xfx :-, ->
1200 fx :-, ?-
1100 xfy ;
1050 xfy ->
1000 xfy ', '
900 fy \+
700 xfx < = \= =.. := =< == \==
700 xfx =\= > >= is @< @=< @> @>=
500 yfx + - /\ \/
400 yfx * / // rem mod1 << >>
200 xfx **
200 xfy ^
200 fy -2, \

```

¹ sicutus módban 300 xfx operátor
² sicutus módban 500 fx operátor
³ iso módban 200 fy operátor

Egyéb beépített operátorok

```

1150 fx dynamic multifile
      block meta_predicate
900 fy spy nosp
550 xfy :
500 yfx #
500 fx +3

```

A PROLOG VÉGREHAJTÁSI MECHANIZMUSA



Prolog végrehajtási példa

```

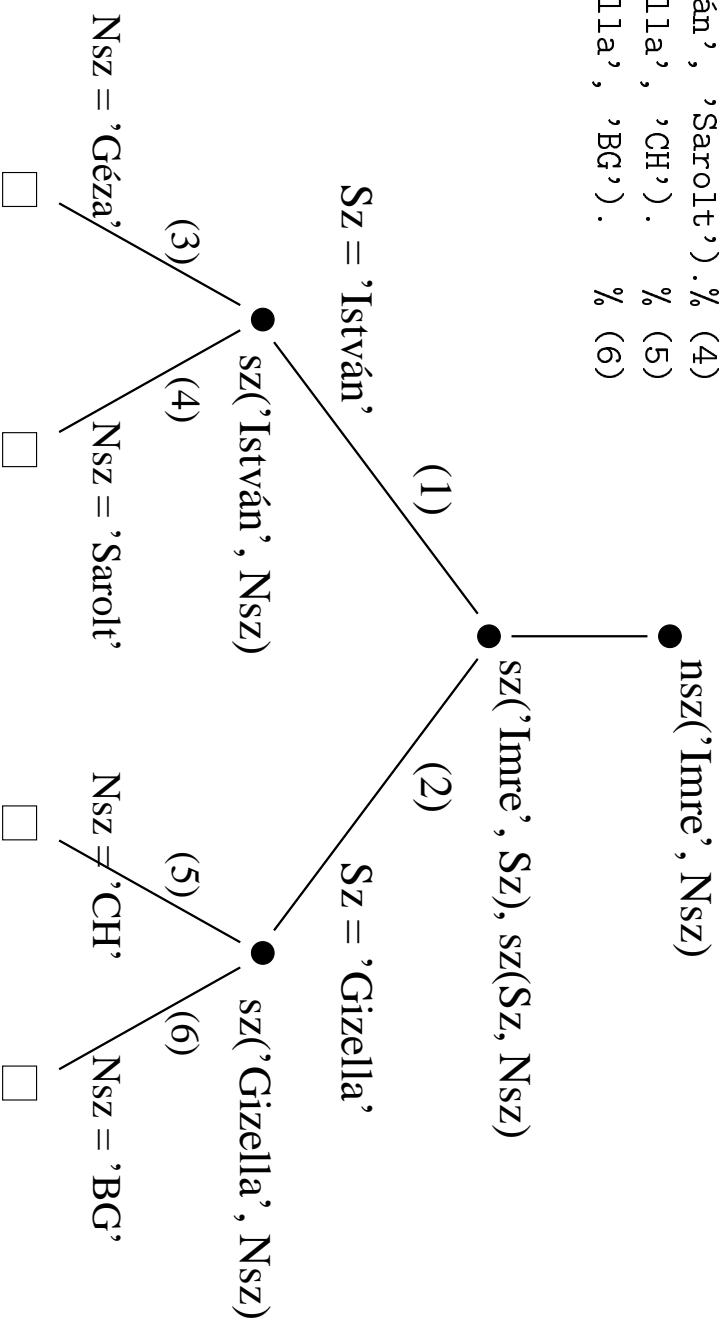
z('Imre', 'István'). % (1)
z('Imre', 'Gizella'). % (2)
z('István', 'Géza'). % (3)
z('István', 'Sarolt').% (4)
z('Gizella', 'CH'). % (5)
z('Gizella', 'BG'). % (6)

```

```

nsz(Gy, N) :-
    sz(Gy, Sz), sz(Sz, N).

```



A végrehajtás alapelemei: egyesítés

Az eljáráshívás és egy klózfej azonos alakra hozása, változók behelyettesítésével

Példák

- Bemenő paraméterátadás:
hívás: `nsz('Imre', Nsz)`,
fej: `nsz(Gy, N)`,
behelyettesítés: `Gy = 'Imre', N = Nsz`
- Kimenő paraméterátadás:
hívás: `sz('Imre', Sz)`,
fej: `sz('Imre', 'István')`,
behelyettesítés: `Sz = 'István'`
- Bemenő/kimenő paraméterátadás:
hívás: `ose(N, 'Imre', Os)`
fej: `ose(0, E, E)`
behelyettesítés: `N = 0, E = 'Imre', Os = 'Imre'`

A végrehajtás alapelemei: redukción lépés

Redukción lépés

- Egy célsorozat (hívássorozat) redukálása egy újabb célsorozattá egy klóz segítségével
- A redukción lépés végrehajtása:
 - A klózt lemásoljuk, minden változót szisztematikusan új változóra cserélve.
 - A célsorozatot szétbontjuk az első hívásra és a maradékra.
 - Az első hívást egyesítjük a klózfejjel
 - Az egyesítéshez szükséges behelyettesítéseket elvégezzük a klóz törzsén és a célsorozatot maradékán
 - Az új célsorozat: a klóztörzs és utána a maradék célsorozat

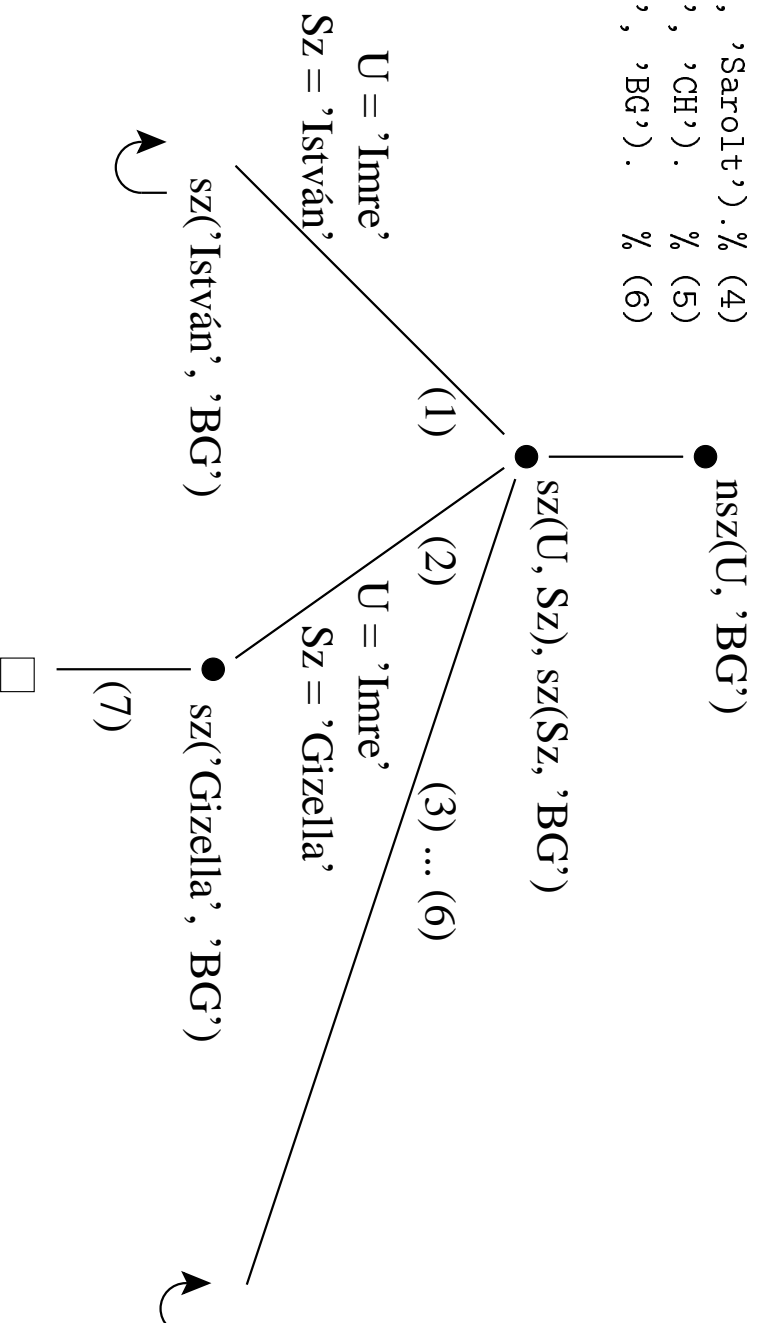
Újabb végrehajtási példa

```

z('Imre', 'István'). % (1)
z('Imre', 'Gizella'). % (2)
z('István', 'Géza'). % (3)
z('István', 'Sarolt'). % (4)
z('Gizella', 'CH'). % (5)
z('Gizella', 'BG'). % (6)

nsz(Gy, N) :-
    sz(Gy, Sz), sz(Sz, N).

```



A Prolog végrehajtási algoritmus

1. *(Kezdeti beállítások:)* A verem üres, $CS := \text{célsorozat}$
2. *(Beépített eljárások:)* Ha CS első célja beépített akkor hajtsuk végre,
 - a. Ha sikertelen \Rightarrow 6. lépés.
 - b. Ha sikeres, elvégezzük a behelyettesítéseket, CS -ből elhagyjuk az első hívást, \Rightarrow 5. lépés.
3. *(Klózszámláló kezdőértékezése:)* $I = 1$.
4. *(Redukciós lépés:)* CS első hívásához tartozó eljárásdefinicióban N klóz van.
 - a. Ha $I > N \Rightarrow$ 6. lépés.
 - b. Redukciós lépés az I -edik klóz és a CS célsorozat között.
 - c. Ha sikertelen, akkor $I := I+1 \Rightarrow$ 4. lépés.
 - d. Ha $I < N$ (nem utolsó), akkor vermeljük $<CS, I>$ -t.
 - e. $CS :=$ a redukciós lépés eredménye
5. *(Siker:)* Ha CS üres, akkor sikeres vég, egyébként \Rightarrow 2. lépés.
6. *(Sikertelenség:)* Ha a verem üres, akkor sikertelen vég.
7. *(Visszalépés:)* Ha a verem nem üres, akkor leermeljük a veremből $<CS, I>$ -t, $I := I+1$, és \Rightarrow 4. lépés.

Előzetes — aritmetikai beépített eljárások

- X is Kif : A Kif aritmetikai kifejezés értékét egyesíti X -szel
- $Kif1 < Kif2$, $Kif1 = < Kif2$, $Kif1 > Kif2$, $Kif1 >= Kif2$, $Kif1 := Kif2$, $Kif1 = \backslash = Kif2$: A $Kif1$ és $Kif2$ aritmetikai kifejezések értéke a megadott relációban van egymással ($= := \Rightarrow$ egyenlő, $= \backslash = \Rightarrow$ nem-egyenlő).

- Ha Kif , $Kif1$, $Kif2$ valamelyike nem aritmetikai kifejezés \Rightarrow hiba.

- Legfontosabb aritmetikai operátorok: $+$, $-$, $*$, $/$, mod , $//$ (egész-osztás)

?- X is $1*2+3$.

= 5 ?

?- X is alma.

DOMAIN ERROR: _78 is alma - arg 2: expected expression, found alma}

?- $X := 1*2+3$.

INITIATION ERROR: _84:=1*2+3 - arg 1}

?- $1+2*3 > 2*3+1$.

o

?-

Előzetes — programfejlesztési beépített eljárások

- `consult(File)` vagy `[File]: A File állományban levő programot beolvassa és értelmezendő alakban eltárolja.` (`File = user` \Rightarrow terminálról olvas.)
- `listing` vagy `listing(Predikátum):` Az értelmezendő alakban eltárolt összes ill. adott nevű predikátumokat kilistázza.
- `compile(File):` A `File` állományban levő programot beolvassa, lefordítja.
- A beolvasott program elemei Prolog kifejezések (':-', ', ', 'operátorok!')
- `halt: A Prolog rendszer befejezi működését.`

`sicstus`

ICSStus 3.8.5 (x86-linux-glibc2.1): Fri Oct 27 10:16:41 CEST 2000

?- `consult(fakt).`

`consulted /home/user/fakt.pl in module user, 10 msec 384 bytes}`

?- `listing(fakt).`

`akt(0, 1).`

`akt(A, B) :-`

`A>0, C is A-1, fakt(C, D), B is D*A.`

?- `halt.`

Előzetes — kiíró és egyéb eljárások

- `write(X)`: Az `X` Prolog kifejezést kiírja (ha kell, operátorokkal).
- `display(X)`: Az `X` Prolog kifejezést struktúra-alakban kiírja.
- `nl`: Kiír egy újsort.
- `true`, `fail`: Mindig sikerül ill. mindig meghiúsul.
- `trace`, `notrace`: `A` (teljes) nyomkövetést be- ill. kikapcsolja.
- `spy` Predikátum: Töréspontot helyez a Predikátum-ra.

```
?- write(+ (1,*(2,3))), write('          '), display(1+2*3), nl.  
+2*3          + (1,*(2,3))
```

és

```
?- szuloje('István', X), write(X), nl, fail.
```

éza

```
arolt
```

o

Prolog végrehajtási példa — rekurzió verem nélkül

```
* (1) */ fakt(0, 1).  
* (2) */ fakt(N, F) :-
```

```
    N>0, NN is N-1, fakt(NN, FF), F is FF*N.
```

- Kezdeti célsorozat: `fakt(2,X)`, `write(X)`
- Redukció (2)-vel: `2>0`, `NN` is `2-1`, `fakt(NN, FF)`, `X` is `FF*2`, `write(X)`
- Beépítettek végrehajtása: `fakt(1, FF)`, `X` is `FF*2`, `write(X)`
- Redukció (2), beép.: `fakt(0, FF1)`, `FF` is `FF1*1`, `X` is `FF*2`, `write(X)` (*)
- Redukció (1)-gyel (választási pont!): `FF` is `1*1`, `X` is `FF*2`, `write(X)`
- Beép.: `write(2)`, mellékhatás: \Rightarrow 2 kiírása, sikeres vég
- További megoldás kérése esetén visszalépés (*)-hoz, redukció (2)-vel: `0>0`, `NN2` is `0-1`, `fakt(NN2, FF2)`, ..., beép. végrehajt., meghívásulás.

Prolog végrehajtási példa — nyomkövetés

?- trace, fakt(2, X), write(X), nl, fail.	
1 1 Call: fakt(2,X) ?	1 1 Redo: fakt(2,2) ?
2 2 Call: 2>0 ?	4 2 Redo: fakt(1,1) ?
2 2 Exit: 2>0 ?	7 3 Redo: fakt(0,1) ?
3 2 Call: NM is 2-1 ?	10 4 Call: 0>0 ?
3 2 Exit: 1 is 2-1 ?	10 4 Fail: 0>0 ?
4 2 Call: fakt(1,FF) ?	7 3 Fail: fakt(0,FF1) ?
5 3 Call: 1>0 ?	4 2 Fail: fakt(1,FF) ?
5 3 Exit: 1>0 ?	1 1 Fail: fakt(2,X) ?
6 3 Call: NM1 is 1-1 ?	
6 3 Exit: 0 is 1-1 ?	no
7 3 Call: fakt(0,FF1) ?	{trace}
7 3 Exit: fakt(0,1) ?	?-
8 3 Call: FF is 1*1 ?	
8 3 Exit: 1 is 1*1 ?	
4 2 Exit: fakt(1,1) ?	
9 2 Call: X is 1*2 ?	
9 2 Exit: 2 is 1*2 ?	
1 1 Exit: fakt(2,2) ?	

Prolog végrehajtás — egy aritmetikai példa

Példa: „jó” számok

Keressük azokat a kétjegyű számokat amelyek négyzete háromjegyű és a szám fordítottjával kezdődik:

```
dec1(J) : J egy pozitív decimális számjegy.  
dec1(1). dec1(2). dec1(3). dec1(4).  
dec1(5). dec1(6). dec1(7). dec1(8). dec1(9).  
  
dec(J) : J egy decimális számjegy.  
dec(0).  
dec(J) :- dec1(J).
```

Szam négyzete háromjegyű és a Szam fordítottjával kezdődik.

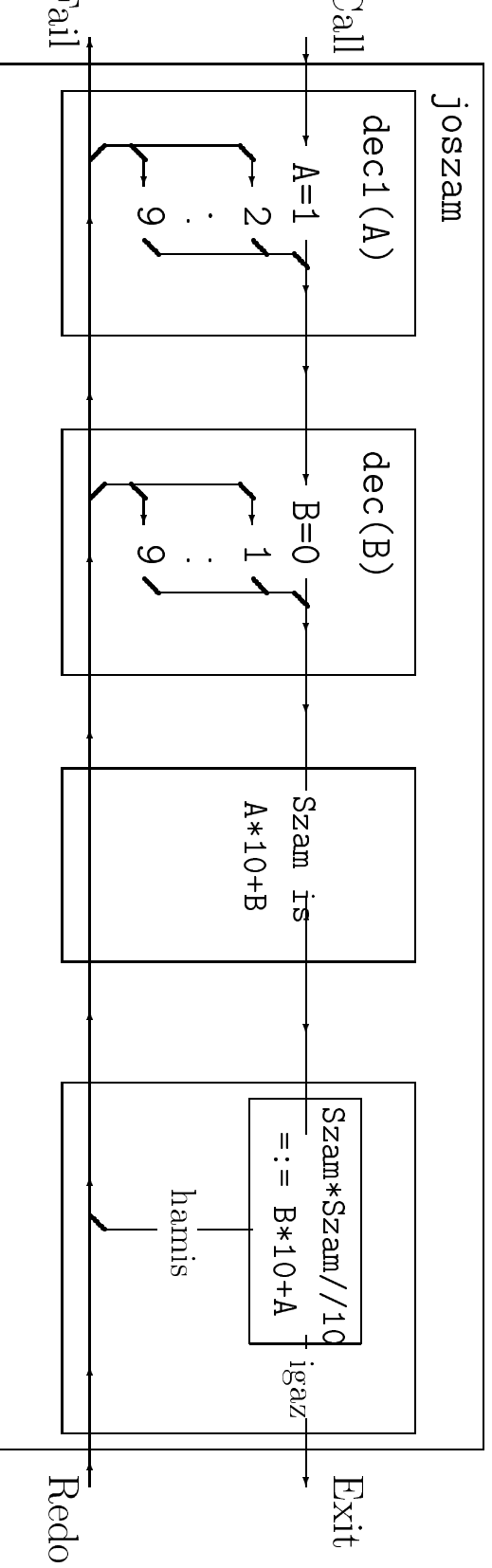
```
oszam(Szam) :-  
    dec1(A), dec(B),  
    Szam is A * 10 + B, Szam * Szam // 10 := B * 10 + A.
```

Prolog végrehajtás — a 4-kapus doboz modell

osszam(Szam) :-

dec1(A), dec(B),

Szam is A * 10 + B, Szam * Szam // 10 := B * 10 + A.



Prolog végrehajtás — számintervallum felsorolása

```
between(M, N, I) : M =< I =< N, I egész.
```

```
between(M, N, M) :-
```

```
    M =< N.
```

```
between(M, N, I) :-
```

```
    M < N,
```

```
    M1 is M+1,
```

```
    between(M1, N, I).
```

```
dec(X) :- between(0, 9, X).
```

```
?- between(1, 2, _X), between(3, 4, _Y), Z is 10*_X+_Y.
```

```
X = 13 ? ;
```

```
X = 14 ? ;
```

```
X = 23 ? ;
```

```
X = 24 ? ;
```

```
no
```

Prolog végrehajtás — diszjunkció mint szemantikus édesítőszert

példa:

Sz-nek gyermeke Gy.

gyermeke(Sz, Gy) :- fia(Sz, Gy).

gyermeke(Sz, Gy) :- lanya(Sz, Gy).

Azonos fejű szabályok összevonhatók egy diszjunkció bevezetésével:

```
gyermeke(Sz, Gy) :-
    (
        fia(Sz, Gy)
        /*vagy:*/;   lanya(Sz, Gy)
    ).
```

A fejek azonossá is tehetők segédvált. és az $X=Y$ beép. eljárás használatával:

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

Az $X = Y$ beépített eljárás definíciója:

```
X = Y: X és Y egyesíthető
      = X.
```

Prolog végrehajtás — diszjunktció kiváltása

Utodnak valódi őse Os.

```
alodi_ose(Utod, Os) :-  
    szuloje(Utod, Szulo),  
    ( Os = Szulo  
    ; valódi_ose(Szulo, Os)  
    ).
```

- Meghatározzuk a diszjunktcióban szereplő változókat: Os, Szulo.
- Meghatározzuk a diszjunktción kívül szereplő változókat: Utod, Os, Szulo.
- A segéd eljárás argumentumai a közös változók: `seged(Szulo, Os)`.
- Minden alternatívájából egy külön klóz lesz a segéd eljárásban.
- A diszjunktciót a segéd eljárás hívásával helyettesítjük.

```
alodi_ose(Utod, Os) :-  
    szuloje(Utod, Szulo), seged(Szulo, Os).
```

```
seged(Szulo, Os) :- Os = Szulo.           % egyszerűsítő: seged(Szulo, Szulo).  
seged(Szulo, Os) :- valódi_ose(Szulo, Os).
```

A PROLOG ADATFOGALMA



Összetett adatstruktúrák — példa

- egy ember egy két mezőből álló rekord: vezetéknév, keresztnév
- egy kétargumentumú összetett kifejezéssel ábrázoljuk
- a struktúránév legyen pl. a mínusz jel (-): $-(VNév, KNév) \equiv VNév-KNév$

```
szuloje(Gy, Sz): Gy szülője Sz.           % keresztneve(E, KN): E keresztneve KN.
szuloje(szabo-laszlo, szabo-gyorgy).      keresztneve(_Nev-Keszt, Keszt).
szuloje(szabo-laszlo, laszlo-amalia).
szuloje(szabo-amalia, szabo-gyorgy).       % vezetékeve(E, VN): E vezetékeve VN.
szuloje(szabo-amalia, laszlo-amalia).      vezetékeve(Nev-_, Nev).
```

```
?- szuloje(E1, E2), keresztneve(E1, K), vezetékeve(E2, K).
   K = laszlo, E1 = szabo-laszlo, E2 = laszlo-amalia ? ;
no
?- szuloje(V1-K1, K1-K2).
   K1 = laszlo, K2 = amalia, V1 = szabo ?
yes
?- szuloje(E1, E2), E1 = _-Nev, E2 = Nev-_.
   E1 = szabo-laszlo, E2 = laszlo-amalia, Nev = laszlo ?
```

Variációk egy témára — a logikai változó fogalma

E keresztnéve megegyezik egyik szülőjének vezetéknévével

```
erdekes(E) :-
    szuloje(E, Sz),
    keresztneve(E, KN),
    vezetekneve(Sz, KN).
```

```
erdekes2(V-K) :-
    szuloje(V-K, K-).
```

```
erdekes3(E) :-
    keresztneve(E, KN),
    vezetekneve(Sz, KN),
    szuloje(E, Sz).
```

```
erdekes4(E) :-
    E = _-K,
    szuloje(E, K-).
```

?- spy szuloje, erdekes3(E).

```
+ 1      1 Call: szuloje(_965-_951,_951-_978) ?
```

Többszörösen összetett adatok — többtagú nevek

- többtagú kereszt- és vezetéknévnek kezelése, pl. Kovacs Eva Maria
- a tagokat egy másik struktúránévvel kapcsoljuk össze, pl. szabo-eva/maria
- ugyanazt a - struktúránevet használjuk: kis-szabo-laszlo, szabo-(eva-maria)

```
reszneve(Osszetett, Resz): Osszetett név része a Resz név.
```

```
eszneve(Nev, Nev).
```

```
eszneve(Elso_, Nev) :-
```

```
    reszneve(Elso, Nev).
```

```
eszneve(_Masodik, Nev) :-
```

```
    reszneve(Masodik, Nev).
```

```
?- reszneve(kis-kovacs-bela, Resz).      | ?- reszneve(szabo-(eva-maria), Resz).
```

```
Resz = kis-kovacs-bela ? ;
```

```
Resz = kis-kovacs ? ;
```

```
Resz = kis ? ;
```

```
Resz = kovacs ? ;
```

```
Resz = bela ? ;
```

```
no
```

```
Resz = szabo-(eva-maria) ? ;
```

```
Resz = szabo ? ;
```

```
Resz = eva-maria ? ;
```

```
Resz = eva ? ;
```

```
Resz = maria ? ;
```

```
no
```

Típusok Prologban

A Prolog nem típusos nyelv, de érdemes meghatározni a kezelt adatok típusát, például az alábbi formális típus-leírással.

```
:- type név1 == {vnév - knév}.      % egy név1 típusú kifejezés az egy - struktúra
                                     % vnév és knév típusú argumentumokkal.
                                     % név1 = {  $v-k$  |  $v \in \text{vnév}, k \in \text{knév}$  }
:- type vnév == atom.              % egy vnév típusú kifejezés az egy atom
:- type knév == atom.              % egy knév is egy atom
:- pred szuloje(név1, név1).      % szuloje mindkét argumentuma név1 típusú.
szuloje(szabo-laszlo, szabo-gyorgy). %...
```

Rekurzívan definiált típusok

```
:- type név2 == atom \/           % név2 az atom vagy egy
    {név2 - név2}.               % két név2-ből álló - /2 struktúra
:- pred reszneve(név2, név2).      % reszneve argumentumai név2 típusúak.
reszneve(Osszetett, Resz): Osszetett név része a Resz név.
reszneve(Nev, Nev).
reszneve(Elso-, Nev) :-          reszneve(Elso, Nev).
reszneve(_Masodik, Nev) :-      reszneve(Masodik, Nev).
```

A Prolog adatfoglalma — az egyesítési algoritmus

A Prolog adatfoglalma: a Prolog kifejezés

- konstans (szám- ill. névkonstans)
- struktúra-kifejezés
- változó („teljes jogú”, struktúra-kifejezések mélyén is lehet)

Adatstruktúrák szétszedése, összerakása: egyesítési algoritmus

- bemenete: két Prolog kifejezés (belső, faststruktúra formában!);
- célja: azon *legáltalánosabb* változó-behelyettesítések meghatározása, amelyekkel a két kifejezés azonos alakra hozható;
- eredménye:
 - siker, változó-behelyettesítések; vagy
 - meghiúsulás (a kifejezések nem hozhatók azonos alakra).

Egyesítés: a behelyettesítés fogalma

A behelyettesítés

- Egy függvény, amely változókhöz kifejezéseket rendel.
- Pl. $\sigma = \{X \leftarrow a, Y \leftarrow s(b, B), Z \leftarrow C\}$ X-hez a-t, Y-hoz s(b, B)-t stb. rendel.
- $K\sigma$: σ alkalmazása K kif.-re, pl. $f(g(Z, h), A, Y)\sigma = f(g(C, h), A, s(b, B))$
- Két behelyettesítés kompozíciója (függvénykompozíció):

$$\sigma \otimes \theta = \{ x \leftarrow x\sigma\theta \mid x \in D(\sigma) \} \cup \{ x \leftarrow x\theta \mid x \in D(\theta) \setminus D(\sigma) \}$$
- σ általánosabb mint θ , ha létezik olyan ρ , hogy $\theta = \sigma \otimes \rho$

Legáltalánosabb egyesítő (*mgu* — most general unifier)

- A és B kifejezések egyesíthetők ha létezik egy olyan σ behelyettesítés, hogy $A\sigma = B\sigma$. Ezt σ behelyettesítést A és B egyesítőjének nevezzük.
- A és B legáltalánosabb egyesítője σ ($mgu(A, B) = \sigma$), ha σ A és B minden egyesítőjénél általánosabb (Tétel: átnevezéstől eltekintve egyértelmű.)

Az egyesítési algoritmus

Az egyesíthetőség eldöntése, $\sigma = mgu(A, B)$ előállítása

1. Ha A és B azonos változók vagy konstansok, akkor $\sigma = \emptyset$.
 2. Egyébként, ha A változó, akkor $\sigma = \{A \leftarrow B\}$.
 3. Egyébként, ha B változó, akkor $\sigma = \{B \leftarrow A\}$.
 4. Egyébként, ha A és B azonos nevű és argumentumszámú összetett kifejezések és argumentum-listáik A_1, \dots, A_N ill. B_1, \dots, B_N , és
 - a. A_1 és B_1 legáltalánosabb egyesítője σ_1 ,
 - b. $A_2\sigma_1$ és $B_2\sigma_1$ legáltalánosabb egyesítője σ_2 ,
 - c. $A_3\sigma_1\sigma_2$ és $B_3\sigma_1\sigma_2$ legáltalánosabb egyesítője σ_3 ,
 - d. \dots
- akkor $\sigma = \sigma_1 \otimes \sigma_2 \otimes \sigma_3 \otimes \dots$

5. Minden más esetben a A és B nem egyesíthető.

Egyesítési példák

$A = \text{ose}(0, E, E), B = \text{ose}(N, \text{'Géza'}, 0s)$

• (4.) A és B neve és argumentumszáma megegyezik

• (a.) $\text{mgu}(0, N) \text{ (3. szerint) } = \{N \leftarrow 0\} = \sigma_1$

• (b.) $\text{mgu}(E\sigma_1, \text{'Géza'}) = \text{mgu}(E, \text{'Géza'}) \text{ (2. szerint) } = \{E \leftarrow \text{'Géza'}\} = \sigma_2$

• (c.) $\text{mgu}(E\sigma_1\sigma_2, 0s) = \text{mgu}(\text{'Géza'}, 0s) \text{ (3. szerint) } = \{0s \leftarrow \text{'Géza'}\} = \sigma_3$

• tehát $\text{mgu}(A, B) = \sigma_1 \otimes \sigma_2 \otimes \sigma_3 = \{N \leftarrow 0, E \leftarrow \text{'Géza'}, 0s \leftarrow \text{'Géza'}\}$

$A = \text{keresztneve}(V\text{-}K, K), B = \text{keresztneve}(E, \text{jozsi})$

• (4.) A és B neve és argumentumszáma megegyezik

• (a.) $\text{mgu}(V\text{-}K, E) \text{ (3. szerint) } = \{E \leftarrow V\text{-}K\} = \sigma_1$

• (b.) $\text{mgu}(K\sigma_1, \text{jozsi}) = \text{mgu}(K, \text{jozsi}) \text{ (2. szerint) } = \{K \leftarrow \text{jozsi}\} = \sigma_2$

• tehát $\text{mgu}(A, B) = \sigma_1 \otimes \sigma_2 = \{E \leftarrow V\text{-}K, K \leftarrow \text{jozsi}\}$

Egyesítési példák a gyakorlatban

?- kis-kovacs-bela = X-Y.

X = kis-kovacs, Y = bela ? ;

no

?- kis-kovacs-bela = kis-X.

no

?- f(X, 3/Y-X, Y) = f(U, B-a, 3).

B = 3/3, U = a, X = a, Y = 3 ?

?- f(f(X), U+2*2) = f(U, f(3)+Z).

U = f(3), X = 3, Z = 2*2 ?

?- ose(0, V-jozsi, szabo-K) . % = ose(0, E, E).

K = jozsi, V = szabo ?

?- keresztneve(szabo-(eva-maria), N-maria) . % = keresztneve(_-K, K).

N = eva ?

Az egyesítés kiegészítése: előfordulás-ellenőrzés (*occurs check*)

Kérdés: X és $s(X)$ egyesíthető-e?

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

?- $X = s(1, X)$.

$X = s(1, s(1, s(1, s(\dots))))$?

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

?- $X = s(X)$, $Y = s(s(Y))$, $X = Y$.

$X = s(s(s(s(s(\dots))))))$, $Y = s(s(s(s(s(\dots))))))$?

Változót tartalmazó kifejezések — végtelen választás veszélye

```
?- keresztneve(E, K).
    E = _A-K ? ; no

?- keresztneve(E, K), szuloje(E, K-K2).
    E = szabo-laszlo, K = laszlo, K2 = amalia ? ; no

?- szuloje(E, K-K2), keresztneve(E, K).
    E = szabo-laszlo, K = laszlo, K2 = amalia ? ; no

?- szuloje(V-K, E), reszneve(E, K).
    V = szabo, K = laszlo, E = laszlo-amalia ? ;
    V = szabo, K = amalia, E = laszlo-amalia ? ; no

?- reszneve(E, K), szuloje(V-K, E).
    V = szabo, K = laszlo, E = laszlo-amalia ? ;
    C
```

rolog interruption (h for help)? a

Execution aborted}

```
?- reszneve(E, K).
    K = E ? ;
    E = K-_A ? ;
    E = K-_A-_B ?
```

PÉLDA — ÜTVONALKERESÉS

Az útvonalkeresési feladat

A feladat: tekintsük (autóbusz)járatok egy halmazát. Mindegyik járáshoz a két végpont és az útvonal hossza van megadva. Írjunk Prolog eljárást, amellyel megállapítható, hogy két pont összeköthető-e pontosan N csatlakozó járatral!

`járat(A, B, H):` Az A és B városok között van járat, és hossza H km.

`árat('Budapest', 'Prága', 515).`

`árat('Budapest', 'Bécs', 245).`

`árat('Bécs', 'Berlin', 635).`

`árat('Bécs', 'Párizs', 1265).`

`útszakasz(A, B, H):` A -ból B -be eljuthatunk egy H úthosszú járatral.

`útszakasz(Kezdet, Cél, H) :-`

`(járat(Kezdet, Cél, H)`

`; járat(Cél, Kezdet, H)`

`).`

Az útvonalkeresési feladat — folytatás

útvonala(N, A, B, H): A és B között van (pontosan)

N szakaszból álló útvonal, amelynek összhossza H .

útvonala($0, \text{Kezdet}, \text{Kezdet}, 0$).

útvonala($N, \text{Kezdet}, \text{Cél}, H$) :-

$N > 0$,

$N1$ is $N-1$,

útszakasz($\text{Kezdet}, \text{Közben}, H1$),

útvonala($N1, \text{Közben}, \text{Cél}, H2$),

H is $H1+H2$.

?- útvonala($2, \text{'Párizs'}, \text{Hová}, H$).

$H = 1900, \text{Hová} = \text{'Berlin'}$? ;

$H = 2530, \text{Hová} = \text{'Párizs'}$? ;

$H = 1510, \text{Hová} = \text{'Budapest'}$? ;

?-

Körmentes út keresése

A körök kizárására gyűjtenünk kell a már érintett városokat. A gyűjtődatastruktúra legyen pl. Honnan-Közben1-Közben2- ...

útvon1_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) :-

 útvon1_2(N, Honnan, Hová, *Honnan*, H).

útvon1_2(N, A, B, K, H): A és B között van pontosan

N szakaszból álló *körmentes*, *K* *elemein át nem menő* H hosszú út.

útvonal_2(0, Hová, Hová, -, 0).

útvonal_2(N, Honnan, Hová, *Kizártak*, H) :-

 N > 0, N1 is N-1,

 útszakasz(Honnan, Közben, H1),

 \+ *reszneve(Kizártak, Közben)*,

 útvon1_2(N1, Közben, Hová, *Kizártak-Közben*, H2),

 H is H1+H2.

A meghívásulások negáció (NF — Negation by Failure)

A \setminus + Hívás beépített meta-eljárás (vö. $\not\vdash$ — nem bizonyítható)

- végrehajtja a Hívás hívást,
- ha Hívás sikeresen fut le, akkor meghívul,
- egyébként (behelyettesítés nélkül) sikerül.
- \setminus + H jelentése: $\neg\exists X(H)$, ahol X a H -ban a *hívás pillanatában* behelyettesíthető változókat jelöli.
- A „zárt világ feltételezése” (CWA) — ami nem bizonyítható, az nem igaz.

```
?- \+ szuloje(szabo-laszlo, X).           -----> no
?- \+ szuloje(szabo-gyorgy, X).           -----> true ?
?- /* T1 testvére T2:*/ szuloje(T1, _A), szuloje(T2, _A), \+ T1 = T2.
    T1 = szabo-laszlo, T2 = szabo-amalia ?
?- \+ X = 1, X = 2.                       -----> no
?- X = 2, \+ X = 1.                       -----> X = 2 ?
```


A példa nyomkövetése

```
?- spy reszneve, útvonal_2(2, 'Párizs', Hová, H).
The debugger will first zip -- showing spyoints (zip)}
Plain spyoint for user:reszneve/2 added, BID=1}

+ 1 1 Call: reszneve('Párizs', 'Bécs') ? 1
+ 1 1 Fail: reszneve('Párizs', 'Bécs') ? 1
+ 7 2 Call: reszneve('Párizs', 'Bécs', 'Berlin') ? 1
+ 8 3 Call: reszneve('Párizs', 'Berlin') ? 1
+ 8 3 Fail: reszneve('Párizs', 'Berlin') ? 1
+ 9 3 Call: reszneve('Bécs', 'Berlin') ? 1
+ 9 3 Fail: reszneve('Bécs', 'Berlin') ? 1
+ 7 2 Fail: reszneve('Párizs', 'Bécs', 'Berlin') ? 1

H = 1900, Hová = 'Berlin' ? ;

+ 13 2 Call: reszneve('Párizs', 'Bécs', 'Párizs') ? 1
+ 14 3 Call: reszneve('Párizs', 'Párizs') ? 1
+ 14 3 Exit: reszneve('Párizs', 'Párizs') ? 1
+ 13 2 Exit: reszneve('Párizs', 'Bécs', 'Párizs') ? 1
+ 16 2 Call: reszneve('Párizs', 'Bécs', 'Budapest') ? n

H = 1510, Hová = 'Budapest' ?
```

Körmentes út keresése — probléma a gyűjtő adatstruktúrával

```

reszneve(Osszetett, Resz): Osszetett név része a Resz név.
eszneve(Nev, Nev).
eszneve(Elso-, Nev) :-      reszneve(Elso, Nev).
eszneve(_-Masodik, Nev) :-  reszneve(Masodik, Nev).

árat(kál, kál-kápolna, 20).
árat(hatvan, kál, 30).

?- útvonal_2(2, hatvan, kál-kápolna, H).
   H = 50 ? ; no

?- útvonal_2(2, kál-kápolna, hatvan, H).
   no

?- spy reszneve, útvonal_2(2, kál-kápolna, hatvan, H).
   + 1 1 Call: reszneve(kál-kápolna,kál) ?
   + 2 2 Call: reszneve(kál,kál) ?
   + 2 2 Exit: reszneve(kál,kál) ?
   + 1 1 Exit: reszneve(kál-kápolna,kál) ?
   0
   ?-
```

Probléma a gyűjtővel — általános gyűjtő-fogalom

```
bfa_resze(Bfa, Resz): Bfa része Resz.  
fa_resze(Bfa, Bfa).  
fa_resze(Balfa_, Resz) :- bfa_resze(Balfa, Resz).  
fa_resze(_-Jobbfa, Resz) :- bfa_resze(Jobbfa, Resz).
```

Milyen típusú adatokat kezel a fenti `bfa_resze`?

- T típusú elemekből épített bináris $fa(bfa(T))$ az
- vagy két ugyanilyen bináris fa - struktúránévvel összekapcsolva $(bfa(T) - bfa(T))$;
- vagy pedig egy T típusú elem;
- formálisabban: $\% :- type\ bfa(T) == \{bfa(T) - bfa(T)\} \setminus T$.
- ez egy *nem megkülönböztetett* únio, mi van ha T funktora - /2?

Probléma a gyűjtő adatstruktúrával — megkülönböztetett úniók

Bináris fa megkülönböztetett únióval

```
:- type bfa(T) == {bfa(T) - bfa(T)} \ / {level(T)}.
```

egy szintaktikus egyszerűsítést bevezetve:

```
:- type bfa(T) ---> bfa(T) - bfa(T) ; level(T).
```

Egy T-kből álló bfa az vagy két ilyenből álló $- / 2$, struktúra vagy egy level/1 struktúrába csomagolt T típusú adat.

Unáris vagyis lineáris fa (nekünk ez is elég)

```
:- type gyujtemeny(T) ---> gyujtemeny(T)-T ; semmi.
```

```
leme(_Gy-E, E).
```

```
leme(Gy-, E) :- eleme(Gy, E).
```

```
?- eleme(semmi-1-2, E).
```

```
    E = 2 ? ; E = 1 ? ; no
```

```
?- eleme(semmi-(kal-kapolna)-hatvan, kal).
```

```
no
```

```
?- eleme(semmi-(kal-kapolna)-hatvan, X).
```

```
    X = hatvan ? ;
```

```
    X = kal-kapolna ? ; no
```

Körmentes út keresése — javított megoldás

útvonat_3(N, A, B, H): A és B között van (pontosan)

N szakaszból álló körmentes útvonal, amelynek összhossza H.

útvonat_3(N, Honnan, Hová, H) :-

 útvonat_3(N, Honnan, Hová, *senmi*-Honnan, H).

útvonat_3(N, A, B, K, H): A és B között van pontosan

N szakaszból álló körmentes, K elemein át nem menő H hosszú út.

útvonat_3(0, Hová, Hová, -, 0).

útvonat_3(N, Honnan, Hová, Kizártak, H) :-

 N > 0, N1 is N-1,

 útszakasz(Honnan, Közben, H1),

 \+ *eleme(Kizártak, Közben)*,

 útvonat_3(N1, Közben, Hová, Kizártak-Közben, H2), H is H1+H2.

?- útvonat_3(2, kál-kápolna, hatvan, H).

H = 50 ? ; no

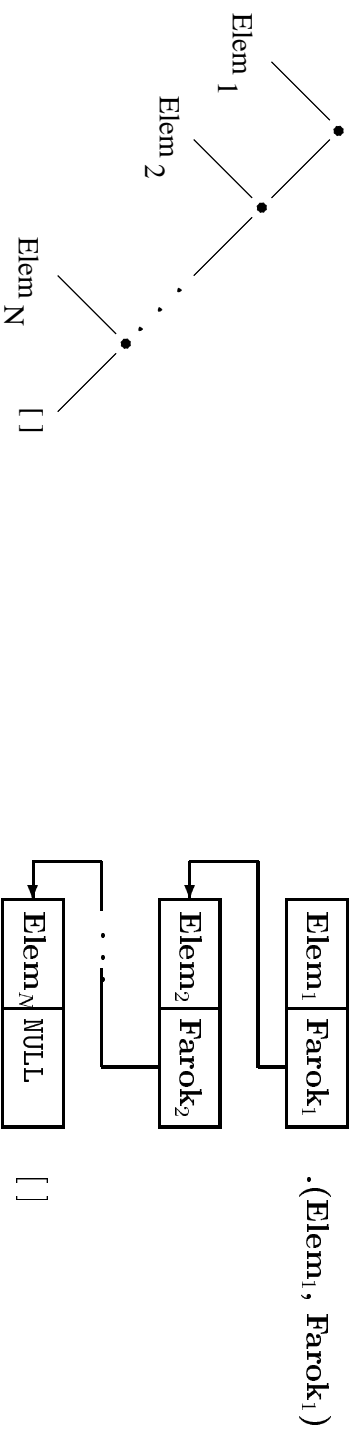
LISTÁK PROLOGBAN



A Prolog lista-fogalma

- közöséges adattípus: `% :- type list(T) ----> .(T, list(T)) ; []`.
- T típusú elemekből álló lista az vagy egy `'./2` struktúra, vagy a `[]` atom. A struktúra első argumentuma T típusú, a lista feje (első eleme). A második argumentum `list(T)` típusú, a lista farka (a többi elemből álló lista);
- egyszerűsített írásmód („szintaktikus édesítés”);
- hatékonyabb megvalósítás.

A listák faststruktúra alakja és megvalósítása



listák jelölése — szintaktikus édesítőszerek

- $[Fej|Farok] \equiv .(Fej, Farok)$
 - $[Elem_1, Elem_2, \dots, Elem_N|Farok] \equiv [Elem_1| [Elem_2| [\dots [Elem_N|Farok] \dots]]]$
 - $[Elem_1, Elem_2, \dots, Elem_N] \equiv [Elem_1, Elem_2, \dots, Elem_N| []]$
- | ?- $[1, 2] = [X|Y].$ $\Rightarrow X = 1, Y = [2] ?$
 - | ?- $[1, 2] = [X, Y].$ $\Rightarrow X = 1, Y = 2 ?$
 - | ?- $[1, 2, 3] = [X|Y].$ $\Rightarrow X = 1, Y = [2, 3] ?$
 - | ?- $[1, 2, 3] = [X, Y].$ $\Rightarrow \text{no}$
 - | ?- $[1, 2, 3, 4] = [X, Y|Z].$ $\Rightarrow X = 1, Y = 2, Z = [3, 4] ?$
 - | ?- $L = [1|_], L = [_ , 2|_].$ $\Rightarrow L = [1, 2|_A] ? \% \text{ nyílt végű}$
 - | ?- $L = .(1, [2, 3| []]).$ $\Rightarrow L = [1, 2, 3] ?$
 - | ?- $L = [1, 2| .(3, [])].$ $\Rightarrow L = [1, 2, 3] ?$
 - | ?- $[X| [3-Y/X|Y]] = .(A, [A-B, 6]).$ $\Rightarrow A=3, B=[6]/3, X=3, Y=[6] ?$

Listaelemek keresése: `member(E, L): E az L lista eleme`

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

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

Eldöntendő kérdés

```
| ?- member(2, [1,2,3]).      => yes
```

Megválaszolandó kérdések

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

Vegyes használat, listák metszete

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

Lista elemévé tesz, végtelen választás!

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

member/2 általánosítása: select/3

```
% select(Elem, Lista, Marad): Elemet a Lista-ból elhagyva marad Marad.
select0(Elem, [Elem|Marad], Marad).    % Elhagyjuk a fejét, marad a fark.
select0(Elem, [X|Farak], Marad) :-
    select0(Elem, Farok, Marad0), % A farkból hagyunk el elemet,
    Marad = [X|Marad0].           % a maradék elé tesszük a fejet.
```

% A második klóz tömörebben (logikai stílusban) --- jobbrekurzív!

```
select(Elem, [Elem|Marad], Marad).
select(Elem, [X|Farak], [X|Marad0]) :-
    select(Elem, Farok, Marad0).
```

```
?- select(X, [1,2,3], L).
```

```
    L=[2,3], X=1 ? ;    L=[1,3], X=2 ? ;    L=[1,2], X=3 ? ; no
```

```
?- select(3, L, [1,2]).
```

```
    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]).
```

```
    no    % a logikai stílusban 1 lépés, a funkcionálisban 10!
```

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

- Tömör (ground) kifejezés: változót nem tartalmazó kifejezés
- Minta: egy általában nem nem tömör kifejezés, mindazon kifejezéseket „képviseeli”, 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 v.	Milyen listákat képvisel	Nyílt v. Milyen listákat képvisel	
[X]	egyelemű	X	tetszőleges
[X, Y]	kételemű	[X Y]	nem üres (legalább 1 elemű)
[X, X]	két egyforma elemből álló	[X, Y Z]	legalább 2 elemű
[X, 1, Y]	3 elemből áll, 2. eleme 1	[a, b Z]	legalább 2 elemű, elemei: a, b, ...

Biztonságos” a futás, azaz véges a keresési tér, ha:

- member/2 második argumentuma zárt végű.
- select/3 2. és 3. argumentuma közül az egyik zárt végű.

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: $L3 = L1 \oplus L2$).

append([], L, L).

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

?- trace, append([1,2], [3,4], L).

```
1 1 Call: append([1,2], [3,4], L) ?
2 2 Call: append([2], [3,4], L3) ? g
```

ncestors:

```
1 1 Call: append([1,2], [3,4], [1|L3])
2 2 Call: append([2], [3,4], L3) ?
3 3 Call: append([], [3,4], L31) ? g
```

ncestors:

```
1 1 Call: append([1,2], [3,4], [1,2|L31])
2 2 Call: append([2], [3,4], [2|L31])
3 3 Call: append([], [3,4], L31) ?
3 3 Exit: append([], [3,4], [3,4]) ?
```

Az append(L1, ...) komplexitása: futási ideje arányos L1 hosszával (ha L1 zárt).

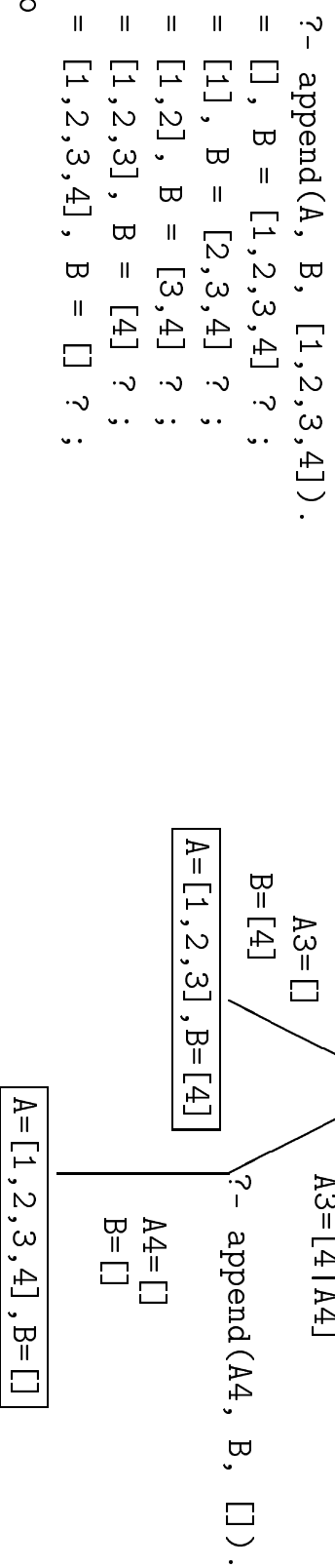
listák szétbontása az append/3 segítségével

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).



Variációk appendre 1. — Három lista összefűzése

$L1 \oplus L2 \oplus L3 = L_{123}$, ahol $L1$ és $L2$ adott.

$\text{append}(L1, L2, L3, L_{123}) :-$

$\text{append}(L1, L2, L_{12}), \text{append}(L_{12}, L3, L_{123}).$

• Nem hatékony, pl.: $\text{append}([1, \dots, 100], [1, 2, 3], [1], L)$ 103 helyett 203 lépés!

• Szétszedésre nem alkalmas — végtelen választási pontot hoz létre
(véges a keresési tér, ha az 1. és 3. argumentum legalább egyike zárt végű lista.)

Öszetszedésre is alkalmas, hatékony változat

$L1 \oplus L2 \oplus L3 = L_{123}$, ahol vagy $L1$ és $L2$ vagy L_{123} adotttá(zárt végű).

$\text{append}(L1, L2, L3, L_{123}) :-$

$\text{append}(L1, L23, L_{123}), \text{append}(L2, L3, L23).$

Az első $\text{append}/3$ hívás nyílt végű listát állít elő:

?- $\text{append}([1, 2], L23, L).$ \Rightarrow $L = [1, 2|L23]$?

Egy érdekes feladvány

Egy szakadékon egy hosszú és keskeny palló ível át, amely egyszerre legfeljebb két embert bír el. A palló egyik oldalán áll négy ember: 10, 20, 50 és 100 évesek. Az N éves ember N perc alatt tud átmenni a hídon. Ha ketten mennek át akkor a lassabb embernek megfelelő idő alatt érnek át.

Höötét van, világítás nélkül lehetetlen átérni, de a társaságnak csak egyetlen seblámpája van. A feladat: megszervezni az átkelést úgy, hogy a teljes ársaság a lehető legrövidebb idő alatt átkeljen a túloldalra.

Kérdés: mennyi idő alatt tudnak leggyorsabban átkelni?

Prolog megoldás: a következő előadáson!

Variációk appendre 2. — lista folytonos része

```
L123 folytonos részlistája L2 (L123 = _ ⊕ L2 ⊕ _).  
L123 adott, L2 ismeretlen.
```

```
sublist(L2, L123) :-  
    append(_L1, L23, L123),  
    append(L2, _L3, L23).
```

Adott L123-nak folytonos része egy adott L2.

```
check_csublist(L2, L123) :-  
    append(L2, _L3, L23),  
    append(_L1, L23, L123).
```

A két változat hatékonyságának összehasonlítása

- L123 = [0, 1, 2, 3, 4, $\underbrace{\dots}_{\times 100000}$, 10], L2 = [0, 1, 2, 3, 4, 10]
csublist(L2, L123): 570 msec
- check_csublist(L2, L123): 430 msec

Mintakeresés append/3-mal

párban előforduló elemek

```
párban(Lista, Elem): A Lista számlistának Elem olyan
    eleme, amely egy ugyanilyen értékű elemmel szomszédos.
    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
```

Listák megfordítása

Naív (négyzetes lépésszámú) megoldás

```
nrev(L, R): Az R lista az L megfordítása.  
rev([], []).  
rev([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 a `member/2`, `select/3`, `append/3` és `reverse/2` eljárások definícióját:

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

append és revapp — listák gyűjtési iránya

• Prolog megvalósítás

<pre>append([], L, L). append([X L1], L2, [X/L3]) :- append(L1, L2, L3).</pre>	<pre>revapp([], L, L). revapp([X L1], L2, L3) :- revapp(L1, [X/L2], L3).</pre>
--	--

• C++ megvalósítás

<pre>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; }</pre>	<pre>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; }</pre>
--	---

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

A select/3 továbbfejlesztése

cserél(X, XL, Y, YL): egy X elemet XL-ben Y-ra cserélve kapjuk YL-t.

Deklaratívabban: X ugyanannyiadik eleme XL-nek, mint Y YL-nek, és a két lista csak ebben az elemben különbözik.

A végességhez XL és YL közül legalább az egyik zárt végű kell legyen.

cserél(X, [X|Xlista], Y, [Y|Ylista]).

cserél(X, [Head|Xlista], Y, [Head|Ylista]) :-

 cserél(X, Xlista, Y, Ylista).

```
?- cserél(2, [1,2,3], 4, L).
```

```
    L = [1,4,3] ? ;
```

```
no
```

```
?- cserél(X, [1,2,3], 4, L).
```

```
    L = [4,2,3], X = 1 ? ;
```

```
    L = [1,4,3], X = 2 ? ;
```

```
    L = [1,2,4], X = 3 ? ;
```

```
no
```

```
?- cserél(X, [1,2,3], Y, L).
```

```
    L = [Y,2,3], X = 1 ? ;
```

```
    L = [1,Y,3], X = 2 ? ;
```

```
    L = [1,2,Y], X = 3 ? ;
```

```
no
```

2000 tavaszi kis házi feladat

Állítsa elő egy Sz nem negatív egész szám A alapú számrendszerben vett jegyeinek listáját ($A > 1$ egész)! Írjon egy szám/3 Prolog eljárást, amely a legnagyobb helyiértékű jegyet helyezi a lista elejére, és egy másik fszám/3 eljárást, amely a legkisebb helyiértékű jeggyel kezdi a listát.

szám(Szám, Alap, Jk): A Szám szám Alap alapú számrendszerben vett jegyeinek (balról jobbra haladó) listája Jk. (A 0 szám egy jegyből áll.)
 szám(0, –, [0]).

szám(Sz, Alap, Jk) :-

$Sz > 0$, szám(Sz, Alap, [] , Jk).

szám(Szám, Alap, Jk0, Jk): A Szám szám Alap alapú számrendszerben vett jegyeinek listáját Jk0 elé fűzve kapjuk Jk-t (A 0 jegylistája üres).

Jelölés: $LL = L-L0 <--->$ az LL listát $L0$ elé fűzve kapjuk L -t.

szám(0, –, Jk, Jk).

szám(Sz, Alap, Jk0, Jk) :-

$Sz > 0$, Sz1 is Sz//Alap, UtsoJegy is Sz mod Alap,

szám(Sz1, Alap, [UtsoJegy|Jk0] , Jk).

2000 tavaszi kis házi feladat — számjegyek fordított sorrendben

$\text{fiszám}(\text{Sz}, \text{A}, \text{Jk})$: Az Sz szám A alapú fordított jegylistája Jk.

$\text{szám}(0, -, [0])$.

$\text{szám}(\text{Sz}, \text{Alap}, \text{Jk}) :- \text{Sz} > 0, \text{fiszám}(\text{Sz}, \text{Alap}, [], \text{Jk})$.

$\text{fiszám}(\text{Sz}, \text{A}, \text{Jk0}, \text{Jk})$: Az Sz szám A alapú fordított jegylistája Jk-Jk0.

$\text{szám}(0, -, \text{Jk}, \text{Jk})$.

$\text{szám}(\text{Sz}, \text{Alap}, \text{Jk0}, [\text{UtsoJegy}|\text{Jk}]) :-$

$\text{Sz} > 0, \text{Sz1 is Sz} // \text{Alap}, \text{UtsoJegy is Sz mod Alap},$

$\text{fiszám}(\text{Sz1}, \text{Alap}, \text{Jk0}, \text{Jk})$.

A kétféle irányú gyűjtés összehasonlítása

$\text{szám}(0, -, \text{Jk}, \text{Jk})$.

$\text{szám}(0, -, \text{Jk}, \text{Jk})$.

$\text{szám}(\text{Sz}, \text{A}, \text{Jk0}, [\text{U}|\text{Jk}]) :-$

$\text{szám}(\text{Sz}, \text{A}, \text{Jk0}, \text{Jk}) :-$

$\text{Sz} > 0, \text{Sz1 is} \dots,$

$\text{Sz} > 0, \text{Sz1 is} \dots,$

$\text{U is} \dots,$

$\text{U is} \dots,$

$\text{fiszám}(\text{Sz1}, \text{A}, \text{Jk0}, \text{Jk})$.

$\text{szám}(\text{Sz1}, \text{A}, [\text{U}|\text{Jk0}], \text{Jk})$.

2000 tavaszi kis házi feladat — egyszerűsítés

szám/3 egyszerűsíthető

- `fszám/4` minden hívása `fszám(_, _, [], _)` alakú.

- `fszám(Sz, A, [], Jk) ⇒ fszám12(Sz, A, Jk)`

`fszám(Szám, Alap, Jegyek): A Szám >= 0 szám Alap > 1 alapú`

`számrendszerben jobbról balra vett jegyeinek listája Jegyek.`

`szám1(0, -, [0]).`

`szám1(Sz, Alap, Jk) :-`

`Sz > 0, fszám12(Sz, Alap, Jk).`

`fszám12(Szám, Alap, Jk): A Szám >= 0 szám Alap > 1 alapú`

`számrendszerben jobbról balra vett jegyeinek listája Jk.`

`szám12(0, -, []).`

`szám12(Sz, Alap, [UtsoJegy|Jk]) :-`

`Sz > 0, Sz1 is Sz//Alap, UtsoJegy is Sz mod Alap,`

`fszám12(Sz1, Alap, Jk).`

Körmentes út keresése — megoldás listák használatával

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

útvonal_4(N, A, B, Út, H): A és B között van (pontosan)
N szakaszból álló körmentes Út útvonal, amelynek összhossza H.
tvonal_4(N, Honnan, Hová, Út, H) :-
    útvonal_4(N, Honnan, Hová, [Honnan], Út, H).

útvonal_4(N, A, B, K, Út, H): A és B között van pontosan
N szakaszból álló körmentes, K elemein át nem menő H hosszú Út út.
tvonal_4(0, Hová, Hová, Kizártak, Út, 0) :-
    reverse(Kizártak, Út).
tvonal_4(N, Honnan, Hová, Kizártak, Út, H) :-
    N > 0, N1 is N-1,
    útszakasz(Honnan, Közben, H1),
    \+ member(Közben, Kizártak),
    útvonal_4(N1, Közben, Hová, [Közben|Kizártak], Út, H2), H is H1+H2.

?- útvonal_4(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

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

smétlődésmentes útvonal keresése listával ábrázolt gráfban

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

útvonal_5(N, G, A, B, L, H): A G gráfban van egy A-ból
B-be menő N szakaszból álló L út, melynek összhossza H.
tvonal_5(0, _Gráf, Hová, Hová, [Hová], 0).
tvonal_5(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_5(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.
1_végpontok_hossz(él(A,B,H), A, B, H).
1_végpontok_hossz(él(A,B,H), B, A, H).

?- hálózat(_Gráf), útvonal_5(2, _Gráf, 'Budapest', -, Út, H).
H = 880, Út = ['Budapest', 'Bécs', 'Berlin'] ? ;
H = 1510, Út = ['Budapest', 'Bécs', 'Párizs'] ? ;
no
```

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*.
 - *Célt* lefuttatja, és a változó-behelyettesítéseket kiírja.
 - *parancs*:
:- *Cél*.
- A *Célt* csendben lefuttatja. Különféle deklarációkat parancsként helyezhetünk el a programban.
- *szabály*:
 $Fej :- Törzs.$
A szabályt felveszi a programba.
- *nyelvtani szabály*:
 $Fej \rightarrow Törzs.$
Prolog szabályt alakítja és felveszi (lásd a DCG nyelvtanokat).
- *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 ismertettett eljárások hatása lényegében nem változik.

Kifejezések szintaxisa — kétszintű nyelvtanok

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

$$\begin{aligned}
 \langle \text{kifejezés} \rangle &::= && \langle \text{tag} \rangle \\
 &| \langle \text{kifejezés} \rangle \langle \text{additív művelet} \rangle \langle \text{tag} \rangle \\
 \langle \text{tag} \rangle &::= && \langle \text{tényező} \rangle \\
 &| \langle \text{tag} \rangle \langle \text{multiplikatív művelet} \rangle \langle \text{tényező} \rangle \\
 \langle \text{tényező} \rangle &::= && \langle \text{szám} \rangle | \langle \text{azonosító} \rangle | (\langle \text{kifejezés} \rangle)
 \end{aligned}$$

- Ugyanez kétszintű nyelvtannal:

$$\begin{aligned}
 \langle \text{kifejezés} \rangle &::= && \langle \text{kif } 2 \rangle \\
 \langle \text{kif } N \rangle &::= && \langle \text{kif } N-1 \rangle \\
 &| \langle \text{kif } N \rangle \langle N \text{ prioritású művelet} \rangle \langle \text{kif } N-1 \rangle \\
 \langle \text{kif } 0 \rangle &::= && \langle \text{szám} \rangle | \langle \text{azonosító} \rangle | (\langle \text{kif } 2 \rangle) \\
 \{ \text{az additív ill. multiplikatív műveletek prioritása } 2 \text{ ill. } 1 \}
 \end{aligned}$$

Kifejezések szintaxisa

$\langle \text{programelem} \rangle ::=$	$\langle \text{kifejezés } 1200 \rangle \mid \langle \text{záró-pont} \rangle$
$\langle \text{kifejezés } N \rangle ::=$	$\langle \text{op } N \text{ fx} \rangle \mid \langle \text{köz} \rangle \mid \langle \text{kifejezés } N-1 \rangle$ $\mid \langle \text{op } N \text{ fy} \rangle \mid \langle \text{köz} \rangle \mid \langle \text{kifejezés } N \rangle$ $\mid \langle \text{kifejezés } N-1 \rangle \mid \langle \text{op } N \text{ xfx} \rangle \mid \langle \text{kifejezés } N-1 \rangle$ $\mid \langle \text{kifejezés } N-1 \rangle \mid \langle \text{op } N \text{ xfy} \rangle \mid \langle \text{kifejezés } N \rangle$ $\mid \langle \text{kifejezés } N \rangle \mid \langle \text{op } N \text{ yfx} \rangle \mid \langle \text{kifejezés } N-1 \rangle$ $\mid \langle \text{kifejezés } N-1 \rangle \mid \langle \text{op } N \text{ xf} \rangle$ $\mid \langle \text{kifejezés } N \rangle \mid \langle \text{op } N \text{ yf} \rangle$ $\mid \langle \text{kifejezés } N-1 \rangle$
$\langle \text{kifejezés } 1000 \rangle ::=$	$\langle \text{kifejezés } 999 \rangle \mid \langle \text{kifejezés } 1000 \rangle$
$\langle \text{kifejezés } 0 \rangle ::=$	$\langle \text{név} \rangle \mid \langle \text{argumentumok} \rangle$ $\mid \{ A \mid \langle \text{név} \rangle \text{ és a } \langle \text{közvetlenül egymás után áll!} \rangle \}$ $\mid \langle \text{kifejezés } 1200 \rangle \mid \{ \langle \text{kifejezés } 1200 \rangle \}$ $\mid \langle \text{lista} \rangle \mid \langle \text{füzér} \rangle$ $\mid \langle \text{név} \rangle \mid \langle \text{szám} \rangle \mid \langle \text{változó} \rangle$

Kifejezések szintaxisa — folytatás

$\langle \text{op } N \ T \rangle ::=$	$\langle \text{név} \rangle \{ \text{feltéve, hogy } \langle \text{név} \rangle \ N \text{ prioritású és } T \text{ típusú operátornak lett deklarálva} \}$
$\langle \text{argumentumok} \rangle ::=$	$\langle \text{kifejezés } 999 \rangle$ $\langle \text{kifejezés } 999 \rangle , \langle \text{argumentumok} \rangle$
$\langle \text{lista} \rangle ::=$	$[]$ $[\langle \text{listakif} \rangle]$
$\langle \text{listakif} \rangle ::=$	$\langle \text{kifejezés } 999 \rangle$ $\langle \text{kifejezés } 999 \rangle , \langle \text{listakif} \rangle$ $\langle \text{kifejezés } 999 \rangle \mid \langle \text{kifejezés } 999 \rangle$
$\langle \text{szám} \rangle ::=$	$\langle \text{előjeltelen szám} \rangle$ $+ \langle \text{előjeltelen szám} \rangle$ $- \langle \text{előjeltelen szám} \rangle$
$\langle \text{előjeltelen szám} \rangle ::=$	$\langle \text{természetes szám} \rangle$ $\langle \text{lebegőpontos szám} \rangle$

Kifejezések szintaxisa — megjegyzések

- A \langle kifejezés $N \rangle$ -ben \langle köz \rangle csak akkor kell ha az öt követő kifejezés nyitó-zárójellel kezdődik.
- A $\{ \langle$ kifejezés $\rangle \}$ azonos a $\{ \{ \langle$ kifejezés $\rangle \}$ struktúrával, ez pl a DCG nyelvtanoknál hasznos.
- Egy \langle füzér \rangle " jelek közé zárt karaktersorozat, általában a karakterek kódjainak listájával azonos.

?- op(500, fx, succ).

es

?- display(succ(1,2)), nl, display(succ(1,2)).

ucc(, (1,2))

ucc(1,2)

es

?- 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 \rangle

- (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 \rangle

- 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;
- prefix operátor és (közé kötelező;
- < záró-pont >: egy . karakter amit egy formázó elem követ.

TÍPUSOK PROLOGBAN



Típusok leírása Prologban

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

$$\{ \text{str}(T_1, \dots, T_n) \} \equiv \{ \text{str}(e_1, \dots, e_n) \mid e_1 \in T_1, \dots, e_n \in T_n \}, n \geq 0$$

$$\{ \text{személy}(\text{atom}, \text{atom}, \text{int}) \}$$
 az olyan személy/3 funktorú struktúrák halmaza, amelyben az első két argumentum `atom`, a harmadik egész.

- Típusok, mint halmazok únója képezhető a `\` operátorral.

$$\{ \text{személy}(\text{atom}, \text{atom}, \text{int}) \} \setminus \{ \text{atom-atom} \} \setminus \text{atom}$$

- Egy típusleírás elnevezhető (kommentben): `% :- type tnév == tleírás.`

`% :- type t1 == {atom-atom} \setminus atom, % :- type ember == {ember-atom} \setminus {semmi}.`

- Megkülönböztetett únó: csupa különböző funktorú összetett típus únója. Egyszerűsített jelölés:

$$:- \text{type } T == \{ S_1 \} \setminus \dots \setminus \{ S_n \}. \Rightarrow :- \text{type } T \dashrightarrow S_1 ; \dots ; S_n.$$

$$\% :- \text{type ember} \dashrightarrow \text{ember-atom ; semmi.}$$

$$\% :- \text{type egészlista} \dashrightarrow [] ; [\text{int egészlista}].$$

Típusok leírása Prologban — folytatás

Paraméteres típusok — példák

```
% :- type list(T) ---> [] ; [T|list(T)]. % T típusú elemekből álló lista.
% :- type pair(T1, T2) ---> T1 - T2.    % egy '-' nevű kétargumentumú struktúra,
                                         % első argumentuma T1, a második T2 típusú.
% :- type assoc_list(KeyT, ValueT)
%      == list(pair(KeyT, ValueT)). % KeyT és ValueT típusú párokból álló lista.
% :- 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 > ::= < típusnév > | < típusváltozó > |
                  < típusleírás > \ / < típusleírás > |
                  { < típusnév > (< típusleírás >, ...) }
< típusazonosító > ::= < típusnév > | < típusváltozó >, ...)
```

Predikátum-deklarációk

- Predikátumtípus-deklaráció

```
:- pred < eljárásnév > (< típusazonosító >, ...)
```

- Példák:

```
:- pred member(T, list(T)).
:- pred append(list(T), list(T), list(T)).
```

- Predikátummód-deklaráció (Nem kötelező, több is megadható.)

```
:- mode < eljárásnév > (< módazonosító >, ...) ahol < módazonosító > ::= in | out.
```

- Példák:

```
:- mode append(in, in, in). % ellenőrzésre
:- mode append(in, in, out). % két lista összefűzésére
:- mode append(out, out, in). % egy lista szétszedésére
```

- Vegyes típus- és móddeklaráció

```
:- pred < eljárásnév > (< típusazonosító > :: < módazonosító >, ...)
```

- Példa:

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


A HÍD FELADVÁNY



A feladat

Egy szakadékon egy hosszú és keskeny palló ível át, amely egyszerre legfeljebb két embert bír el. A palló egyik oldalán áll négy ember: 10, 20, 50 és 100 évesek. Az N éves ember N perc alatt tud átmenni a hídon. Ha ketten mennek át akkor a lassabb embernek megfelelő idő alatt érnek át.

Ötötét van, világítás nélkül lehetetlen átérni, de a társaságnak csak egyetlen seblámpája van. A feladat: megszervezni az átkelést úgy, hogy a teljes társaság a lehető legrövidebb idő alatt átkeljen a túloldalra.

Kérdés: mennyi idő alatt tudnak leggyorsabban átkelni?

Az adatstruktúrák

```
-- type állapot ---> lámpa-list(ember).    % Hol a lámpa és hol vannak az emberek?
-- type lámpa == oldal.                    % A lámpa az egyik oldalon lehet.
-- type ember == list(pair(int,oldal)).      % Minden adott korú ember mellett
                                              % ott a tartózkodási helye.

-- type oldal ---> bal ; jobb.
```

Az állapotátmenet

```

útszakasz(Áll0, Áll1, Idő): Áll0-ból egy lépésben Áll1-be lehet jutni Idő alatt.
:- pred útszakasz(áll::in, áll::out, int::out).

tszakasz(Innen-Helyzet0, Ide-Helyzet, Idő) :-
    másik(Innen, Ide),      % A lámpa az Innen oldalon van, Ide a túloldal.
    cserél(E1-Innen, Helyzet0, E1-Ide, Helyzet),      % E1 megy át
    Idő = E1.

tszakasz(Innen-Helyzet0, Ide-Helyzet, Idő) :-
    másik(Innen, Ide),
    cserél(E1-Innen, Helyzet0, E1-Ide, Helyzet1),      % E1 átmegy
    cserél(E2-Innen, Helyzet1, E2-Ide, Helyzet),      % E2 átmegy
    E1 > E2, Idő = E1.

másik(Egyik, Másik): Egyik és Másik két különböző oldal.
:- pred másik(oldal, oldal).
    másik(bal, jobb).
    másik(jobb, bal).

véghelyzet(Ho1, Áll): Az Áll állapotban a zseblámpa és az emberek a Ho1
    oldalon vannak.
:- pred véghelyzet(oldal, áll).
    éghelyzet(Ho1, Ho1-[10-Ho1,20-Ho1,50-Ho1,100-Ho1]).

```

Intuitív

Át lehet menni a bal véghelyzetből a jobb véghelyzetbe
Hossz Idő alatt az Út állapotlistán keresztül.

```
:- pred átmegy(list(ál1)::out, int::out).
    atmegy(Út, Hossz) :-
        véghelyzet(bal, Kezd),
        véghelyzet(jobb, Vég),
        between(1, 10, N),
        útvonala_4(N, Kezd, Vég, Út, Hossz).
```

```
?- átmegy(Út, H).
    H = 190, Út = [bal-[10-bal,20-bal,50-bal,100-bal],... - ...]] ?
es
?- átmegy(Út, H), H < 190.
    H = 170, Út = [bal-[10-bal,20-bal,50-bal,100-bal],... - ...]] ?
es
?- átmegy(Út, H), H < 170.
    no
    % mintegy 20 másodperc után!
```

2. megoldás — a keresési tér korlátozása, lépések gyűjtése

```
:- type lépés ---> átmenők>oldal.      % az adott oldalra átmennek az átmenők.
:- type átmenők == int \ / {int+int}.  % egy vagy két adott korú ember.

útszakasz(Áll0, Áll1, Idő, Lépés): Egy Lépés lépéssel Idő alatt az Áll0
állapotból az Áll1 állapotba lehet jutni.
:- pred útszakasz(áll::in, áll::out, int::out, lépés::out).
tszakasz(Innen-Helyzet0, Ide-Helyzet, Idő, E1>Ide) :-      % E1 megy át.
    másik(Innen, Ide),
    cserél(E1-Innen, Helyzet0, E1-Ide, Helyzet),          % E1 átmegy
    Idő = E1.
tszakasz(Innen-Helyzet0, Ide-Helyzet, Idő, E1+E2>Ide) :-  % E1+E2 megy át.
    másik(Innen, Ide),
    cserél(E1-Innen, Helyzet0, E1-Ide, Helyzet1),         % E1 átmegy
    cserél(E2-Innen, Helyzet1, E2-Ide, Helyzet),          % E2 átmegy
    E1 > E2, Idő = E1.
```

Lépések-kei át lehet menni a bal véghezlyzetből a jobb véghezlyzetbe

Hossz idő alatt, ahol Hossz < Max.

```
:- pred átmegy(int::in, list(lépés)::out, int::out).
```

```
tmegy(Max, Lépések, Hossz) :-
```

```
    véghezlyzet(bal, Kezd), véghezlyzet(jobb, Vég),
```

```
    útvonat_6(Kezd, Vég, Max, Lépések, Nyereség),
```

```
    Hossz is Max - Nyereség.
```

```
útvonat_6(A, B, Max, Lk, Nyer): A és B között van egy Max-nál
```

```
Nyer-rel rövidebb út (Nyer > 0), amelynek lépéssorozata Lk.
```

```
:- pred útvonat_6(áll::in, áll::in, int::in, list(lépés)::out, int::out).
```

```
tvonat_6(Hová, Hová, Max, [], Max) :- Max > 0.
```

```
tvonat_6(Honnan, Hová, Max0, [Lép|Lépl], Nyer) :-
```

```
    Max0 > 0,
```

```
    útszakasz(Honnan, Közben, H1, Lépl),
```

```
    Max1 is Max0-H1,
```

```
    útvonat_6(Közben, Hová, Max1, Lépl, Nyer).
```

```
?- átmegy(190, Lk, H).
```

```
H = 170, Lk = [20+10>jobb,10>bal,100+50>jobb,20>bal,20+10>jobb] ? ;
```

```
H = 170, Lk = [20+10>jobb,20>bal,100+50>jobb,10>bal,20+10>jobb] ? ;
```

```
no
```

1. kis házi feladat

Adott egy lista, amelynek elemei piros/1, fehér/1 vagy zöld/1 funktorú struktúrák, tetszőleges sorrendben. A struktúrák argumentuma tetszőleges Prolog kifejezés lehet, ezek az argumentumok a feladat szempontjából irredektelenek.

A feladat a lista rendezése úgy, hogy az elején álljanak a piros/1 funktorúak, utánuk fehér/1 funktorúak, végül pedig a zöld/1 funktorúak. Az egyes csoportokon belül az elemek sorrendje ne változzék.

Adjon egy olyan zasz1o/2 Prolog eljárást, amely megvalósítja a leírt rendezést. Ha a fenti háromtól különböző funktorú elem van a listában, akkor az eljárás kiusúljon meg.

Törekedjék arra, hogy a megoldás hatékony legyen! Vigyázzon arra is, hogy az eljárás ne sikerüljön többször!

Pontérték: 1 plusz pont

Beadási határidő: 2001 március 19, 24:00

Beadás módja: a honlapon meghirdetendő módon.

1. kis házi feladat — folytatás

```
:- lista == list(szin).
:- szín ---> piros(any) ; fehér(any) ; zold(any).
zaszlo(Bemenet, Kimenet): a Kimenet lista a Bemenet lista elemeinek
fent leírt módon rendezett listája.
:- pred zaszlo(lista::in, lista::out).
```

Példák

```
?- zaszlo([piros(a),kek(b)], Z).
no

?- zaszlo([zold(c),piros(d),feher(e)], Z).
Z = [piros(d),feher(e),zold(c)] ? ;
no

?- zaszlo([piros(f),zold(g),piros(h),feher(i)], Z).
Z = [piros(f),piros(h),feher(i),zold(g)] ? ;
no
```


BEVEZETÉS A FUNKCIONÁLIS PROGRAMOZÁSBÁ



Az előadássorozat áttekintése

- Bevezetés. Az SML nyelv alapjai.
- Egyszerű és összetett adattípusok. Programfejlesztés.
- Polimorfizmus. Listaműveletek. A legfontosabb programkönyvtárak.
- Programhelyesség, programbizonyítás.
- Magasabbrendű függvények.
- Modulok. Absztrakt adattípusok. Paraméterezhető modulok.
- Nemlineáris rekurzív adattípusok.
- Nagyobb SML-példák.
- Új irányzatok a funkcionális programozásban.

A funkcionális programozás motivációi

- Rekurzio, teljes indukció (vö. gépi kód, Fortran, Basic) – 1950-es évek
- Lineáris rekurzív adatszerkezet (lista, vö. ciklus)
- Függvények – vissza a matematikához! (vö. mellékhatás) – 1960-as évek
- Erős típusok, ellenőrzés fordításkor (vö. típusnélküli nyelvek) – 1970-es évek
- Rekurzív adattípusok (fa, vö. láncolt adatszerkezetek)
- Absztrakt adattípusok (vö. objektumok)
- Végrehajtható specifikációk (vö. tesztelés) – 1990-es évek

Mi az alapvető különbség a deklaratív és az imperatív programozás között?

- A deklaratív programozás *időtlen*, nem törődik az idővel.
- Idő → állapot → emlékezet.

A funkcionális programozás rövid története

- A függvényfogalom fejlődése – l. külön fóliákon: [fffp.pdf](#).
- Euler (1748): $\sin x$ később $\sin x$ vagy $\sin(x)$
- Alfred N. Whitehead, Bertrand Russel (1910) ... Alonzo Church: λ -*kalkulus*, λ -jelölés: $\lambda x.x + x$
- Church, 1936: λ -kalkulus (funkcionális) \equiv Turing-gép (imperatív) \longrightarrow funkcionális programozás \equiv imperatív programozás
- Church-tétel: kiszámítható függvények halmaza \equiv rekurzív függvények halmaza – ez a funkcionális programozás alapja
- 1960: ALGOL (ALGOrithmic Language) – rekurzív eljárás és függvényeljárás (!)
- 1960: LISP (LISt Processing language) – alapja a λ -kalkulus, eredeti célja: *szimbolikus differenciálás*
- 1962-től: APL, ML, HOPE, ERLANG, Miranda, SML, Haskell, gofer, clean stb.

Az ML (Meta Language) rövid története és jelene

Az ML rövid története

- ML, Edinborough 1977, tételbizonyításra (kijelentések igazolására)
- Definition of Standard ML, 1990
 - Alapnyelv (Core Language)
 - Modulnyelv (Module Language)
- Revised Definition of Standard ML, 1997
- SML Basis Library (Alapkönyvtár), 1997

SML-megvalósítások

- Moscow ML (mosml): <http://www.dina.dk/~sestoft/mosml.html>
- Standard ML of New Jersey (sml):
<http://cm.bell-labs.com/cm/cs/what/smlnj>

Információk a funkcionális programozásról

Hálózati információforrások:

Comp.Lang.ML FAQ

<http://www.cis.ohio-state.edu/hypertext/faq/usenet/meta-lang-faq/faq.html>

Andrew Cumming: A Gentle Introduction to ML

<http://www.dcs.napier.ac.uk/course-notes/sml/manual.html>

Stephen Gilmore: Programming in Standard ML '97

<http://www.dcs.ed.ac.uk/home/stg>

Robert Harper: Programming in Standard ML

<http://www.cs.cmu.edu/People/rwh/>

Fox project at CMU

<http://foxnet.cs.cmu.edu/sml.html>

5 ML-irodalom (csak angolul)

Törzsművek az előadásokhoz

Jeffrey D. Ullman: *Elements of ML Programming* (2nd Edition, ML97)

MIT Press 1997

<http://www-db.stanford.edu/ullman/emlp.html>

Lawrence C. Paulson: *ML for the Working Programmer* (2nd Edition, ML97)

Cambridge University Press 1996

<http://www.cl.cam.ac.uk/users/lcp/MLbook/>

Richard Bosworth: *A Practical Course in Functional Programming Using Standard ML*

McGraw-Hill 1995

A FÜGGVÉNY FOGALMA ÉS TULAJDONSÁGAI



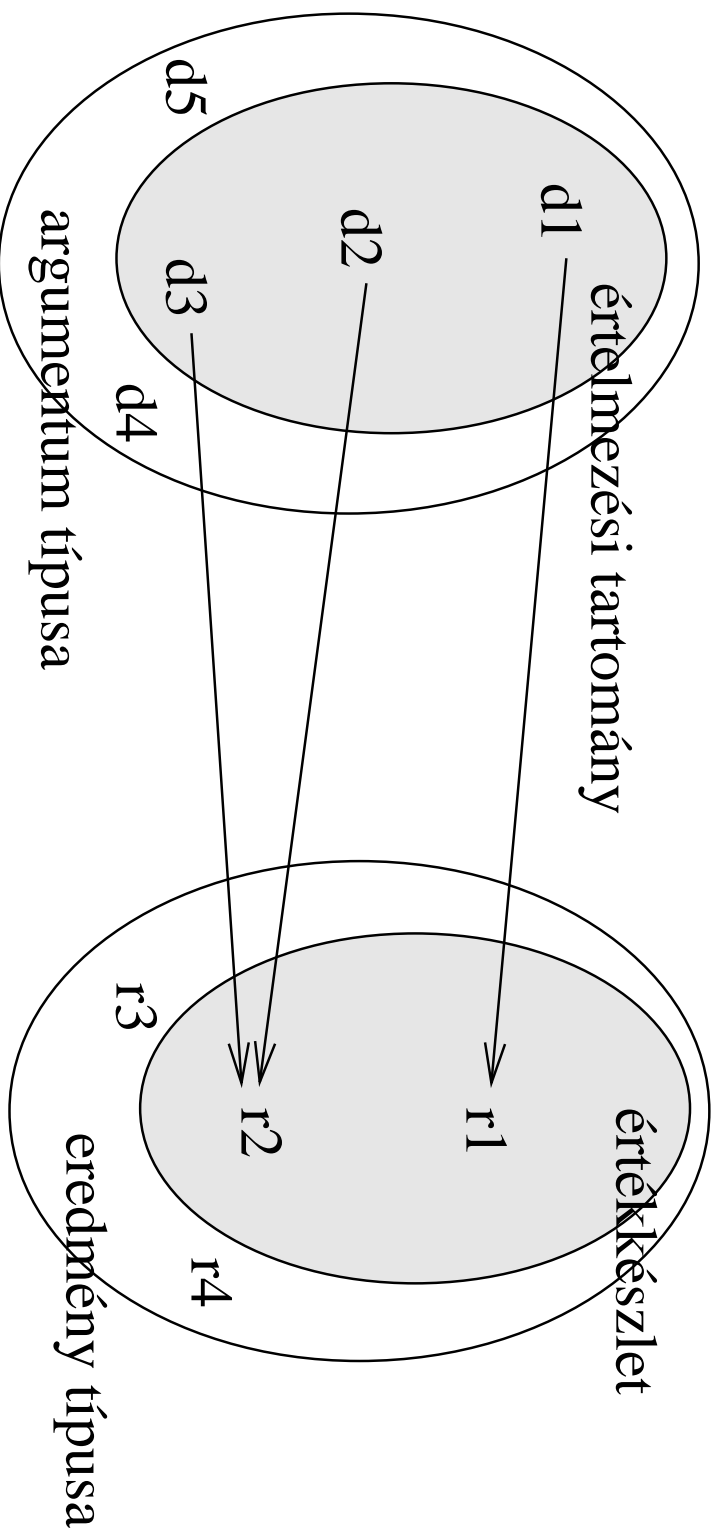
A típus és a függvény fogalma

- A típus fogalma
- A típus értékek egy halmaza (pl. egész típus = az egész számok halmaza)
- Jelölése: α, β, \dots (az ún. *típuselméletben* így használják)
- A függvény fogalma
- A függvény valamely D halmaznak valamely R halmazba való olyan *egyértelmű* leképezése, amelyet meghatároz a $(d; r)$ rendezett párok halmaza, ahol $d \in D$ és $r \in R$.
- A d a függvény argumentuma (paramétere), az r az eredménye
- A D a függvény értelmezési tartománya, az R az értékkészlete
- A típusos nyelvekben d is, r is *meghatározott* típusú
- Függvény értelmezési tartománya \subseteq argumentum típusa
- Függvény értékkészlete \subseteq eredmény típusa

A függvény mint érték

- A függvény „teljes jogú” (*first-class*) érték a funkcionális programozási nyelvekben
- A függvény típusa általában: $\alpha \rightarrow \beta$, ahol az α az argumentum, a β az eredmény típusát jelöli
- A függvény – érték: *függvényérték*
- Fontos: a függvényérték *nem* a függvény *alkalmazásának* az eredménye!
- Példák függvényértékre
 - \sin (a típusa: *valós* \rightarrow *valós*)
 - round (a típusa: *valós* \rightarrow *egész*)
 - $f \circ g$ (a típusa: $\alpha \rightarrow \beta$)
- Példa függvényalkalmazásra
 - $\text{round } 5.4 = 5$, azaz ennek a függvényalkalmazásnak egy *egész* típusú érték az eredménye

A függvény mint leképezés



Függvények tulajdonságai és osztályozása

- Parciális függvény: értelmezési tartomány \subset argumentum típusa

Figyelem: ez hibák forrása lehet!

- Teljes függvény: értelmezési tartomány = argumentum típusa
- Szürjektív függvény: értékkészlet = eredmény típusa
- Nem-szürjektív függvény: értékkészlet \subset eredmény típusa
- Injektív függvény: a leképezés kölcsönösen egyértelmű
- Az $f : \alpha \rightarrow \beta$ injektív függvény inverze: $f^{-1} : \beta \rightarrow \alpha$
- Bijektív = injektív + szürjektív, azaz f bijektív, ha f^{-1} teljes függvény

Függvények alkalmazása

- *Függvényalkalmazást* jelöl az f és e jelek egymás mellé írása („*juxtapozicionálása*”): $f\ e$ azt jelenti, hogy f -et alkalmazzuk e -re.
- Általánosabban: az $f\ e$ kifejezésben az e tetszőleges olyan kifejezés, amelynek az értéke az f értelmezési tartományába esik.
- Még általánosabban: az $f\ e$ kifejezésben az f függvényértéket eredményező tetszőleges kifejezés, e pedig tetszőleges olyan kifejezés, amelynek az értéke az f értelmezési tartományába esik.

Két- vagy többargumentumú függvények

- Függvény alkalmazása két- vagy több argumentumra
- 1. Az argumentumokat *összetett adatnak* – párnak, rekordnak, listának stb. – tekintjük, pl. $f(1, 2)$
 - az f függvény alkalmazását jelenti az $(1, 2)$ párra.
- 2. A függvényt több egymás utáni lépésben alkalmazzuk az argumentumokra, pl. $f_{12} \equiv (f_1)^2$ azt jelenti, hogy
 - az első lépésben az f függvény alkalmazzuk az 1 értékre, ami egy függvényt ad eredményül,
 - a második lépésben az első lépésben kapott függvényt alkalmazzuk a 2 értékre, így kapjuk meg az f_{12} függvényalkalmazás (vég)eredményét.
- Infix jelölés: $x \oplus y \equiv \text{az } \oplus \text{ függvény alkalmazása az } (x, y) \text{ párra mint argumentumra}$

FÜGGVÉNYEK AZ SML-BEN



Függvények alkalmazása az SML-ben

- Az SML-ben az f és az e tetszőleges *név* lehet, amelyeket megfelelően *szeparálni* kell egymástól: f e , vagy $f(e)$, vagy $(f)e$
- Szeparátor: nulla, egy vagy több *formázó* karakter (\lfloor , $\backslash t$, $\backslash n$ stb.). Nulla db formázó karakter elegendő pl. a (előtt és a) után.
- A szeparátor a legerősebb balra kötő infix operátor az SML-ben.
- Példák:
 $\text{Math.sin } 1.00$, $(\text{Math.cos})\text{Math.pi}$, $\text{round}(3.17)$, $2 + 3$, $(\text{real}) (3 + 2 * 5)$
- Függvények egy csoportosítása az SML-ben
 - Beépített függvények, pl. $+$, $*$ (infix), real , round (prefix)
 - Könyvtári függvények, pl. Math.sin , Math.cos , Math.pi
 - Felhasználó által definiálható függvények, pl. terület , $/\backslash$, head

SML-példa: Egyszeres Hamming-távolságú ciklikus kód

- A függvényt *táblázattal* adjuk meg:
- | | | |
|----|----|-------------|
| 00 | 01 | fn 00 => 01 |
| 01 | 11 | 01 => 11 |
| 11 | 10 | 11 => 10 |
| 10 | 00 | 10 => 00 |
- Változatok („klózek”): minden lehetséges esetre egy változat.
 - Az fn (olvasd: *lambda*), névtelen függvényt, *függvénykifejezést* vezet be.
 - A függvény néhány alkalmazása:
- (fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00) 10
 - (fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00) 11
 - (fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00) 111
- Mintaillesztés, egyesítés
- Érthető, de nem robusztus (vö. parciális a függvény!).

SML-példa: modulo n alapú inkrementálás

- A függvényt most *algoritmussal* adjuk meg, nem táblázattal
 - n nem lehetne változó, túl sok változatot kellene felírni stb.
- $\text{fn } i \Rightarrow (i + 1) \bmod n$
 - $\text{az } i$ ún. kötött változó, a névtelen függvény argumentuma
 - $\text{az } n$ éppen a kifejezésben szabad változó, és nincs értéke (!)
 - $\text{az } n\text{-et}$ is le kell kötni mint a függvény argumentumát
- $\text{fn } i \Rightarrow \text{fn } n \Rightarrow (i + 1) \bmod n$

- A függvény néhány alkalmazása:

- $(\text{fn } i \Rightarrow (\text{fn } n \Rightarrow (i + 1) \bmod n) \ 4) \ 1)$
- $(\text{fn } i \Rightarrow (\text{fn } n \Rightarrow (i + 1) \bmod n) \ 128) \ 111$
- $(\text{fn } i \Rightarrow (\text{fn } n \Rightarrow (i + 1) \bmod n) \ 4) \ \sim 7$
- $(\text{fn } i \Rightarrow (\text{fn } n \Rightarrow (i + 1) \bmod n) \ 128) \ 6.0 - \text{hibás!}$

Értékdeklaráció SML-ben: függvényérték deklarációja

- Név kötése függvényértékhez

- `val incMod = fn i => fn n => (i + 1) mod n`
- `val kovKod = fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00`

- Szintaktikai édesítőszerez

- `fun incMod n i = (i + 1) mod n`

- **Figyelem:** `i` és `n` sorrendje megfordult!

- ```
fun kovKod 00 = 01
 | kovKod 01 = 11
 | kovKod 11 = 10
 | kovKod 10 = 00
```

- Alkalmazásuk argumentumra

- `incMod 128 111`
- `kovKod 01`

## fejkomment

---

egyen *fejkomment* minden (függvény)érték-deklarációhoz!

- `(* incMod n i = (i+1) modulo n szerint  
PRE: n > 0, n > i >= 0  
*)  
fun incMod n i = (i + 1) mod n`
- `(* kovKod cc = a kétbites, egyszeres Hamming-távolságú, ciklikus  
kódkészlet cc-t követő eleme  
PRE: cc in {00, 01, 11, 10}  
*)  
fun kovKod 00 = 01  
| kovKod 01 = 11  
| kovKod 11 = 10  
| kovKod 10 = 00`

# TÍPUSOK ÉS ÉRTÉKEK AZ SML-BEN

---

# Típusok

- Típusok és programozási nyelvek
  - Típus nélküli nyelvek, pl. assembly, LISP, Prolog
  - Gyengén típusos nyelvek, pl. Fortran, Algol, BASIC, C, C++, Pascal
  - Erősen típusos nyelvek, pl. Ada, SML, clean
  - Erős típus: a típusok ( $\sim$  halmazok) diszjunktak (nincs közös elemük)
- Egyszerű SML-típusok
  - `int` – előjeles egész szám, a  $\mathbb{Z}$  egy részhalmaza
  - `word`, `word8` – előjel nélküli pozitív egész, az  $N_0$  egy részhalmaza
  - `real` – előjeles racionális (valós?!) szám, a  $\mathbb{Q}$  egy részhalmaza
  - `bool`, `char`, `order`, `unit`
  - `string`
- Összetett SML-típusok (példák)
  - `rekord`
  - `lista`

## Értékdeklaráció az SML-ben: **név kötése tetszőleges értékhez**

- Függvényértéket így kötöttünk tetszőleges névhez:

```
val incMod = fn i => fn n => (i + 1) mod n
```

- Tetszőleges típusú érték köthető tetszőleges névhez:

```
val harom = 2 + 1 : int
val MHz = 94.5 : real
val veege = true : bool
val kisa = #"a" : char
val palindrom = "ABBA" : string
val kisebb = LESS : order
val ezNemSemmi = () : unit
val rat = {num = 3, den = 4} : {den : int, num : int}
val blista = [2,3,4] @ [3,2] : int list
val telenek = [0w123, 0wxcd] : word list
```

true, false

LESS, EQUAL, GREATER  
Egyetlen érték a ()!

Mezőnevek ábécé sorrendben.

- Típusmegkötés:

|                                     |                        |
|-------------------------------------|------------------------|
| val id = fn (n : int) => n          | Példák: id 3;, id 4.5; |
| val telenek = [0w65, 0wx41 : word8] | Típusa: word8 list     |

# EGYSZERŰSÍTETT SML-SZINTAXIS





## SML-szintaxis: különleges állandó

- Előjeles egész állandó  
Példák:     0     ~0     4     ~04   999999   0xFFFF   ~0x1ff  
Ellenpéldák: 0.0   ~0.0   4.0   1E0   -317     0xFFFF   -0x1ff
- Valós állandó  
Példák:     0.7   ~0.7   3.32E5   3E~7   ~3E~7   3e~7   ~3e~7  
Ellenpéldák: 23     .3     4.E5     1E2.0   1E+7     1E-7
- Előjel nélküli egész állandó  
Példák:     0w0     0w4     0w999999   0wxFFFF   0wx1ff  
Ellenpéldák: 0w0.0   ~0w4   -0w4     0w1E0     0wXXXX   0wXXXX
- Füzérállandó: "-ek között álló nulla vagy több nyomtatható karakter, szóköz vagy \ jellel kezdődő *escape-szekvencia* (l. a táblázatot a következő lapon).
- Karakterállandó: # jelet közvetlenül követő, egykarakteres füzérállandó.  
Példák:     #"a"   #"\"   #"\"~Z"   #"\"255"   #"\""  
Ellenpéldák: # "a"   #c     #"""

## SML-szintaxis: escape-szekvenciák

---

### ● Escape-szekvenciák

|                      |                                                                                                                                 |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <code>\a</code>      | Csengőjel (BEL, ASCII 7).                                                                                                       |
| <code>\b</code>      | Visszalépés (BS, ASCII 8).                                                                                                      |
| <code>\t</code>      | Vízszintes tabulátor (HT, ASCII 9).                                                                                             |
| <code>\n</code>      | Újsor, soremelés (LF, ASCII 10).                                                                                                |
| <code>\v</code>      | Függőleges tabulátor (VT, ASCII 11).                                                                                            |
| <code>\f</code>      | Lapdobás (FF, ASCII 12).                                                                                                        |
| <code>\r</code>      | Köcsi-vissza (CR, ASCII 13).                                                                                                    |
| <code>\^c</code>     | Vezérlő karakter, ahol $64 \leq c \leq 95$ (@ ... _), és <code>\^c</code> ASCII-kódja 64-gyel kevesebb <i>c</i> ASCII-kódjánál. |
| <code>\ddd</code>    | A <i>ddd</i> kódú karakter ( <i>d</i> decimális számjegy).                                                                      |
| <code>\xxxx</code>   | A <i>xxxx</i> kódú karakter ( <i>x</i> hexadecimális számjegy).                                                                 |
| <code>\"</code>      | Idézőjel (").                                                                                                                   |
| <code>\\</code>      | Hátrátört-vonal (\).                                                                                                            |
| <code>\f...f\</code> | Figyelemmen kívül hagyott sorozat. <i>f...f</i> nulla vagy több formázókaraktert (szóköz, HT, LF, VT, FF, CR) jelent.           |

## SML-szintaxis: név

---

- Alfanumerikus: kis- és nagybetűk, számjegyek, perccjel (') és aláhúzás-jelek ( \_ ) olyan sorozata, amely betűvel vagy perccjellel kezdődik
- Példák: `tothGyorgy`   `Toth_3_Gyorgy`   `toth'gyorgy`
- Szimbolikus: az alábbi jelek tetszőleges, nem üres sorozata  
! % & \$ # + - / : < = > ? @ \ ~ ' ^ \_ \*  
Példák: `++` `<->` `||` `##` `|=`
- Speciális a szerepe az alábbi fenntartott jeleknek  
( ) [ ] { } , ; . . . .
- Más jelentés nem rendelhető az alábbi fenntartott nevekhez  
`abstype` and `andalso` as `case` `do` `datatype` `else` `end` `eqtype` `exception`  
`fn` `fun` `functor` `handle` `if` `in` `include` `infix` `infixr` `let` `local` `nonfix`  
`of` `open` `orelse` `raise` `rec` `sharing` `sig` `signature` `struct` `structure`  
`then` `type` `val` `where` `with` `withtype` `while` : :: :> \_ | = => -> #

## SML-szintaxis: szintaktikai kategóriák (egyszerűsítve)

- A nevek és más azonosítók *szintaktikai kategóriákba* sorolhatók
  - vid*      értéknév      value identifier      long
  - tyvar*    típusváltozó    type variable
  - tycon*    típuskonstruktor    type constructor      long
  - lab*      mezőnév      record label
  - strid*    struktúranév    structure identifier    long
  - sigid*    szignatúránév    signature identifier
  - unitid*   állománynév    unit identifier
- Az *értéknév* tetszőleges név; jelölhet állandó értéket, függvényértéket, adatkonstruktor, kivételkonstruktor. Példák: pi + sin nil true Match
- A *típusváltozó* perccel kezdődő alfanumerikus név. Példa: 'a.
- A *típuskonstruktor* tetszőleges név; jelölhet típusállandót vagy típusfüggvény-értéket. Példák: int order \$ \* -> list
- A *mezőnév* tetszőleges név vagy (nem 0-val kezdődő) pozitív egész szám.  
Példák: num    2

## SML-szintaxis: szintaktikai kategóriák (folyt.)

- Minden, az előző felsorolásban „long”-gal megjelölt  $X$  szintaktikai kategóriának van egy *long* $X$  párja. A *long* $X$  szintaktikai kategóriába tartozó nevek rövid és hosszú (ún. minősített) alakban is felírhatók. A *rövid alak* csak egy névből, a *hosszú alak* egy hosszú struktúranévből, egy pontból és egy névből áll:

|                 |     |     |  |                       |                |                      |
|-----------------|-----|-----|--|-----------------------|----------------|----------------------|
| <i>long</i> $x$ | ::= | $x$ |  | név                   |                | identifier           |
|                 |     |     |  |                       |                |                      |
|                 |     |     |  | <i>longstrid.</i> $x$ | minősített név | qualified identifier |

Példák:

- `explode`
- `Real.toString`
- `Int. +`
- `List.filter`

## SML-szintaxis: szintaktikai kategóriák (folyt.)

---

- A *strukturánév* és a *szignatúránév* a *modulnyelv* fogalmkörébe tartozó tetszőleges nevek.

Példák: Char   Int   List   TextIO

- Az *állománynév* a *modulnyelv* fogalmkörébe tartozó tetszőleges olyan név, amelyet az adott operációs rendszer is megenged; forráskódú vagy tárgykódú struktúra- vagy szignatúra-állományt azonosít.

- A *strid* struktúránév a `unitid.no` tárgykódú struktúra-állományra hivatkozik, ahol `unitid = strid`. A `unitid.sml` struktúra-állomány fordításakor már léteznie kell a `unitid.ui` tárgykódú szignatúra-állomálynak, összeszerkesztésekor pedig már léteznie kell a `unitid.no` tárgykódú struktúra-állomálynak.

- A *sigid* szignatúránév a `unitid.ui` tárgykódú szignatúra-állományra hivatkozik, ahol `unitid = sigid`. A `unitid.ui` tárgykódú szignatúra-állományt a `unitid.sig` forráskódú szignatúra-állomány lefordításával kell előállítani.

## Struktúra, szignatúra, tárgykódú és forráskódú állományok

### Példák

- Struktúra a megfelelő szignatúrával
  - `structure Rat :> Rat = struct implementáció end`
  - `signature Rat = sig specifikáció end`
- A Rat struktúrát és szignatúrát tartalmazó állományok
  - `Rat.sml`: a forráskódú struktúra-állomány (a `.sml` kiterjesztés használata ajánlott, de nem kötelező)
  - `Rat.sig`: a forráskódú szignatúra-állomány (a `.sig` kiterjesztés használata kötelező)
  - `Rat.no`: a tárgykódú struktúra-állomány (a `.no` kiterjesztés használata kötelező)
  - `Rat.ui`: a tárgykódú szignatúra-állomány (a `.ui` kiterjesztés használata kötelező)

## Függvényjel helyzete és kötése

---

- Függvényjel helyzete és kötése (általában)
  - Egy függvényjel *prefix*, *infix* vagy *postfix* helyzetű lehet.
  - Az infix helyzetű függvényjelet gyakran *operátornak* nevezik.
  - Egy (infix helyzetű) operátor lehet *asszociatív* vagy *nem-asszociatív*, köthet balra vagy jobbra. Asszociatív operátor esetén a kötési iránynak nincs jelentősége.

- Infix Prolog-operátor kötése

- $xfx = f$  mindkét oldalán  $f$  csak zárójelben ismétlődhet,
- $yfx = f$  bal oldalán  $f$  zárójelezés nélkül ismétlődhet ( $f$  „balra köt”),
- $xfy = f$  jobb oldalán  $f$  zárójelezés nélkül ismétlődhet ( $f$  „jobbra köt”).



## Függvényjel helyzete és kötése az SML-ben

---

- Kifejezések és típuskifejezések az SML-ben
  - Az SML-ben a szokásos kifejezések mellett vannak *típuskifejezések* is.
  - A függvényeket *értékekre*, a típusfüggvényeket *típusokra* alkalmazhatjuk.
- Függvényjel és típusfüggvényjel helyzete és kötése az SML-ben
  - Függvényjel: *prefix* vagy *infix*.
  - Típusfüggvényjel: *infix* vagy *postfix*.
  - Az *infix* helyzetű függvényjel és típusfüggvényjel (szokásos néven operátor, ill. típusoperátor) vagy balra, vagy jobbra köt.
- Infix helyzetben csak a két beépített típusoperátor (\* és ->) lehet.
- A \* balra, a -> jobbra köt. A \* erősebben köt, mint a ->.
- A típusoperátorok erősebben kötnek az összes többi operátornál.

## Függvényjel helyzete és kötése az SML-ben

- Tetszőleges kétargumentumú függvényjelet lehet adott preferenciájú (infix helyzetű) operátorként deklarálni az infix vagy az infixr direktívával.
- Az infix balra, az infixr jobbra kötő operátort deklarál.
- Egy minősített nevet, vagy egy olyan nevet, amelyet az op direktíva előz meg, csak *prefix* helyzetben lehet alkalmazni.
- A nonfix direktíva az (infix helyzetű) operátort tartósan prefix helyzetűvé alakítja. (Az op direktíva csak átmenetileg teszi prefix helyzetűvé.)
- A  $d$  0 és 9 közötti számjegy, az operátor precedenciája (opcionális, alapértelmezés szerinti értéke 0). Nagyobb szám erősebb kötést jelent (éppen fordítva, mint a Prologban!).

- Az  $id_i$  tetszőleges név ( $n \geq 1$ ).

|        |                     |                   |            |                    |
|--------|---------------------|-------------------|------------|--------------------|
| infix  | $\langle d \rangle$ | $id_1 \dots id_n$ | balra köt  | binds to the left  |
| infixr | $\langle d \rangle$ | $id_1 \dots id_n$ | jobbra köt | binds to the right |
| nonfix |                     | $id_1 \dots id_n$ | prefix     | prefix             |

## A beépített operátorok és precedenciájuk az SML-ben

Az alábbi táblázatban wordint, num és numtxt az alábbi típusnevek helyett állnak.

wordint = int, word, word8.                      num = int, real, word, word8.

numtxt = int, real, word, word8, char, string.

| <i>Prec.</i> | <i>Operátor</i> | <i>Típus</i>                 | <i>Eredmény</i>                     | <i>Kivétel</i> |
|--------------|-----------------|------------------------------|-------------------------------------|----------------|
| 7            | *               | num * num -> num             | szorzat                             | Overflow       |
|              | /               | real * real -> real          | hányados                            | Div, Overflow  |
|              | div, mod        | wordint * wordint -> wordint | hányados, maradék                   | Div, Overflow  |
|              | quot, rem       | int * int -> int             | hányados, maradék                   | Div, Overflow  |
| 6            | +, -            | num * num -> num             | összeg, különbség                   | Overflow       |
|              | ^               | string * string -> string    | egybeírt szöveg                     | Size           |
| 5            | ::              | 'a * 'a list -> 'a list      | elemmel bővített lista (jobbra köt) |                |
|              | @               | 'a list * 'a list -> 'a list | összefűzött lista (jobbra köt)      |                |
| 4            | =, <>           | 'a * 'a -> bool              | egyenlő, nem egyenlő                |                |
|              | <, <=           | numtxt * numtxt -> bool      | kisebb, kisebb-egyenlő              |                |
| 3            | >, >=           | numtxt * numtxt -> bool      | nagyobb, nagyobb-egyenlő            |                |
|              | :=              | 'a ref * 'a -> unit          | értékadás                           |                |
| 2            | o               | ('b -> 'c) * ('a -> 'b)      | a két függvény kompozíciója         |                |
|              |                 | -> ('a -> 'c)                |                                     |                |
| 0            | before          | 'a * 'b -> 'a                | a bal oldali argumentum             |                |

## SML-szintaxis: nemterminális szimbólumok, nyelvtani jelölések

- Minden nemterminális szimbólumot *változatok* sorozataként definiálunk, soronként egy változattal. Üres sor üres változatot jelent.
- $A <$  és  $a >$  csúcsos zárójelpárok opcionális kifejezést fognak közre.
- Bármely  $X$  nemterminális szimbólumra az alábbiak szerint definiáljuk az  $Xseq$  nemterminális szimbólumot:
 

|              |                                       |                      |
|--------------|---------------------------------------|----------------------|
| $Xseq ::= X$ | egyelemű sorozat                      | singleton sequence   |
|              | üres sorozat                          | empty sequence       |
|              | $X_1, \dots, X_n$ sorozat, $n \geq 1$ | sequence, $n \geq 1$ |
- A változatokat prioritásuk csökkenő sorrendjében soroljuk föl.
- A változatokat számozzuk, a példákban utalunk az alkalmazott változatra.
- A függvényjelek és operátorok általában balra kötnek, az eltérést jelezzük.
- Minden ismétlődő konstrukció (pl. a *klózsorozat*) a lehető legmesszebb terjeszkedik jobbra. Ezért pl. egy case-kifejezést egy másik case- vagy fn-kifejezésen, valamint egy fun-definíción belül zárójelbe kell tenni.

## SML-szintaxis: kifejezések és klózsorozatok (egyszerűsítve)

### • Kifejezés (*exp*: expression)

|                    |                                 |                   |                     |
|--------------------|---------------------------------|-------------------|---------------------|
| (1) <i>exp</i> ::= | <i>infixexp</i>                 |                   |                     |
| (2)                | <i>exp</i> : <i>ty</i>          | típusmegkötés     | type constraint     |
| (3)                | raise <i>exp</i>                | kivételjelzés     | raise exception     |
| (4)                | case <i>exp</i> of <i>match</i> | esetszétválasztás | case analysis       |
| (5)                | fn <i>match</i>                 | függvénykifejezés | function expression |

### • Példák:

```

fn (n : int) => n; vö. (2), (5)
case c of 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00; vö. (4), (19)
fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00; vö. (5), (19)
fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00
 | _ => raise Domain; vö. (3), (5), (19)

```

## SML-szintaxis: kifejezések és klózsorozatok (folyt.)

- Infix kifejezés (*infixp*: infix expression)

(6) *infixp* ::= *appexp*

(7) *infixp*<sub>1</sub> *id infixp*<sub>2</sub> | infix alkalmazás | infixed application

- Applikatív kifejezés (*appexp*: applicative expression)

(8) *appexp* ::= *atexp*

(9) *appexp atexp* | (prefix) alkalmazás | (prefixed) application

- Példák:

3 + 4;

vö. (7)

Real.toString 3.56;

vö. (9)

Int.toString(round 3.56); vö. (9), (17)

## SML-szintaxis: kifejezések és klózsorozatok (folyt.)

### ● Atomi kifejezés (*atexp*: atomic expression)

|                       |                                              |                      |                     |
|-----------------------|----------------------------------------------|----------------------|---------------------|
| (10) <i>atexp</i> ::= | <i>scon</i>                                  | különleges állandó   | special constant    |
| (11)                  | $\langle \text{op} \rangle \textit{longvid}$ | értéknév             | value identifier    |
| (12)                  | $\{ \langle \textit{exprow} \rangle \}$      | rekord               | record              |
| (13)                  | $\# \textit{lab}$                            | rekordszelektor      | record selector     |
| (14)                  | $(\textit{exp}_1, \textit{exp}_2)$           | pár                  | pair                |
| (15)                  | $()$                                         | nullas               | 0-tuple             |
| (16)                  | $[\textit{exp}_1, \dots, \textit{exp}_n]$    | lista, $n \geq 0$    | list, $n \geq 0$    |
| (17)                  | $(\textit{exp})$                             | kifejezés zárójelben | parenthesized expr. |

### ● Példák:

|                               |                      |
|-------------------------------|----------------------|
| 1.12, # "Z", 0w123            | vö. (10)             |
| Math.pi, false, Math.sin, sin | vö. (11)             |
| #den {num=1, den=2}           | vö. (12), (13), (18) |
| (2, 3.5), (), [1, 2, 3]       | vö. (14), (15), (16) |

## ML-szintaxis: kifejezések és klózsorozatok (folyt.)

- Kifejezésor (*exprow*: expression row)
- (18) *exprow* ::= *lab* = *exp* < , *exprow*>
- Klózsorozat (*match*)
- (19) *match* ::= *mrule* < | *match*>
- Klóz (*mrule*: match rule)
- (20) *mrule* ::= *pat* => *exp*
- Példák:

$$\begin{array}{l} \text{num}=1, \text{ den}=2 \\ 00 \Rightarrow 01 \mid 01 \Rightarrow 11 \mid 11 \Rightarrow 10 \mid 10 \Rightarrow 00 \text{ v\"o. (18)} \\ \text{v\"o. (19), (20)} \end{array}$$



# SML-szintaxis: deklarációk és kötések

## Deklaráció (*dec*: declaration)

|      |                                                                                     |                                        |                                |
|------|-------------------------------------------------------------------------------------|----------------------------------------|--------------------------------|
| (20) | <i>dec</i> ::= <i>val tyvarseq valbind</i>                                          | értékkdeklaráció                       | value declaration              |
| (21) | <i>fun tyvarseq fvalbind</i>                                                        | függvénydeklaráció                     | function declaration           |
| (22) | <i>type typpbind</i>                                                                | típusdeklaráció                        | type declaration               |
| (23) |                                                                                     | üres deklaráció                        | empty declaration              |
| (24) | <i>dec</i> <sub>1</sub> <; > <i>dec</i> <sub>2</sub>                                | deklaráció-sorozat                     | sequential declaration         |
| (25) | <i>infix</i> < <i>d</i> > <i>id</i> <sub>1</sub> ... <i>id</i> <sub><i>n</i></sub>  | <i>infix</i> -direktíva, <i>n</i> ≥ 1  | <i>infix</i> (left) directive  |
| (26) | <i>infixr</i> < <i>d</i> > <i>id</i> <sub>1</sub> ... <i>id</i> <sub><i>n</i></sub> | <i>infixr</i> -direktíva, <i>n</i> ≥ 1 | <i>infix</i> (right) directive |
| (27) | <i>nonfix id</i> <sub>1</sub> ... <i>id</i> <sub><i>n</i></sub>                     | <i>nonfix</i> -direktíva, <i>n</i> ≥ 1 | <i>nonfix</i> directive        |

## Példák:

```

val xy = "XY"; fun ++ x y = x ^ y vö. (20), (21), (24)
type Rat = {num : int, den : int} vö. (22)
infixr 4 ++; fun x ++ y = x ^ y vö. (21), (26)

```

## SML-szintaxis: deklarációk és kötések (folyt.)

- Értékkötés (*valbind*: value binding)

(28)  $valbind ::= pat = exp$  értékkötés value binding

(29)  $rec\ valbind$  rekurzív kötés recursive binding

- Függvényérték-kötés (*fvalbind*: function value binding)

(30)  $fvalbind ::=$  
 $\begin{array}{l} \langle op \rangle\ var\ atpat_{11} \dots atpat_{1n} \langle :ty \rangle = exp_1 \\ \mid \langle op \rangle\ var\ atpat_{21} \dots atpat_{2n} \langle :ty \rangle = exp_2 \\ \mid \dots \\ \mid \langle op \rangle\ var\ atpat_{m1} \dots atpat_{mn} \langle :ty \rangle = exp_m \\ \langle and\ fvalbind \rangle \end{array}$ 
  $m, n \geq 1$

*Megjegyzés:* Ha *var* infix, akkor egy *fvalbind* definícióban vagy infix helyzetben kell használni, vagy elé kell írni az *op* direktívát; azaz a definícióban a bal oldalon (*atpat var atpat*') vagy *op var (atpat, atpat)* írható. A zárójelek elhagyhatók, ha *atpat*' után közvetlenül *:ty* vagy = áll.

- Példák:

```
val even = fn 0 => true | x => not(odd(x-1))
and odd = fn 0 => false | y => not(even(y-1)); vö. (28)
fun (f o g) x = g(f x); vö. (30)
```

## SML-szintaxis: típuskifejezések

### • Típus (*ty*: type)

|      |               |                                                  |                         |                          |
|------|---------------|--------------------------------------------------|-------------------------|--------------------------|
| (31) | <i>ty</i> ::= | <i>tyvar</i>                                     | típusváltozó            | type variable            |
| (32) |               | <i>tycon</i>                                     | típuskonstruktor        | type constructor         |
| (33) |               | { < <i>tyrow</i> > }                             | rekordtípus-kifejezés   | record type expression   |
| (34) |               | <i>ty</i> <sub>1</sub> * <i>ty</i> <sub>2</sub>  | pár-típus               | pair type                |
| (35) |               | <i>ty</i> <sub>1</sub> -> <i>ty</i> <sub>2</sub> | függvénytípus-kifejezés | function type expression |
| (36) |               | ( <i>ty</i> )                                    | típus zárójelben        | parenthesized type       |

### • Típuskifejezés-sor (*tyrow*: type-expression row)

(37) *tyrow* ::= *lab* : *ty* < , *tyrow* >

### • Példák:

|                                                   |                |
|---------------------------------------------------|----------------|
| 'a, 'c, 'gamma                                    | vö. (31)       |
| int, real, word, word8, char, bool, string, order | vö. (32)       |
| int * int -> int, unit -> unit                    | vö. (34), (35) |
| ('a -> 'b) -> ('a list -> 'b list)                | vö. (35), (36) |
| {num : int, den : int}, num : int, den : int      | vö. (33), (37) |

## SML-szintaxis: minták

### ● Atomi minta (*atpat*: atomic pattern)

|      |                  |                                                                   |                     |                       |
|------|------------------|-------------------------------------------------------------------|---------------------|-----------------------|
| (38) | <i>atpat</i> ::= | <i>_</i>                                                          | mindenesjel         | wildcard              |
| (39) |                  | <i>scon</i>                                                       | különleges állandó  | special constant      |
| (40) |                  | <i>&lt;op&gt; longvid</i>                                         | értéknév            | value identifier      |
| (41) |                  | { <i>&lt;patrow&gt;</i> }                                         | rekord              | record                |
| (42) |                  | ( <i>pat</i> <sub>1</sub> * <i>pat</i> <sub>2</sub> )             | pár                 | pair                  |
| (43) |                  | ( ), { }                                                          | nullas              | 0-tuple               |
| (44) |                  | [ <i>pat</i> <sub>1</sub> , ..., <i>pat</i> <sub><i>n</i></sub> ] | lista, <i>n</i> ≥ 0 | list, <i>n</i> ≥ 0    |
| (45) |                  | ( <i>pat</i> )                                                    | minta zárójelben    | parenthesized pattern |

### ● Példák:

fun le GREATER = false | le EQUAL = true | le LESS = true;  
fun le GREATER = false | le \_ = true;  
fun neg Bool.false = true | neg (true) = Bool.false;  
fun prod [a, b] = a\*b | prod [a, b, c] = a\*b\*c  
| prod [a] = a | prod () = 1;

vö. (40)  
vö. (38), (40)  
vö. (40), (45)  
vö. (43), (44)

O deklaratív programozás, BME, 2001 tavaszi félév

10. előadás (funkcionális programozás)

## SML-szintaxis: minták (folyt.)

- Mintasor (*patrow*: pattern row)

|      |                   |                                              |              |             |
|------|-------------------|----------------------------------------------|--------------|-------------|
| (46) | <i>patrow</i> ::= | ...                                          | mindenesjel  | wildcard    |
| (47) |                   | <i>lab</i> = <i>pat</i> <, <i>patrow</i> >   | mintasor     | pattern row |
| (48) |                   | <i>lab</i> <: <i>ty</i> > <, <i>patrow</i> > | mezőnév mint | label as    |
|      |                   |                                              | változó      | variable    |

- Példák:

```
fun // {den = 0, ...} = raise Domain
 | // {num = n, den = d} = (real n) / (real d); vö. (46), (47)
fun // {den = 0, ...} = raise Domain
 | // {num, den} = (real num) / (real den); vö. (46), (48)
```

## SML-szintaxis: minták (folyt.)

### ● Minta (*pat*: pattern)

|      |                                                      |                        |                     |
|------|------------------------------------------------------|------------------------|---------------------|
| (49) | $pat ::= atpat$                                      | atomi minta            | atomic pattern      |
| (50) | $\langle op \rangle longvid$<br>$atpat$              | értékkonstrukció       | value construction  |
| (51) | $pat_1 vid pat_2$                                    | infix értékkonstrukció | infix value constr. |
| (52) | $pat : ty$                                           | minta típusmegkötéssel | typed pattern       |
| (53) | $\langle op \rangle var \langle : ty \rangle as pat$ | réteges minta          | layered pattern     |

### ● Példa:

|                                                            |                |
|------------------------------------------------------------|----------------|
| $fun\ sum\ [] = 0$                                         | vö. (50)       |
| $  sum\ [a : real] = a$                                    | vö. (52)       |
| $  sum\ (x :: z :: (yxs\ as\ y :: xs)) = x + z + sum\ yxs$ | vö. (51), (53) |
| $  sum\ (x :: y :: xs) = x + y + sum\ xs$                  | vö. (51)       |
| $  sum\ (op :: (x, xs)) = x + sum\ xs$                     | vö. (50)       |

## SML-szintaxis: szintaktikai korlátozások

---

- Nem illeszthető minta kétszer ugyanarra a névre (*vid*). Nem illeszthető kifejezésor, mintasor vagy típuskifejezés-sor kétszer ugyanarra a mezőnévre (*lab*).
- Ugyanaz a név nem köthető le kétféleképpen egy *valbind*, *typbind*, *datbind* vagy *exbind* deklarációban. A *datbind* deklarációban ugyanez érvényes az adatkonstruktorokra is.
- Ugyanaz a típusváltozó (*tyvar*) nem szerepelhet kétszer egy *tyvarseq* sorozatban valamely *typbind* vagy *datbind* deklaráció bal oldali *tyvarseq* *tycon* részében. Minden olyan típusváltozónak (*tyvar*), amelyik előfordul a jobb oldalon, szerepelnie kell *tyvarseq*-ben.
- A *rec-et* követő minden *pat* = *exp* értékkötésben az *exp*-nek, szükség esetén zárójelben, *fn match* alakúnak kell lennie, ahol egy vagy több névhez típusmegkötés is társítható.
- *true*, *false*, *nil*, *::* és *ref* nem kaphat értéket *valbind*, *datbind* vagy *exbind*, it pedig *datbind* vagy *exbind* deklarációban.

# RACIONÁLIS SZÁMOK





## Példa: racionális számok

- A racionális számokat *rekordként* ábrázoljuk; az új (gyenge) típus neve `rat`.  

```
type rat = {num : int, den : int};
```
- Nevet adunk néhány állandónak.  

```
val ratZero = {num = 0, den = 1}; val ratOne = {num = 1, den = 1};
val ratHalf = {num = 1, den = 2}; val ratThird = {num = 1, den = 3};
```
- A `rat` típusú számokat *normalizált* alakban tároljuk, különben pl.  $\frac{1}{2}$  és  $\frac{2}{4}$  nem lenne egyenlő. A normalizáláshoz szükségünk van a számláló és a nevező legnagyobb közös osztójára (`gcd`). A közös osztó egyik fontos tulajdonsága, hogy  $d|n$  és  $d|m \Rightarrow d|n \bmod m$ .  

```
* gcd : int -> int -> int
gcd n m = n és m legnagyobb közös osztója
)
in gcd n 0 = abs n
| gcd n m = gcd (abs m) (abs(n mod m));
```

## Példa: racionális számok (folyt.)

- *gcd ún. részlegesen alkalmazható függvény.* Ha összes argumentumánál kevesebbre alkalmazzuk, függvényértéket ad eredményül.
- Sajnos, a `normalize` függvényben  $n$  és  $m$  legnagyobb közös osztóját kétszer is kiszámoljuk: később látni fogjuk, hogyan javíthatunk a hatékonyságán.
- \* `normalize : rat -> rat`  
`normalize r = r normalizált alakban`  
`)`
- `un normalize {num = n, den = 0} = raise Domain`  
`| normalize {num = n, den = d} = {num = n div (gcd n d), den = d div (gcd n d)}`
- Két egészről *konstruktorfüggvény*nel (`toRat`) érdemes létrehozni a racionális számot, különben a normalizált tárolás követelménye sérülhet.
- \* `toRat : int -> int -> rat`  
`toRat n d = n nevezőjű és d számlálójú racionális szám, normalizált alakban`  
`)`
- `un toRat n d = normalize {num = n, den = d};`

# PÁR ÉS TÍPUSA



## Kitérő: pár és típusa

---

Mi a +-szal jelölt összeadás-művelet típusa SML-ben?

- A + kétoperandusú művelet, argumentuma egy *pár*, pl.  $3 + 4$ .
- + :  $\text{int} * \text{int} \rightarrow \text{int}$  vagy + :  $\text{real} * \text{real} \rightarrow \text{real}$ , ahol \* egy újabb típusművelet, a *keresztsszorzat* (*Descartes-szorzat*) jele.
- A + műveleti jel (függvényjel) *többszörös terhelésű*.
- + prefix helyzetben is használható, ha elírjuk az op kulcsszót, pl.  $\text{op}+(3, 4)$ .  
Ilyenkor az operandusait *párként, zárójelbe zárva* kell megadni.

A beépített infix típusoperátorok precedenciája és kötése

- Két beépített infix típusoperátor van az SML-ben:  $\rightarrow$  (leképzés) és \* (keresztsszorzat). A \* precedenciája a nagyobb. A  $\rightarrow$  jobbra köt, a \* nem köt sem balra, sem jobbra.

- Példák:  $'a * 'b * 'c = ('a * 'b) * 'c$   
 $'a \rightarrow 'b \rightarrow 'c = 'a \rightarrow ('b - 'c)$   
 $'a * 'b \rightarrow 'c = ('a * 'b) \rightarrow 'c$

# RACIONÁLIS SZÁMOK



## Példa: racionális számok – a négy alapművelet

```

** *, //, ++, -- : rat * rat -> rat
r1 ** r2 = az r1 és r2 racionális számok szorzata
r1 // r2 = az r1 és r2 racionális számok hányadosa
r1 ++ r2 = az r1 és r2 racionális számok összege
r1 -- r2 = az r1 és r2 racionális számok különbsége
)
infix 7 ** //; infix 6 ++ --;

fun (r1 : rat) ** (r2 : rat) = toRat (#num r1 * #num r2) (#den r1 * #den r2);
fun (r1 : rat) // (r2 : rat) = toRat (#num r1 * #den r2) (#num r2 * #den r1);
fun {num= n1, den= d1} ++ {num= n2, den= d2} = toRat (n1*d2 + n2*d1) (d1*d2);
fun {num= n1, den= d1} -- {num= n2, den= d2} = toRat (n1*d2 - n2*d1) (d1*d2);

```

## Példa: racionális számok – relációs műveletek

- Az = és a <> relációt *készen kapjuk*: két összetett érték strukturálisan összehasonlítható, ha az elemeiken az egyenlőségvizsgálat elvégezhető.

```
* <<, >>, <=>, >>= : rat * rat -> bool
r1 << r2 = igaz, ha r1 kisebb r2-nél
r1 >> r2 = igaz, ha r1 nagyobb r2-nél
r1 <=> r2 = igaz, ha r1 nem nagyobb r2-nél
r1 >>= r2 = igaz, ha r2 nem nagyobb r1-nél
)
```

```
 infix 4 << >> <=> >>=;
```

```
fun (r1 : rat) << (r2 : rat) = #num r1 * #den r2 < #num r2 * #den r1;
fun (r1 : rat) >> (r2 : rat) = #num r1 * #den r2 > #num r2 * #den r1;
fun r1 <=> r2 = not(r1 >> r2); fun r1 >>= r2 = not(r1 << r2);
```

# POLIMORFIZMUS





## Polimorfizmus

- Nézzük az identitásfüggvényt:  $\text{fun id } x = x$ .
- Mi az  $x$  típusa? Bármilyen típusú lehet: típusát *típusváltó* jelöli.  
 $> \text{val 'a id} = \text{fn : 'a} \rightarrow \text{'a}$
- $\text{id}$  *polimorf* függvényt jelöl,  $x$  és  $\text{id}$  *polítípusú* nevek.
- A *percjellel* kezdődő típusnév (pl.  $\text{'a}$ , olvasd *alfa*): *típusváltó*.  
Polimorfizmus többféle változatban fordul elő a programozásban.
- Egy *polimorf* *név* egyetlen olyan algoritmust azonosít, amely tetszőleges típusú argumentumra alkalmazható; ez a *paraméteres polimorfizmus*.
- Egy *többszörösen terhelt* *név* több különböző algoritmust azonosít: ahány típusú argumentumra alkalmazható, annyiféle; ez az *ad-hoc* vagy *többszörös terheléses* polimorfizmus.
- A polimorfizmus harmadik változata az *öröklődéses polimorfizmus* (vö. objektum-orientált programozás).

# KÉT FÜGGVÉNY KOMPOZÍCIÓJA



## Kitérő: két függvény kompozíciója

- Az  $f \circ g$  függvénykompozíció az SML-ben

$( * \ f \ o \ g = \text{az } f \text{ és } g \text{ függvények kompozíciója} )$

`infix 2 o; fun (f o g) = fn x => f(g x); vagy fun (f o g) x = f(g x);`

- Az `o` típusa `? * ? -> ?` szerzetű. Mit írjunk a `?-ek` helyébe? Vezessük le!

- A függvénydefiníció jobb oldalán álló kifejezés elemzésével kezdjük.

`x : 'a          g : 'a -> 'b          f : 'b -> 'c`

- A függvénydefinícióban az egyenlőségjel (=) bal és jobb oldalán álló kifejezéseknek azonos értéket kell eredményül adniuk, ezért `f o g` és `f` eredményének azonos a típusa (azaz `'c`).

`(f o g) : 'a -> 'c          o : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)`

- Példa: `round : real -> int,    chr : int -> char`  
`chr o round : real -> char`

# RACIONÁLIS SZÁMOK



## Példa: racionális számok (folyt.)

- A racionális számokon értelmezett  $<=>$  és  $>=>$  másképpen:

$\text{al op} <=> = \text{not } 0 \text{ op} >>; \quad \text{val op} >=> = \text{not } 0 \text{ op} <<;$

- Egy racionális számot függérré alakítás után írunk ki a képernyőre.

```
* toString : rat -> string
toString r = az r racionális szám függérré (számláló/nevező alakban,
 ha a nevező = 1, egyébként egészként)
)
```

```
fun toString {num, den = 1} = Int.toString num
| toString {num, den} = Int.toString num ~ "/" ~ Int.toString den
```

- Példák rat típusú értékek használatára

```
normalize (toRat 15 3); toString(toRat 2 3 ** toRat 5 4);
normalize (toRat 15 ~3); toString(toRat 2 3 // toRat 5 3);
normalize (toRat ~15 3); toString(toRat 1 4 ++ toRat 3 10);
normalize (toRat ~15 ~3); toString(toRat 3 10 -- toRat 1 4);
```

## Példa: racionális számok (folyt.)

### • Példák rat típusú értékek használatára (folyt.)

```
oRat 2 3 << toRat 5 4; toRat 2 3 >> toRat 5 3;
oRat 1 4 << toRat 3 10; toRat 3 10 >> toRat 1 4;
nfix 8 /- /;
un n /- / d = toRat n d;
```

```
oString(2/-/3 ** 5/-/4); 2/-/3 << 5/-/4; 1/-/4 << 3/-/10;
oString(2/-/3 // 5/-/3); 2/-/3 << 2/-/3; 3/-/10 >> 1/-/4;
oString(1/-/4 ++ 3/-/10); 2/-/3 <<= 2/-/3;
oString(3/-/10 -- 1/-/4); 2/-/3 >> 5/-/3; 3/-/10 >>= 3/-/10;
```

### • Példák gcd részleges alkalmazására

```
* gcd120 : int -> int gcd120 45;
gcd m = m legnagyobb közös osztója 120-szal gcd120 48;
) gcd120 ~96;
al gcd120 = gcd 120; gcd120 630;
```

# POLIMORFIZMUS



## Paraméteres polimorfizmus

- Az identitásfüggvény és típusa: `fun id x = x, id : 'a -> 'a`.  
Az `mosml` válasza: `val 'a id = fn : 'a -> 'a. Az id politípusú` név.
- Az = és `a <>` műveletet *készen kapjuk* a legtöbb típusra (vö. `rat`).  
A típusuk: `=, <> : ''a * ''a -> bool`. A `''` *egyenlőségi típust* jelöl, az ilyen típusú értékeken az egyenlőségvizsgálat elvégezhető.
- Az egyenlőségvizsgálat *korlátozottan* polimorf: nem minden értékre végezhető el. Pl. egy `f` és egy `g` függvény akkor és csak akkor egyenlő, ha  $\forall x. fx = gx$ . Ezt *általánosságban* lehetetlen eldönteni.
- Mi `a <, >, <=, >=` típusa?  
Pl. az `op<=-re` az `mosml` válasza: `val it = fn : int * int -> bool`.  
E négy művelet *ad-hoc* módon polimorf, a nevek *többszörösen terhelhetők*, alapértelmezés szerint `int` típusú értékekre alkalmazhatók.
- Az = részlegesen alkalmazható változata legyen: `fun eq x y = x = y`.  
Típusa: `eq : ''a -> ''a -> bool`.



Példák `eq` használatára (`''a eq : ''a -> ''a -> bool`)

---

A kifejezés

Az mosml válasza

```
eq 3 3;
eq "id" "idn";
eq id id;
```

```
> val it = true : bool
> val it = false : bool
! Toplevel input:
! eq id id;
!
! Type clash: expression of type
! 'e -> 'e
! cannot have equality type ''f
> val it = fn : int -> bool
> val it = fn : string -> bool
> val eqStr_id = fn : string -> bool
```

`eq 3;`

`eq "id";`

`val eqStr_id = eq "id";`

- Az `id` függvény, típusa (`'e -> 'e`) nem egyenlőségi típus!

- Az `eq "id"` függvényértéket ad eredményül, ezért az `eqStr_id` függvényt jelöl. Olyan függvényt, amely az `"id"` füzérre alkalmazva `true`, minden más esetben `false` értéket ad eredményül.

## Példák id használatára ('a id : 'a -> 'a)

### A kifejezés Az mosml válasza

```
id 3; > val it = 3 : int
id "id"; > val it = "id" : string
id round; > val it = fn : real -> int
id id; ! Warning: Value polymorphism:
 ! Free type variable(s) at top level in value identifier it
id id 6.9; > val it = fn : 'b -> 'b
fn x => id id x; > val it = 6.9 : real
 > val 'b it = fn : 'b -> 'b
```

### Az SML ún. *érték-polimorfizmust* használ.

- Az SML a típusváltozókat, ahol csak tudja, általánosítja (pl. `fn x => id id x`).

- Az `mosml` a nem általánosítható típusváltozókat meghagyja *szabad típusváltozónak* (pl. `id id`).

## Érték-polimorfizmus

---

- Tekintsük a val  $x = e$  deklarációt.
- Az SML az  $x$  típusában előforduló szabad típusváltozókat akkor általánosítja, ha  $e$  ún. *nem-expanzív* kifejezés.
- Ez csupán *szintaktikai* követelmény: egy kifejezés *nem-expanzív*, ha megfelel a *nexp* szintaktikai kategóriát leíró nyelvtani szabályoknak.

# Nem-expanzív kifejezés (egyszerűsítve)

- Nem-expanzív kifejezés ( *nexp*: non-expansive expression)

|                             |                                    |                                        |
|-----------------------------|------------------------------------|----------------------------------------|
| <i>nexp</i> ::= <i>scon</i> | különleges állandó                 | special constant                       |
| <i>longvid</i>              | (esetleg minősített)               | (possibly qualified) value             |
|                             | értéknév                           | identifier                             |
| { < <i>nexprow</i> > }      | nem-expanzív                       | record of non-expansive expressions    |
| ( <i>nexp</i> )             | nem-expanzív kifejezés             | parenthesized non-expansive expression |
| <i>nexp</i> : <i>ty</i>     | nem-expanzív kifejezés             | typed non-expansive expression         |
| fn <i>match</i>             | típusmegkötéssel függvénykifejezés | function expression                    |

- Nem-expanzív kifejezéssor ( *nexprow*: non-expansive expression row)

*nexprow* ::= *lab* = *nexp* < , *nexprow* >

## Példák nem-expandív és expandív kifejezésekre

- Egy nem-expandív kifejezés egyszerűen: érték (azaz tovább nem egyszerűsíthető, ún. *kanonikus* kifejezés).

```
val x = length;
> val 'a x = fn : 'a list -> int
```

length egy név, ezért nem-expandív. Az x típusát leíró 'a list -> int típuskifejezésben az 'a szabad típusváltozó általánosítható, ezt tükrözi a definíció bal oldalán az 'a x.

- Az (fn f => f) length kifejezés értéke is length, de expandív, mert nem vezethető le a fenti nyelvtani szabályok alapján.

```
val x = (fn f => f) length;
! Warning: Value polymorphism:
! Free type variable(s) at top level in value identifier x
> val x = fn : 'a list -> int
```

## Példák nem-expanzív és expanzív kifejezésekre (folyt.)

---

Az 'a típusváltozót az SML nem általánosítja. Az `mosml` meghagyja szabad típusváltozónak, és majd csak az *x első alkalmazásakor* köti le.

```
x ["abc", "def"];
! Warning: the free type variable 'a has been instantiated to string
> val it = 2 : int
x;
> val it = fn : string list -> int
```

● Ha már az 'a-t lekötöttük, más típushoz nem köthető; x nem politípusú név.

```
x [123, 456, 789];
! Toplevel input:
! x [123, 456, 789];
! ~~~
! Type clash: expression of type
! int
! cannot have type
! string
```

## $\eta$ -expanzió

- A típusváltó átalánosítása mindig kikényszeríthető a deklaráció jobb oldalának  $\eta$ -*expanziójával*.

Az  $\eta$ -expanzió az  $e$  kifejezést a nem-expanzív  $\text{fn } y \Rightarrow e$   $y$  kifejezéssel helyettesíti.

```
val x1 = fn y => ((fn f => f) length) y;
> val 'b x1 = fn : 'b list -> int
```

A fenti deklarációban a külső zárójelpár el is hagyható:

```
val x1 = fn y => (fn f => f) length y;
```

- Az  $x1$  politípusú név.

```
x1 ["abc", "def"];
> val it = 2 : int
x1 [123, 456, 789];
> val it = 3 : int
```

# LISTÁK





## lista: definíciók, konstruktorok

### Definíciók

1. A *lista* azonos típusú elemek véges (de nem korlátos!) sorozata.
2. A lista olyan *rekurzív* lineáris adatszerkezet, amely azonos típusú elemekből áll, és
  - vagy üres,
  - vagy egy elemből és az elemet követő listából áll.

### Konstruktorok

- Az üres lista jele a `nil` *konstruktorállandó*. `nil` típusa `'a list`.
- A `::` *konstruktoroperátor* új listát hoz létre egy elemből és egy (esetleg üres) listából (infix, 5-ös precedenciájú, jobbra köt, típusa `'a * 'a list -> 'a list`).
- A `nil` helyett általában a `[]` jelet használjuk (szintaktikai édesítőszert).
- A `::`-ot négyespontnak vagy *cons*-nak olvassuk (vö. *constructor*, ami a függvény hagyományos neve a  $\lambda$ -kalkulusban és egyes funkcionális nyelvekben).

## lista: jelölések, minták

### Példák

#### Lista létrehozása konstruktorokkal

```
[] nil #"" :: nil
3 :: 5 :: 9 :: nil = 3 :: (5 :: (9 :: nil))
```

#### Szintaktikus édesítőszert lista jelölésére

```
[3, 5, 9] = 3 :: 5 :: 9 :: nil
```

#### Vigyázat! A Prolog listajelölése hasonló, de vannak lényeges különbségek:

| SML       | Prolog    | SML    | Prolog        |
|-----------|-----------|--------|---------------|
| []        | []        | azonos | (x::xs)       |
| [1, 2, 3] | [1, 2, 3] | azonos | (x::y::z::zs) |
|           |           |        | [X, Y, Z Zs]  |
|           |           |        | különböző     |

### Minták

A [] és a nil állandók, a :: operátor, valamint a [x1, x2, ..., xn] listajelölés mintában is alkalmazhatók.

lista: fej (hd), fark (tl)

---

- A nem-üres lista első eleme a lista *feje*.

```
(* hd : 'a list -> 'a *)
fun hd (x :: _) = x;
```

- A nem-üres lista első utáni elemeiből áll a lista *farka*.

```
(* tl : 'a list -> 'a list *)
fun tl (_ :: xs) = xs;
```

- hd és tl *parciális* függvények. Ha könyvtárbeli megfelelőiket (List.hd, List.tl) üres listára alkalmazzuk, Empty néven *kiértelt* jeleznek.

Fontos: a parciális függvények nem tévesztendőők össze a parciálisan (azaz részlegesen) alkalmazható függvényekkel!

**lista: hossz (length), elemek összege (isum), szorzata (rprod)**

---

- Egy lista hosszát adja eredményül a már látott length függvény (l.  
List.length).

```
(* length : 'a list -> int *)
fun length (_ :: xs) = 1 + length xs
 | length [] = 0;
```

- Egy egész számokból álló lista elemeinek összegét adja eredményül isum.

```
(* isum : int list -> int *)
fun isum (x :: xs) = x + isum xs
 | isum [] = 0;
```

- Egy valós számokból álló lista elemeinek szorzatát adja eredményül rprod.

```
(* rprod : real list -> real *)
fun rprod (x :: xs) = x * rprod xs
 | rprod [] = 1.0;
```

**Példák: `hd`, `tl`, `length`, `isum`, `rprod`**

● `hd`, `tl`

### A kifejezés Az mosml válasza

|                                                                          |                                                                                                                              |
|--------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| <pre>List.hd [1, 2, 3]; List.hd []; List.tl [1, 2, 3]; List.tl [];</pre> | <pre>&gt; val it = 1 : int ! Uncaught exception: ! Empty &gt; val it = [2, 3] : int list ! Uncaught exception: ! Empty</pre> |
|--------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|

● `length`, `isum`, `rprod`

### A kifejezés Az mosml válasza

|                                                                                                              |                                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>length [1, 2, 3, 4]; length []; isum [1, 2, 3, 4]; isum []; rprod [1.0, 2.0, 3.0, 4.0]; rprod [];</pre> | <pre>&gt; val it = 4 : int &gt; val it = 0 : int &gt; val it = 10 : int &gt; val it = 0 : int &gt; val it = 24.0 : real &gt; val it = 1.0 : real</pre> |
|--------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|

## Listák: adott transzformáció alkalmazása minden elemre (map)

- Példa: vonjunk négyzetgyököket egy valós számokból álló lista minden eleméből!  
`map Math.sqrt [1.0, 4.0, 9.0, 16.0] = [1.0, 2.0, 3.0, 4.0]`
  - Általában: `map f [x1, x2, ..., xn] = [f x1, f x2, ..., f xn]`
  - A függvény típusa: `map : ('a -> 'b) -> 'a list -> 'b list`
  - Egy-egy klózt írunk a triviális és a nem-triviális eset lefedésére
    - `map f [] = []`
    - `map f (x :: xs) = f x :: map f xs`
  - `fun map f (x :: xs) = f x :: map f xs | map f [] = [] ;`
  - map típusa, ha egyargumentumú függvénynek tekintjük (ui. -> jobbra köt):  
`map : ('a -> 'b) -> ('a list -> 'b list).`
- Azaz ha `map`-et egy `'a -> 'b` típusú függvényre alkalmazzuk, akkor olyan függvényt ad eredményül, amelyet egy `'a list` típusú listára alkalmazva egy `'b list` típusú listát kapunk.

# PROGRAMHELYESSÉG



## A program helyességének igazolása a map példáján

---

- A rekurzív programról be kell látnunk, hogy
  - funkcionálisan helyes (azt kapjuk eredményül, amit várunk),
  - a kiértékelése biztosan befejeződik (nem esik „végtelen ciklusba”).
- Bizonyítása hossz szerinti *strukturális indukcióval* (amely visszavezethető a teljes indukcióra) lehetséges.

```
fun map f (x :: xs) = f x :: map f xs | map f [] = [];
```

- Feltesszük, hogy a map jó eredményt ad az eggyel rövidebb listára (azaz a lista farkára). Alkalmazzuk az f-et a lista első elemére (a fejére). A fej transzformálásával kapott eredményt a fark transzformálásával kapott lista elé fűzve valóban a várt eredményt kapjuk.
- A kiértékelés véges számú lépésben befejeződik, mert a lista véges, a map függvényt a *rekurzív ágban* minden lépésben egyre rövidülő listára alkalmazzuk, és gondoskodtunk a rekurzio leállításáról (a *triviális eset* kezeléséről, ui. van nem rekurzív ág).



# LISTÁK



## lista: adott predikátumot kielégítő elemek kiválogatása (`filter`)

- Kitérő: `explode`, `implode`

- `explode` : `string -> char list`, `pl. explode "abc" = ["a", "b", "c"]`
- `implode` : `char list -> string`, `pl. implode ["a", "b", "c"] = "abc"`

- Példa: gyűjtsük ki a kisbetűket egy karakterlistából!

```
List.filter Char.isLower (explode "Valt0gAtVa") =
 ["a", "t", "g", "t", "a"]
```

- Általában: `ha p x1 = true, p x2 = false, p x3 = true, ..., p xn = true, akkor filter p [x1, x2, x3, ..., xn] = [x1, x3, ..., xn]`.

- A függvény típusa: `filter : ('a -> bool) -> 'a list -> 'a list`

- Egy-egy klózt írunk a triviális és a nem-triviális eset lefedésére

- `filter p [] = []`
- `filter p (x :: xs) = if p x then x :: filter p xs else filter p xs`

## Listák: `filter` (folyt.)

---

- Fzzel `filter` definíciója

```
fun filter p (x :: xs) =
 if p x then x :: filter p xs else filter p xs
 | filter _ [] = [];
```

- `filter` típusa, ha egyargumentumú függvénynek tekintjük ( $\rightarrow$  jobbra köt!):  
`filter : ('a  $\rightarrow$  bool)  $\rightarrow$  ('a list  $\rightarrow$  'a list).`

Azaz ha `filter`-t egy `'a  $\rightarrow$  bool` típusú függvényre (predikátumra) alkalmazzuk, akkor olyan függvényt ad eredményül, amelyet egy `'a list` típusú listára alkalmazva egy `'a list` típusú listát kapunk.

## Lista redukciója kétoperandusú művelettel (`foldr`, `foldl`)

- Vissza-visszatérő feladat egy lista redukciója kétoperandusú művelettel. Közös, hogy  $n$  db értékből egyetlen értéket kell előállítani (vö. *redukció*).
- `foldr` jobbról balra, `foldl` balról jobbra haladva egy kétoperandusú műveletet (pontosabban egy *párra alkalmazható, prefix pozíciójú függvényt*) alkalmaz egy listára. Példák szorzat és összeg kiszámítására:
 

```
foldr op* 1.0 [] = 1.0; foldl op+ 0 [] = 0;
```

```
foldr op* 1.0 [4.0] = 4.0; foldl op+ 0 [4] = 4;
```

```
foldr op* 1.0 [1.0, 2.0, 3.0, 4.0] = 24.0; foldl op+ 0 [1, 2, 3, 4] = 10;
```
- Jelöljön  $\oplus$  tetszőleges kétoperandusú infix operátort. Akkor
 

```
foldr op⊕ e [x1, x2, ..., xn] = (x1 ⊕ (x2 ⊕ ... ⊕ (xn ⊕ e) ...))
```

```
foldr op⊕ e [] = e
```

```
foldl op⊕ e [x1, x2, ..., xn] = (xn ⊕ ... ⊕ (x2 ⊕ (x1 ⊕ e)) ...)
```

```
foldl op⊕ e [] = e
```
- Asszociatív műveleteknél `foldr` és `foldl` eredménye azonos.

## Példák foldr és foldl alkalmazására

- A  $\oplus$  művelet e operandusa néhány gyakori műveletben – összeadás, szorzás, konjunkció (logikai „és”), alternáció (logikai „vagy”) – a (jobb oldali) *egységelem* szerepét tölti be.
- isum egy egészlista elemeinek összegét, rprod egy valóslista elemeinek szorzatát adja eredményül.
 

```
val isum = foldr op+ 0; val rprod = foldr op+ 1.0;
val isum = foldl op+ 0; val rprod = foldl op+ 1.0;
```
- A length függvény is definiálható foldl vagy foldr felhasználásával. Kétooperandusú műveletként olyan segédfüggvényt (inc) alkalmazunk, amelyik *nem használja* az első paraméterét.
 

```
(* inc : 'a * int -> int (* lengthl, lengthr : 'a list -> int *)
 inc (_, n) = n + 1 *) val lengthl = fn ls => foldl inc 0 ls;
fun inc (_, n) = n + 1; fun lengthr ls = foldr inc 0 ls;

lengthl (explode "tengertanc"); lengthr (explode "hajdu sogar");
```

## Példák foldr és foldl alkalmazására (folyt.)

---

- Egy lista elemeit egy másik lista elé fűzi foldr és foldl, ha kétoperandusú műveletként a *cons* konstruktorfüggvényt – azaz az `op::-`-ot – alkalmazzuk.  
`foldr op:: ys [x1, x2, x3] = (x1 :: (x2 :: (x3 :: ys)))`  
`foldl op:: ys [x1, x2, x3] = (x3 :: (x2 :: (x1 :: ys)))`
- A :: nem asszociatív, ezért foldl és foldr eredménye különböző!

```
(* append : 'a list -> 'a list -> 'a list
 append xs ys = az xs ys elé fűzésével előálló lista *)
fun append xs ys = foldr op:: ys xs;
```

```
(* revApp : 'a list -> 'a list -> 'a list
 revApp xs ys = a megfordított xs ys elé fűzésével előálló lista *)
fun revApp xs ys = foldl op:: ys xs;

append [1, 2, 3] [4, 5, 6] = [1, 2, 3, 4, 5, 6]; (vö. Prolog: append)
revApp [1, 2, 3] [4, 5, 6] = [3, 2, 1, 4, 5, 6]; (vö. Prolog: revapp)
```

## lista: foldr és foldl definíciója

- $\text{foldr } \text{op} \oplus e [x_1, x_2, \dots, x_n] = (x_1 \oplus (x_2 \oplus \dots \oplus (x_n \oplus e) \dots))$   
 $\text{foldr } \text{op} \oplus e [] = e$ 

( $*$  foldr f e xs = az xs elemeire jobbról balra haladva alkalmazott, kétoperandusú, e egységelemű f művelet eredménye)

```
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *
fun foldr f e (x::xs) = f(x, foldr f e xs)
 | foldr f e [] = e;
```
- $\text{foldl } \text{op} \oplus e [x_1, x_2, \dots, x_n] = (x_n \oplus \dots \oplus (x_2 \oplus (x_1 \oplus e) \dots))$   
 $\text{foldl } \text{op} \oplus e [] = e$ 

( $*$  foldl f e xs = az xs elemeire balról jobbra haladva alkalmazott, kétoperandusú, e egységelemű f művelet eredménye)

```
foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *
fun foldl f e (x::xs) = foldl f (f(x, e)) xs
 | foldl f e [] = e;
```

## lista redukciója bal oldali egységelemű függvénnyel (foldl)

- A kivonás művelete balra köt:  $x_1 - x_2 - x_3 - x_4 = ((x_1 - x_2) - x_3) - x_4$ .
- Nem feleltethető meg sem foldr-nek, sem foldl-nek.
 
$$\text{foldr } \text{op} \oplus e [x_1, x_2, \dots, x_n] = (x_1 \oplus (x_2 \oplus (\dots \oplus (x_n \oplus e) \dots)))$$

$$\text{foldl } \text{op} \oplus e [x_1, x_2, \dots, x_n] = (x_n \oplus (\dots \oplus (x_2 \oplus (x_1 \oplus e)) \dots))$$
- Nevezzük foldl-nek a listában *balról jobbra* haladó, alábbi specifikációjú függvényt. Vegyük észre, hogy  $\oplus$  bal oldali egységelemet vár.
 
$$\text{foldl } \text{op} \oplus e [x_1, x_2, \dots, x_n] = ( \dots ((e \oplus x_1) \oplus x_2) \oplus \dots \oplus x_n )$$
- foldl olyan kétargumentumú függvényt vár, amelynek az „egységelem” (valójában: a részeredmény) az *első* argumentuma:  $f : 'a * 'b \rightarrow 'a$ .
 
$$(* \text{ foldl} : ('a * 'b \rightarrow 'a) \rightarrow 'a \rightarrow 'b \text{ list} \rightarrow 'a$$

$$\text{foldl } f \ e \ xs = \text{az } xs \text{ elemeire balról jobbra haladva alkalmazott,}$$

$$\text{ kétoperandusú, } e \text{ egységelemű } f \text{ művelet eredménye } *)$$

```
fun foldl f e (x::xs) = foldl f (f(e, x)) xs
 | foldl f e [] = e;
```



## Példák listaelemek különbségének és hányadosának képzésére

---

- Az  $e$  argumentum aktuális értéke a sorozat *első* eleme – a *kisebbitendő*, ill. az *osztandó*.

```
foldl op- 20 [] = 20; foldl (op div) 180 [] = 180;
foldl op- 20 [5, 6, 7] = foldl (op div) 180 [2, 3, 5] =
 (((20 - 5) - 6) - 7); (((180 div 2) div 3) div 5);
```

- Ha többször használjuk  $e$  műveleteteket, érdemes nekik nevet adni. A kisebbitendő, ill. az osztandó speciális kezelését elrejtjük.

```
fun subtract ns = foldl op- (hd ns) (tl ns);
subtract [20, 5, 6, 7] = (((20 - 5) - 6) - 7);
```

```
fun divide ns = foldl op div (hd ns) (tl ns);
divide [180, 2, 3, 5] = (((180 div 2) div 3) div 5);
```

## Listaelemek különbsége és hányadosa foldl-lel és foldr-rel

- Igazság szerint foldl felesleges: a feladat jól megoldható foldl-lel vagy foldr-rel is.

```
fun subtract1 ns = hd ns - foldl op+ 0 (tl ns);
subtract1 [20, 5, 6, 7] = (((20 - 5) - 6) - 7);
```

```
fun divide1 ns = hd ns div foldl op* 1 (tl ns);
divide1 [180, 2, 3, 5] = (((180 div 2) div 3) div 5);
```

- foldr és foldl típusa, ha egyparaméteres függvénynek tekintjük őket (a -> jobbra köt!):

```
foldr, foldl : ('a * 'b -> 'b) -> ('b -> 'a list -> 'b)
```

Azaz ha foldr-t vagy foldl-t egy 'a -> \* 'b -> 'b típusú függvényre alkalmazzuk, akkor olyan függvényt ad eredményül, amelyet egy 'b típusú elemszámlálóval és egy 'a list típusú listára alkalmazva 'b típusú (redukált) értéket kapunk.

# KIFEJZÉSEK KIÉRTÉKELÉSE



## Mohó kiértékelés: faktoriális kiszámítása naív rekurzívóval

- A faktoriális matematikai definíciója és megvalósítása SML-ben

```
fac 0 = 1 fac n = n * fac (n - 1)

(* fac : int -> int (--) fontos a klózok sorrendje! --)
 fac n = n!
PRE n >= 0 *)

fun fac 0 = 1 | fac n = n * fac(n-1);
```
- fac mohó kiértékelése  $n = 4$  esetén (egyes triviális lépéseket elhagynk).  
$$\begin{aligned} \text{fac } 4 &\rightarrow 4 * \text{fac } (4-1) \rightarrow 4 * \text{fac } 3 \rightarrow 4 * (3 * \text{fac } (3-1)) \rightarrow \\ &\rightarrow 4 * (3 * \text{fac } (2)) \rightarrow \dots \rightarrow 4 * (3 * (2 * (1 * 1))) \rightarrow \dots \rightarrow 24 \end{aligned}$$
- A rekurzív kiértékelés követi a matematikai definíciót.
- Rontja a hatékonyságot, hogy a rekurzív végrehajtás során minden eredményt a veremben tárolni kell.
- Ha a szorzás asszociativitását kihasználjuk, nem kell tárolni az összes tényezőt, csak az aktuális részeredményt.

## Faktorális kiszámítása jobbrekurzíóval

- Először egy *akkumulátort* (gyűjtőargumentumot) használó *segédfüggvényt* definiálunk. Vegyük észre, hogy a rekurzív hívás *jobbrekurzív*: eredménye közvetlenül, további műveletek elvégzése nélkül adja a végeredményt.

```
(* faci : int -> int -> int (-- fontos a klózok sorrendje! --)
 faci n p = p * n! (-- p az akkumulátor --)
*)
```

```
fun faci 0 p = p
 | faci n p = faci (n-1) (n*p);
```

- faci*-t felhasználjuk az egyparaméteres *fac* függvény definiálására. Az *akkumulátornak* alkalmas *kezdőértéket* adunk.

```
(* fac : int -> int
 fac n = n!
 PRE n >= 0
*)
fun fac n = faci n 1;
```

## Faktoriális kiszámítása jobbrekurzióval (folyt.)

- `fac` nem rekurzív, ezért csak `faci` kiértékelését vizsgáljuk (egyes triviális lépéseket összevonunk).
- A függvény:  $\text{fun } \text{faci } 0 \text{ } p = p \mid \text{faci } n \text{ } p = \text{faci } (n-1) \text{ } (n*p)$   
 $\text{faci } 4 \text{ } 1 \rightarrow \text{faci } (4-1) \text{ } (4*1) \rightarrow \text{faci } 3 \text{ } 4 \rightarrow \text{faci } (3-1) \text{ } (3*4) \rightarrow$   
 $\rightarrow \text{faci } 2 \text{ } 12 \rightarrow \dots \rightarrow \text{faci } 0 \text{ } 24 \rightarrow 24$
- Kiértékelés közben a `p` *akkumulátor* gyűjti a részeredményt, ezért `faci` tárigénye állandó.
- A kiértékelés *iteratív*.
- A jó fordítóprogram felismeri a jobbrekurziót, és hatékony tárgykódot állít elő: az argumentumokat frissíthető lokális változókkban tárolja, a rekurziót iterációval helyettesíti.
- A jobbrekurziót *terminális rekurzió*nak is nevezik (angolul: *tail* vagy *terminal recursion*).
- `foldl` jobbrekurzív, e argumentuma akkumulátorként viselkedik.

## Lokális kifejezés

---

- *Lokális kifejezést* használunk, ha ismétlődő rész kifejezéseket *csak egyszer* akarunk kiszámítani, vagy akkor, ha bizonyos értékeket a program többi része elől *el akarunk rejtetni*.
- Szintaxisa: `let d in e end, ahol`
- *d* nemüres deklarációsorozat,
- *e* nemüres kifejezés.

- Példa:

```
fun fac n =
 let
 fun faci 0 p = p
 | faci n p = faci (n-1) (n*p)
 in
 faci n 1
 end
```

## Lokális deklaráció

---

- *Lokális deklarációt* használunk olyan értékek bevezetésére, amelyeket a program többi része előtt *el akarunk rejtetni*.
- Szintaxisa: `local d1 in d2 end, ahol`
- *d1* és *d2* nemüres deklarációsorozatok.

• Példa:

```
local
 fun faci 0 p = p
 | faci n p = faci (n-1) (n*p)
 in
 fun fac n = faci n 1
 end
```



# LOGIKAI MŰVELETEK



## Logikai műveletek

---

- Típusnév: `bool`, adatkonstruktorok: `false`, `true`, beépített függvény: `not`.
- *Lusta kiértékelésű* beépített operátorok
- Három argumentumú: `if b then e1 else e2`.  
Nem értékeli ki az `e2`-t, ha `a` igaz, ill. az `e1`-et, ha `a` hamis.
- Két argumentumúak:  
`e1 andalso e2` : nem értékeli ki az `e2`-t, ha az `e1` hamis.  
`e1 orelse e2` : nem értékeli ki az `e2`-t, ha az `e1` igaz.

- Mind a három csupán szintaktikai édesítőszerszer!

- `if b then e1 else e2  $\equiv$  (fn true => e1 | false => e2) b`
- `e1 andalso e2  $\equiv$  (fn true => e2 | false => false) e1`
- `e1 orelse e2  $\equiv$  (fn true => true | false => e2) e1`
- `fun ifThenElse b = (fn true => e1 | false => e2) b; ifThenElse true;`

- Tipikus hiba: `if exp then true else false !!!`

## Logikai műveletek (folyt.)

- Nyilvánvaló: `andlso` és `orelse` kifejezhető `if-then-else`-szel is.
  - `if e1 then e2 else false  $\equiv$  e1 andlso e2`
  - `if e1 then true else e2  $\equiv$  e1 orelse e2`
- Használjuk az `andlso`-t és az `orelse`-t az `if-then-else` helyett, ahol csak lehet: olvashatóbb lesz a program.
- Lusta kiértékelésű függvényt a programozó nem definiálhat az SML-ben. Az SML, mielőtt egy függvényt alkalmazna az (egyszerű vagy összetett) argumentumára, kiértékeli.

- Az `andlso` és az `orelse` *mohó kiértékelésű* megfelelői:

```

(* && (a, b) = a /\ b
 && : bool * bool -> bool
 *)
fun op&& (a, b) = a andlso b;
infix 2 &&;

(* || (a, b) = a \/ b
 || : bool * bool -> bool
 *)
fun op|| (a, b) = a orelse b;
infix 1 ||;
```

# LISTÁK



## Listák összefűzése és megfordítása

- Listák összefűzése és megfordítása beépített függvényekkel: @, rev és revAppend (List könyvtár).
- @ a fun append (xs, ys) = foldr op:: ys xs beépített megfelelője: infix, 5-ös precedenciájú, jobbra köt, típusa 'a list \* 'a list -> 'a list.
- revAppend a fun revApp (xs, ys) = foldl op:: ys xs beépített megfelelője: prefix, típusa 'a list \* 'a list -> 'a list.
- rev a fun rev xs = foldl op:: [] xs beépített megfelelője: prefix, típusa 'a list -> 'a list (vö. revApp).

- Az  $[m, n)$  tartományba eső egészek listája: a kézenfekvő megoldás

```
(* upto m n = az [m, n) tartományba eső egészek listája
 upto : int -> int -> int list *)
fun upto m n = if m < n then m :: upto (m+1) n else [];
```

## Listák összefüzése és megfordítása

---

- Az  $[m, n)$  tartományba eső egészek listája: jobbrekurzív megoldás

```
fun upto m n =
 let (* az up számára az n állandó érték,
 ezért nem kell argumentumként átadni *)
 fun up zs m = if m < n then up (m::zs) (m+1) else rev zs
 in up [] m
 end;
```

- Az  $[m, n)$  tartományba eső egészek listája: hatékony jobbrekurzív megoldás

```
fun upto m n =
 let (* hátról visszafelé haladva építjük föl a listát,
 ezért a végén nem kell megfordítani *)
 fun up zs n = if m < n then up (n-1::zs) (n-1) else zs
 in up [] n
 end;
```

## Lista legnagyobb elemének megkeresése

- Egy egészlista legnagyobb elemének kiválasztásához szükségünk van az `Int.max` függvényre.
- Üres listának nincs legnagyobb eleme,
- egyelemű listában az egyetlen elem a legnagyobb,
- legalább kételemű lista legnagyobb elemét úgy kapjuk, hogy az első elem és a maradéklista elemeinek legnagyobbika közül kiválasztjuk a legnagyobbat.

```
(* maxl ns = az ns egészlista legnagyobb eleme
 maxl : int list -> int *)
```

```
fun maxl [n] = n
 | maxl (n::ns) = Int.max(n, maxl ns)
 | maxl [] = raise Empty;
```

- **max egy változata egészszekre**

```
fun max (n, m) = if n > m then n else m
```

## lista legnagyobb elemének megkeresése (folyt.)

- Hogyan tehető polimorfá a `maxl` függvényt? Magasabbrendű, ún. generikus függvényként definiáljuk: *argumentumként* kapja azt a többszörösen terhelhető függvényt, amely két érték közül a nagyobbikat kiválasztja.

```
(* maxl max ns = az ns lista legnagyobb eleme
 maxl : ('a * 'a -> 'a) -> 'a list -> 'a *)

fun maxl max [n] = n
 | maxl max (n::ns) = max(n, maxl max ns)
 | maxl max [] = raise Empty;
```

- max mindig ugyanaz, mégis újra és újra átadjuk argumentumként a rekurzív ágban. Javítja a hatékonyságot, ha *lokális kifejezést* használunk. (Lokális deklaráció használata most nem segítene. Miért nem?)

```
fun maxl max ns = let fun mxl [n] = n
 | mxl (n::ns) = max(n, mxl ns)
 | mxl [] = raise Empty
 in mxl ns end;
```



## lista (folyt.)

---

### • Változatok max-ra

```
(* charMax : char * char -> char *)
fun charMax (n, m) = if ord n > ord m then n else m;
```

```
(* pairMax : ((int * real) * (int * real)) -> (int * real)
fun pairMax (n as (n1 : int, n2 : real), m as (m1, m2)) =
 if n1 > m1 orelse n1 = m1 andalso n2 >= m2 then n else m;
```

### • concat xss = az xss-beli listákat egy listába fűzi. Könyvtári változata: List.concat.

```
(* concat : 'a list list -> 'a list *)
fun concat xss = foldr op@ [] xss;
```

### • ListPair.zip két lista páronkénti elemeiből álló párok listáját, ListPair.unzip párok listájából két listát ad eredményül.

## Adott számú elem egy lista elejéről és végéről (take, drop)

- Legyen  $xs = [x_0, x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_{n-1}]$ , akkor
  - take(xs, i) =  $[x_0, x_1, \dots, x_{i-1}]$  és drop(xs, i) =  $[x_i, x_{i+1}, \dots, x_{n-1}]$ .
  - (\* take (xs, i) = xs, ha  $i < 0$ ;  
az xs első i db eleméből álló lista, ha  $i \geq 0$   
take : 'a list \* int -> 'a list \*)
  - fun take (\_, 0) = []  
| take ([], \_) = []  
| take (x::xs, i) = x :: take(xs, i-1);
  - (\* drop(xs, i) = xs, ha  $i < 0$ ;  
az xs első i db elemének elhagyásával előálló lista, ha  $i \geq 0$   
drop : 'a list \* int -> 'a list \*)
  - fun drop ([], \_) = []  
| drop (x::xs, i) = if i > 0 then drop (xs, i-1) else x::xs;
- Könyvtári változatuk, List.take és List.drop  $i < 0$  vagy  $i > \text{length } xs$  esetén Subscript kivételt jelez.

## Halmazműveletek

---

- ismem igaz értéket ad eredményül, ha a keresett elem benne van a listában.

```
(* isMem(x, ys) = x eleme-e ys-nek
 isMem : 'a * 'a list -> bool *)
fun isMem (x, y::ys) = x = y orelse isMem (x, ys)
 | isMem (_, []) = false;
infix isMem;
```

- newMem egy új elemet rak be egy listába, ha még nincs benne.

```
(* newMem(x, xs) = [x] és xs listaként ábrázolt uniója
 newMem : 'a * 'a list -> 'a list *)
fun newMem (x, xs) = if x isMem xs then xs else x::xs;
```

**newMem, ha a sorrendtől eltekintünk, halmazt hoz létre.**

## Halmazműveletek (folyt.)

---

- `setof` halmazt készít egy listából úgy, hogy kiszedi belőle az ismétlődő elemeket. Rossz hatékonyságú.

```
(* setof xs = xs elemeinek listaként ábrázolt halmaza
 setof : ''a list -> ''a list *)
fun setof (x::xs) = newMem (x, setof xs)
 | setof [] = [];
```

- Szerencsésebb a halmazokat a megszokott halmazműveletekkel kezelni. Öt halmazműveletet definiálunk:

- unió ( $\text{union}, S \cup T$ ),
- metszet ( $\text{inter}, S \cap T$ ),
- részhalmaza-e ( $\text{isSubset}, T \subseteq S$ ),
- egyenlők-e ( $\text{isSetEq}, S = T$ ),
- hatványhalmaz ( $\text{powerset}, pS$ ).

## Halmazműveletek (folyt.)

---

- Listaként kezeljük a halmazokat, később hatékonyabb ábrázolást választhatunk, pl. rendezett listát vagy bináris fát.

- Két halmaz uniója

```
(* union(xs, ys) = az xs és ys elemeiből álló halmazok uniója
 union : ''a list * ''a list -> ''a list *)
fun union (x::xs, ys) = newMem(x, union(xs, ys))
 | union ([], ys) = ys;
```

- Két halmaz metszete

```
(* inter(xs, ys) = az xs és ys elemeiből álló halmazok metszete
 inter : ''a list * ''a list -> ''a list *)
fun inter (x::xs, ys) = let val zs = inter(xs, ys)
 in if x isMem ys then x::zs else zs
 end
 | inter ([], _) = [];
```

## Halmazműveletek (folyt.)

---

- Részhalmaz-e egy halmaz egy másiknak?

```
(* isSubset (xs, ys) = az xs elemeiből álló halmaz részhalmaza-e
 az ys elemeiből álló halmaznak
isSubset : ''a list * ''a list -> bool *)

fun isSubset (x::xs, ys) = (x isMem ys) andalso isSubset(xs, ys)
 | isSubset ([], _) = true;
infix isSubset;
```

- Két halmaz egyenlősége

A listák egyenlőségvizsgálata beépített művelet az SML-ben. Halmazokra mégsem használható, mert pl. [3, 4] és [4, 3, 4] listaként ugyan különböznek, de halmazként egyenlők.

```
(* isSetEq(xs, ys) = az xs és ys elemeiből álló halmazok egyenlők-e
 isSetEq : ''a list * ''a list -> bool *)

fun isSetEq (xs, ys) = (xs isSubset ys) andalso (ys isSubset xs);
```

## Halmazműveletek (folyt.)

### • Halmaz hatványhalmaza

A hatványhalmaz egy halmaz *összes* részhalmazának a halmaza, az eredeti halmazt és az üres halmazt is beleértve.

Jelöljük  $S$ -sel az eredeti halmazt.  $S$  hatványhalmazát úgy állíthatjuk elő, hogy  $S$ -ből kivesszünk egy  $x$  elemet, és aztán *rekurzív módon* előállítsuk az  $S - \{x\}$  hatványhalmazát.

Ha tetszőleges  $T$  halmazra  $T \subseteq S - \{x\}$ , akkor  $T \subseteq S$  és  $T \cup \{x\} \subseteq S$ , így mind  $T$ , mind  $T \cup \{x\}$  eleme  $S$  hatványhalmazának.

A pws függvényben a base argumentum gyűjti a hatványhalmaz elemeit; kezdetben üresnek kell lennie.

```
(* pws(xs, base) = az xs halmaz hatványhalmazának és
 a base halmaznak az uniója
 pws : 'a list * 'a list -> 'a list list *)
fun pws (x::xs, base) = pws(xs, base) @ pws(xs, x::base)
 | pws ([], base) = [base];
```

## Halmazműveletek (folyt.)

---

### ● Halmaz hatványhalmaza (folyt.)

A `pws(xs, base) @ pws(xs, x::base)` kifejezésben `pws(xs, base)` valószínűleg az  $S - \{x\}$  rekurzív hívást (hiszen `x::xs` felel meg  $S$ -nek), azaz állítja elő az összes olyan halmazt, amelyekben `x` nincs benne.

`pws(xs, x::base)` ugyancsak rekurzív módon `base`-ben gyűjti az `x` elemeket, vagyis előállítja az összes olyan halmazt, amelyben `x` benne van.

```
(* powerset xs = az xs halmaz hatványhalmaza
 powerset : 'a list -> 'a list list *)
fun powerset xs = pws(xs, []);
```



# KIFEJZÉSEK KIÉRTÉKELÉSE



## Sztatikus és dinamikus kötés, mohó és lusta kiértékelés

---

- *Sztatikus kötés*: a formális paraméter összes előfordulását *fordítási időben* helyettesítjük az argumentummal (aktuális paraméterrel) a függvény (eljárás) törzsében.
  - *Dinamikus kötés*: a formális paraméter összes előfordulását *futási időben* helyettesítjük az argumentummal (aktuális paraméterrel) a függvény (eljárás) törzsében.
- Kérdés, hogy az aktuális paraméterként átadott kifejezést az értelmező mikor értékeli ki: a behelyettesítés *előtt* vagy *után*.
- *Mohó kiértékelés*: a behelyettesítés *előtt* kiértékeljük az *összes* argumentumot (más megnevezések: *érték szerinti* paraméterátadás, *eager evaluation*, *call-by-value*).
  - *Lusta kiértékelés*: a behelyettesítés *után* csak azt az argumentumot értékeljük ki, amelyekre szükség van, és csak akkor, amikor *szükség van* rá (más megnevezések: *szükség szerinti* paraméterátadás, *lazy evaluation*, *call-by-need*.)

## A *mohó* és a *lusta* kiértékelés összevetése

---

- Más paraméteradási eljárások
- *néu szerinti* paraméteradás (*call-by-name*, Algol).
- *hivatkozás szerinti* paraméteradás (*call-by-reference*, Pascal, C stb.)
- Nézzünk két egyszerű függvényt!

```
(* sq : int -> int (* zero : int -> int
 sq x = x négyzete *) zero x = az x-től függetlenül mindig 0 *)
fun sq x = x * x; fun zero x = 0;
```

Az `sq` függvény argumentumát *lusta kiértékelés* esetén *kétszer* számítjuk ki.

A `zero` függvény argumentumát *mohó kiértékelés* esetén *feleslegesen* számítjuk ki, mert nem használjuk semmire.

## Mohó kiértékelés

- Emlékeztető: az  $f$   $e$  értékét úgy számítjuk ki, hogy először az  $f$  függvényértéket adó kifejezés, majd az  $e$  kifejezés értékét határozzuk meg, és ezután helyettesítjük az  $f$  törzsében a formális paraméter minden előfordulását az  $e$  értékével.

```
fun sq x = x * x; fun zero x = 0;
```

- Nézzük  $\text{sq}(\text{sq}(\text{sq } 2))$  egyszerűsítését! (Az egyszerűsítés eredménye tovább már nem egyszerűsíthető, ún. *kanonikus* kifejezés.)

$\text{sq}$  három alkalmazásából csak a harmadiknak kanonikus kifejezés az argumentuma.

```
sq(sq(sq 2)) → sq(sq(2*2)) → sq(sq 4) → sq(4*4) → sq 16 →
16*16 → 256
```

Az utolsó lépés kivételével  $\text{zero}(\text{sq}(\text{sq}(\text{sq } 2)))$  egyszerűsítési lépései ugyanezek, pedig az eredmény nyilvánvalóan 0!

Mohó kiértékelés mellett a számítógépet feleslegesen dolgoztatjuk!

## Név szerinti paraméterátadás

- Egy függvény alkalmazása előtt sokszor nemcsak fölösleges, hanem káros is előre kiszámítani az argumentumokat, mert végtelen rekurzíó vagy illegális művelet (indexhatár-túllépés, 0-val való osztás stb.) lehet az „eredménye”.
- Az Algol *név szerinti* paraméterátadása a formális paraméter összes előfordulását az argumentumként átadott *teljes* (nem kanonikus) *kifejezéssel* helyettesíti a függvény törzsében.

Ezért `zero(sq(sq(2)))` *név szerinti* paraméterátadás esetén *azonnal*, az argumentum kiértékelése nélkül 0-t ad eredményül!

```
fun sq x = x * x; fun zero x = 0;
```

A *név szerinti* paraméterátadás sem mindig kedvező: pl. `sq(sq(2))` esetén `sq` mindegyik alkalmazása *megkét-szerezi* az argumentumok számát. Aligha ezt akarjuk!

```
sq(sq(sq(2))) → sq(sq(2)) * sq(sq(2)) → (sq 2 * sq 2) * sq(sq(2)) →
((2*2) * sq 2) * sq(sq(2)) → ... → (4*(2*2) * sq(sq(2))) → ...
```

## Lusta kiértékelés

- *Lusta kiértékelés* esetén minden argumentumot csak egyszer kell kiértékelni: akkor, amikor *először* van rá szükség. Az argumentum összes előfordulását egy *rejtett hivatkozással* helyettesítjük (mivel *el van rejtve* a programozó elől, biztonságos): amikor a számítógép az argumentumot először kiértékeli, a kapott értéket elrakja, és később az összes olyan helyen, ahol szükség lesz rá, felhasználja.
- A függvényeket és argumentumaikat *irányított gráffal* ábrázolják: a gráf egy részének kiértékelésekor a gráfot az eredményül kapott értékkel frissítik a számítógépben (ezért nevezik *gráfredukciónak*).
- A lusta kiértékeléshez bonyolult nyilvántartást kell vezetni (időigényes!).
- A lusta kiértékelés működési elvének megértéséhez irányított gráf helyett most  $x = [E]$  -vel jelöljük, hogy az  $x$  *összes előfordulása* osztozik az  $E$  értéken.

## Lusta kiértékelés

- Nézzük pl. `sq(sq(sq 2))` lusta kiértékelését!

```
fun sq x = x * x; fun zero x = 0;
```

( $x = [E]$  jelentése: az  $x$  összes előfordulása osztozik az  $E$  értékén.)

```
sq(sq(sq 2)) → x * x [x = sq(sq 2)] → x * x [x = y * y] [y = sq 2] →
x * x [x = y * y] [y = 2 * 2] → x * x [x = y * y] [y = 4] →
x * x [x = 4 * 4] → x * x [x = 16] → 16 * 16 → 256
```

- Gyakran nyerünk, de néha veszítünk a lusta kiértékeléssel.

Láttuk, hogy `fun fac i(0, p) = p | fac i(n, p) = fac i(n-1, n*p)` *mohó kiértékelés* esetén hatékonyabb `fac`-nál, mert az `n*p` szorzást azonnal végrehajtja. *Lusta kiértékelés* esetén az `n`-et azonnal kiszámítaná (szükség van `n` értékére az implicit `n = 0` vizsgálathoz), a `p` kiértékelését azonban a szorzások akkumulálásával késleltetné:

```
fac i(4, 1) → fac i(4-1, 4*1) → fac i(3-1, 3*(4*1)) →
fac i(2-1, 2*(3*(4*1))) ... → 24
```

# ÖSSZETETT ADATTÍPUSOK





## Unnes és típusa

- Két különböző típusú értékből rekordot vagy párt képezhetünk. Pl.  
 $\{x = 2, y = 1.0\} : \{x : \text{int}, y : \text{real}\}$  és  $(2, 1.0) : (\text{int} * \text{real})$ .
  - A pár is szintaktikai édesítőszers. Pl.  
 $(2, 1.0) = \{1 = 2, 2 = 1.0\} = \{2 = 1.0, 1 = 2\}$ , de  $(2, 1.0)$  és  $\{1 = 1.0, 2 = 2\}$  különböző típusúak. Az 1 és a 2 *mezőnevek* (vö. *syntaxis*).
  - Rekordot kettőnél több értékből is összeállíthatunk. Pl.  
 $\{\text{nev} = \text{"Bea"}, \text{tel} = 3192144, \text{kor} = 19\} : \{\text{kor} : \text{int}, \text{nev} : \text{string}, \text{tel} : \text{int}\}$ .
- Egy hasonló rekord egészszám-mezőnevekkel:  
 $\{1 = \text{"Bea"}, 3 = 3192144, 2 = 19\} : \{1 : \text{string}, 2 : \text{int}, 3 : \text{int}$ .
- Az *utóbbi* azonos az alábbi *enessel* (n-es, n-tuple):  
 $(\text{"Bea"}, 19, 3192144) : (\text{string} * \text{int} * \text{int})$ ,
- azaz  $(\text{string} * \text{int} * \text{int}) \equiv \{1 = \text{string}, 2 = \text{int}, 3 = \text{int}\}$ .
- Egy rekordban a tagok sorrendje közömbös, az értékeket a mezőnév azonosítja.

## Ennes és típusa (folyt.)

- Egy ennesben a tagok sorrendje meghatározó! Pl. (2, 1.0) : (int \* real), de (1.0, 2) : (real \* int). A két ennes különböző!

- Ennes lehet függvény argumentuma és eredménye, összetett adat eleme stb. Példa: Fibonacci-számok iterációval.

A definíció:  $F_0 = 0; F_1 = 1; F_n = F_{n-2} + F_{n-1}, n > 1$ .

```
(* iterfib(n, (prev, curr)) = a (prev, curr) Fibonacci-számpárt követő
 n-edik Fibonacci-szám (n > 0)
 iterfib : int * (int * int) -> int *)
fun iterfib (1, (prev, curr)) = curr
 | iterfib (n, (prev, curr)) = iterfib(n - 1, (curr, prev + curr));

(* fib n = az n-edik Fibonacci-szám
 fib : int -> int *)
fun fib 0 = 0
 | fib n = iterfib(n, (0, 1));
```

# FELHASZNÁLÓI ADATTÍPUSOK



## A datatype deklaráció

---

- person néven új összetett típust hozunk létre:

```
datatype person = King
 | Peer of string * string * int
 | Knight of string
 | Peasant of string;
```

- Az új típusnak négy *adatkonstruktor*a (röviden: *konstruktor*a) van: King, Peer, Knight és Peasant.
- King ún. *adatkonstruktorállandó*, a többi ún. *adatkonstruktorfüggvény*.
- Az adatkonstruktoroknak is van típusuk:

```
King : person
Peer : string * string * int -> person
Knight : string -> person
Peasant : string -> person
```

## A datatype deklaráció (folyt.)

---

- King (király) csak egy van, ezért definiálhattuk konstruktorállandóként.
- A Peer-t (főnembst) nemesi címe (string), birtokának neve (string) és sorszáma (int) azonosítja.
- A Knight-ot (lovagot) és a Peasant-ot (parasztot) csupán a neve (string) azonosítja.
- Példa a person adattípus alkalmazására:
  - val persons = [King, Peasant "Jack Cade", Knight "Gawain",  
Peer("Duke", "Norfolk", 9)];
  - > val persons = [King, Peasant "Jack Cade", ...] : person list
- Az egyes esetek mintaillesztéssel választhatók szét.
- Minden esetet le kell fedni mintával; ha nem, figyelmeztetést kapunk.
- A minták tetszőlegesen összetettek lehetnek.

## A datatype deklaráció (folyt.)

- Az alábbi példában a négy közül az egyik a Peasant name *minta*, és benne name a *mintaazonosító*.

```
(* title p = p megszólítása
 title : person -> string *)
fun title King = "His Majesty the King "
| title (Peer (deg, ter, _)) = "The " ^ deg ^ " of " ^ ter
| title (Knight name) = "Sir " ^ name
| title (Peasant name) = name;
```

- A sirs függvény az összes Knight nevét összegyűjti a person típusú személyek egy listájából (a változatok sorrendje *fontos* az \_ miatt!):

```
(* sirs ps = az összes Knight nevének listája
 sirs : person list -> string list *)
fun sirs [] = []
| sirs ((Knight s)::ps) = s::sirs ps
| sirs (_::ps) = sirs ps;
```

## A datatype deklaráció (folyt.)

---

- Ha más lenne a változatok sorrendje, a `_::ps` minta nemcsak a `King-re`, a `Peer-re` és a `Peasant-ra` illeszkedne (ti. ezek helyett áll a példában), hanem a `Knight-ra` is.
- Az összes diszjunkt eset felsorolása segíti az algoritmus helyességének belátását, bizonyítását.
- Azért vontunk össze három esetet egyetlen változatban, mert a részletezésük hosszabbá tenné a program szövegét is, végrehajtását is.
- A bizonyítás nem okoz gondot, ha a függvény harmadik sorát (`sirs (_::ps) = sirs ps`) *feltételes egyetlennek* tekintjük:

```
sirs(p::ps) = sirs ps if $\forall s.p \neq \text{Knight } s.$
```

## A datatype deklaráció (folyt.)

---

- A sorrend még fontosabb a következő példában, amelyben személyek hierarchiáját vizsgáljuk. Itt 16 helyett csak 7 esetet kell megkülönböztetnünk: azokat, amelyek *igaz* eredményt adnak.

```
(* superior (p, r)= igaz, ha p magasabb rangú r-nél
 superior : person * person -> bool *)
fun superior (King, Peer _) = true
| superior (King, Knight _) = true
| superior (King, Peasant _) = true
| superior (Peer _, Knight _) = true
| superior (Peer _, Peasant _) = true
| superior (Knight _, Peasant _) = true
| superior _ = false;
```



## A felsorolásos típus datatype deklarációjával

---

- Gyakori, hogy egy név csak néhány különböző értéket vehet fel (azaz a név által felvehető értékek halmaza kis számosságú), ilyen esetben érdemes *felsorolásos típust* létrehozni a datatype deklarációjával. Pl.

```
datatype degree = Duke | Marquis | Earl | Viscount | Baron;
```

- A felsorolásos típusnak csak *konstruktorállandói* vannak. Az új típus alkalmazásához a person típust újra deklarálnunk kell:

```
datatype person = King
 | Pear of degree * string * int
 | Knight of string
 | Peasant of string;
```

## A felsorolásos típus datatype deklarációval (folyt.)

---

- A degree típusú adatok feldolgozásakor külön-külön elemezzük az előforduló eseteket, pl.

```
(* lady p = p főnemes hitvesének rangja
 lady : degree -> string *)
fun lady Duke = "Duchess "
 | lady Marquis = "Marchioness"
 | lady Earl = "Countess"
 | lady Viscount = "Viscountess"
 | lady Baron = "Baroness";
```

- A belső bool típushoz hasonló Bool típust és hozzá a Not függvényt például így is deklarálhatnánk, ill. definiálhatnánk:

```
datatype Bool = True | False;
(* Not b = b negáltja
 Not : Bool -> Bool *)
fun Not True = False | Not False = True;
```

## Polimorf adattípusok

---

- Láttuk, hogy a `List` *postfix* pozíciójú *típusoperátor*, nem típus: a `datatype` deklaráció az adatkonstruktorok mellett *típuskonstruktor* is létrehoz.
- A belső '`a`' list típushoz hasonló '`a`' `List` listát és vele együtt a `Nil` és a `Cons` *adatkonstruktorokat* például így definiálhatjuk:  

```
datatype 'a List = Nil | Cons of 'a * 'a List;
```
- A `Cons` *adatkonstruktorfüggvény* alkalmazásával elég körülményes a listák létrehozása. Az 1, 2, 3, 4 sorozatot például így kell megadni:  

```
Cons(1, Cons(2, Cons(3, Cons(4, Nil))));
```
- Bevezethetjük az *infix* pozíciójú `::` *adatkonstruktoroperátort*:  

```
infix 5 :: : val op :: = Cons;
```
- A *hatospontot* közvetlenül a típusdeklarációban is definiálhatjuk:  

```
infix 5 :: : datatype 'a List = Nil | :: : of 'a * 'a List;
```

## Polimorf adattípusok: megkülönböztetett egyesítés

- Következő példánk két típus *megkülönböztetett egyesítése*, más néven diszjunkt uniója:  
  
`datatype ('a, 'b) disun = In1 of 'a | In2 of 'b;`
- Itt három dolgot definiáltunk:
  1. a kétargumentumú `disun` típusoperátort,
  2. az `In1 : 'a -> ('a, 'b) disun` és
  3. az `In2 : 'b -> ('a, 'b) disun` adatkonstruktorfüggvényeket.
- `('a, 'b) disun` az `'a` és `'b` típusok megkülönböztetett egyesítése.  
*Megkülönböztetettnek* nevezzük az egyesítést, mert később is bármikor meg tudjuk mondani, hogy egy `('a, 'b) disun` típusú pár egyik vagy másik eleme melyik alaptípusból származik. Az új típusba tartozó értékek `In1 x` alakúak, ha `x 'a` típusú, és `In2 y` alakúak, ha `y 'b` típusú.
- Az `In1` és `In2` konstruktorfüggvények olyan *címkének* tekinthetők, amelyek az `'a` típust megkülönböztetik a `'b` típustól.

## Megkülönböztetett egyesítés (folyt.)

---

- A megkülönböztetett egyesítés lehetővé teszi, hogy különböző típusokat használjunk ott, ahol egyébként csak egyetlen típust használhatnánk (vö. objektum-orientált programozás, ahol pl. egy *alakzat* osztálynak *téglalap*, *háromszög* vagy *kör* nevű leszármazottai lehetnek).

- Az SML-ben megkülönböztetett egyesítéssel tudunk létrehozni *különböző típusú elemekből* álló listát:

```
[In2 King, In1 "Skócia"] : ((string, person) disun) list
[In1 "zsarnok", In2 1040] : ((string, int) disun) list
```

- A lehetséges eseteket most is *mintaillesztéssel* elemezhetjük, pl.

```
(* concat d = a d diszjunkt unió In1 címkejű elemeinek konkatenációja
 concat : (string, 'a) disun list -> string *)
fun concat [] = ""
 | concat (In1 s :: ls) = s ^ concat ls
 | concat (In2 _ :: ls) = concat ls;
```

## Megkülönböztetett egyesítés (folyt.)

---

- Egy példa concat alkalmazására:
  - concat [In1 "Ű!", In2 King, In1 "Skócia"];  
> val it = "Ű! Skócia : string"
- Az In1 konstruktorfüggvény típusa 'a -> ('a, 'b) disun, ezért a string típusú "Ű!" argumentumra alkalmazva (string, 'b) disun típusú érték az eredmény.
- Az In2 konstruktorfüggvény típusa 'b -> ('a, 'b) disun, ezért a person típusú King kifejezésre alkalmazva ('a, person) disun típusú érték az eredmény.
- Az [In1 "Ű!", In2 King, In1 "Skócia"] kifejezésben mindkét alaptípust lekötjük, ezért ennek a listának a típusa: ((string, person) disun) list.
- Az [In2 "Ű", In2 King, In1 "Skócia"] kifejezés kiértékelése hibajelzést eredményez, mert a 'b típusváltozót nem lehet ugyanabban a kifejezésben egyszer így, másszor úgy lekötni.

# BINÁRIS FÁK

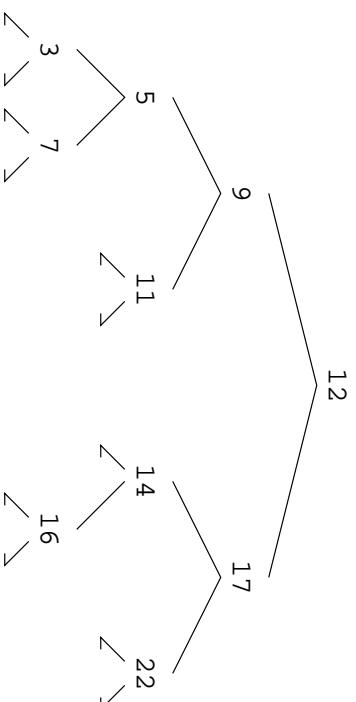


## Bináris fák datatype deklarációval

- A listához hasonlóan rekurzív adatszerkezet a *fa*.
- Először olyan bináris fát deklarálunk, amelynek a levelei üresek, a csomópontjaiban pedig előbb a bal részfát, majd az 'a típusú értéket, és végül a jobb részfát adjuk meg:

`datatype 'a tree = L | B of 'a tree * 'a * 'a tree;`

- Tekintsük például az alábbi fát:



- Az 'a tree adattípus L és B adatkonstruktoraival ez a fa pl. a következő lapon látható módon írható le.



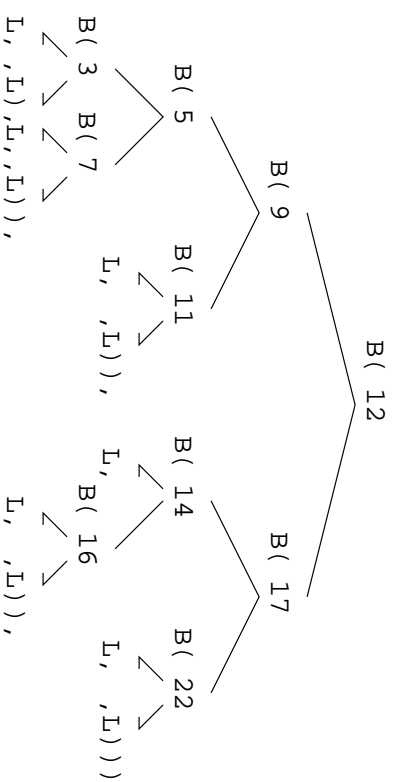
## Bináris fák datatype deklarációval (folyt.)

```

B(B(B(B(L,3,L),
 5,
 B(L,7,L)
),
 9,
 B(L,11,L)
),
 12,
 B(B(L,
 14,
 B(L,16,L)
),
 17,
 B(L,22,L)
)
);

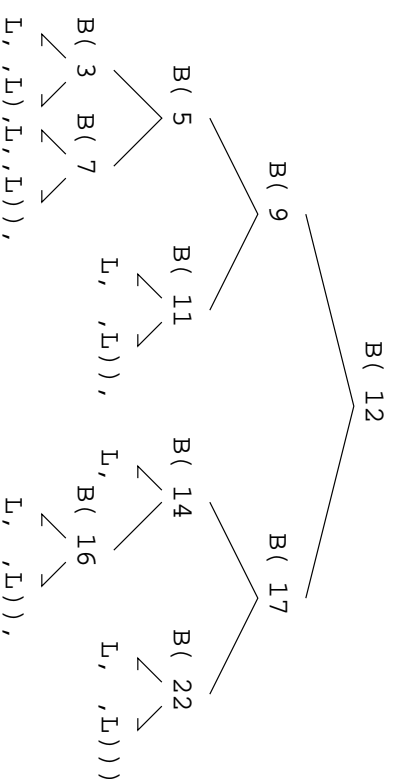
```

A bal oldali kifejezést elég nehéz átlátni. A fastuktúra szöveges leírását megkönnyíti, ha az ábrába beírjuk a megfelelő adatkonstruktorokat.



## Bináris fák datatype deklarációval (folyt.)

- A fastruktúra szöveges leírása átláthatóbb, ha az egyes részfáknak nevet adunk, és a részfákból építjük fel a teljes fát:



```

val tr3 = B(L,3,L); val tr7 = B(L,7,L);
val tr5 = B(tr3,5,tr7); val tr11 = B(L,11,L);
val tr9 = B(tr5,9,tr11); val tr16 = B(L,16,L);
val tr14 = B(L,14,tr16); val tr22 = B(L,22,L);
val tr17 = B(tr14,17,tr22); val tr12 = B(tr9,12,tr17);

```

## Bináris fák datatype deklarációval (folyt.)

- Másféle fastruktúrákat is deklarálhatunk, pl.
  - kezdhetjük az 'a típusú értékkel, majd folytathatjuk előbb a bal, azután a jobb részfa megadásával,
  - felhasználhatjuk a levelet is értékek tárolására, az értéket nem tároló üres csomkokat pedig E-vel jelölhetjük.
- A leírtak szerinti bináris fát hoz létre a következő deklaráció:  
`datatype 'a tree = E | L of 'a | B of 'a * 'a tree * 'a tree;`
- A rekurzív függvényekhez hasonlóan a rekurzív adattípusok deklarációjában is kell lennie nemrekurzív ágnek (ún. triviális esetnek).
- A nemrekurzív ág hiánya miatt az alábbi, szintaktikailag helyes deklarációk használhatatlanok:  
  
`datatype 'a badtree = B of 'a badtree * 'a * 'a badtree;`  
`datatype 'a badtree = L of 'a badtree | B of 'a badtree * 'a * 'a badtree;`

## Egyszerű műveletek bináris fákön

- nodes egy fa csomópontjait számlálja meg. Legyen  
datatype 'a tree = L | N of 'a \* 'a tree \* 'a tree;  
(\* nodes f = az f fa csomópontjainak a száma  
nodes : 'a tree -> int \*)  
fun nodes (N(\_, t1, t2)) = 1 + nodes t2 + nodes t1  
| nodes L = 0;
- nodes akkumulátort használó változata (nodesa):

```
fun nodesa f =
 let (* nodes0(f, n) = n + a csomópontok száma f-ben
 nodes0 : 'a tree * int -> int *)
 fun nodes0 (N(_, t1, t2), n) = nodes0(t1, nodes0(t2, n+1))
 | nodes0 (L, n) = n
 in nodes0(f, 0)
 end;
```

## Egyszerű műveletek bináris fákön (folyt.)

- A fa gyökeréből a leveléhez vezető úton az élek számát (az út hosszát) az adott levél szintjének is nevezzük. A szintek közül a legnagyobbat a fa *mélységének* hívjuk.
- depth egy fa mélységét határozza meg.
 

```
(* depth f = az f fa mélysége
 depth : 'a tree -> int *)
fun depth (N(_, t1, t2)) = 1 + Int.max(depth t2, depth t1)
 | depth L = 0;
```
- depth akkumulátort használó változata (deptha):
 

```
fun deptha f = let fun depth0 (N(_, t1, t2), d) =
 Int.max(depth0(t1, d+1), depth0(t2, d+1))
 | depth0 (L, d) = d
 in
 depth0(f, 0)
 end;
```

## SML-Prolog átvezetés: párhuzamok a két nyelv között

### SML

```
fun append ([], ys) = ys
 | append (x::xs, ys) =
 x::append (xs, ys)
```

### SML „Prologosító”

```
fun append([], L) = L
 | append(X::L1, L2) =
 let val L3 = append(L1, L2)
 in X::L3 end
```

### függvény

#### klóz

változó: egyetlen, ismert érték  
 egyirányú mintaillesztés  
 egyértelmű klózzválasztás  
 egy eredmény  
 egyirányú használat

### adatkonstruktor-függvény

egymásba ágyazott függvényhívások

### Prolog

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

### Prolog „SML-esítő”

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

### predikátum

klóz (lazább a kapcsolat a pred.-mal)  
 változó: egy, esetleg ismeretlen érték  
 kétirányú mintaillesztés  
 többértelmű klózzválasztás  
 több eredmény (nemdeterminizmus)  
 többirányú használat  
 (pl. összerakó és szétszedő append)  
 struktúra (rekord)  
 konjunkció, segéd-változóval

## SML-Prolog átvzetés: további példák

---

*SML*

```
fun append xs ys = foldr op:: ys xs
```

*Prolog*

```
/* Prologban kevésbé használtak
a magasabbrendű eljárások */
```

```
fun fakt 0 = 1
 | fakt n = n * fakt (n-1)
```

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

típusos nyelv

típusatlan nyelv

magasabbrendű függvény

rekurzió, ritkábban magasabbr. pred.

rekurzió

visszalépéses ciklus

kivétel

(pl. két lista közös eleme)  
meghiúsulás, kivétel

(pl. fakt negatív számmal)

## Összefoglalás: Prolog programok szemantikája

---

- Prolog program jelentése = milyen válaszokat (behelyettesítéseket) kapunk egy cél futtatásakor:
- Procedurális szemantika — az ismertett végrehajtási, egyesítési algoritmus.
- Deklaratív szemantika:
  - program: logikai állítások (klózok, azaz implikációk) halmaza;
  - egy cél futási eredménye: olyan behelyettesítés, amelyre a cél *következménye* a programnak.
- A Prolog procedurális szemantika csak olyan választ ad, amely a deklaratív szemantika szerint is helyes! (Ha predikátumaink „igazak”, akkor rossz eredményt nem kaphatunk, csak végtelen ciklust. :- ( )



## ismétlés: A Prolog végrehajtási mechanizmus, dióhéjban

---

- (Kezdet:) Ha célsorozat üres  $\rightarrow$  sikeres lefutás.
- (Folytatás:) Keresünk az *első* céllal egyesíthető klózfejet (a klózból friss másolatot képezve, felülről lefelé haladva a programbeli klózokon ).
- Ha van ilyen:
  - Ha van esély további illesztésre, akkor választási pontot hozunk létre: a futás jelenlegi állapotát (célsorozat + hányadik klózzal illesztettünk) megjegyezzük, azaz a veremre rakjuk.
  - Az egyesítéshez szükséges behelyettesítéseket a klóztörzsön és a célsorozaton is elvégezzük.
  - Az első cél helyébe a klóztörzset rakjuk, ez lesz az új célsorozat, majd vissza a (Kezdet)-hez.
- Ha nincs illeszthető klózfej, akkor visszalépünk a *legutolsó* választási pontnak megfelelő állapotba (azt leemelve a verem tetejéről), és új egyesíthető fejű klóz keresésével folytatjuk a (Folytatás)-nál.

## ismétlés: A Prolog egyesítési algoritmus, dióhéjban

---

- Legáltalánosabb egyesítő behelyettesítés meghatározása
- Azonos változók ill. konstansok behelyettesítés nélkül egyesíthetők.
- Változó minden más kifejezéssel egyesíthető, triviális behelyettesítéssel (tartalmazás-vizsgálat nélkül)
- Két összetett kifejezés egyesíthető, ha funktoraik azonosak, és az argumentumaik sorra egyesíthetőek, úgy, hogy a megelőző argumentumok egyesítéséhez szükséges behelyettesítéseket már elvégeztük. Az argumentumok egyesítését biztosító behelyettesítések kompozíciója a legáltalánosabb egyesítő.
- Minden más esetben a két kifejezés nem egyesíthető, az egyesítési algoritmus meghiúsul.

## 4. fejezet: Prolog programozási módszerek

---

- Az előző előadás-blokk (jegyzetbeli 3. fejezet) célja volt:
  - a Prolog nyelv alapjainak bemutatása,
  - a logikailag „tiszta” résznyelvre koncentrálni.
- A jelen előadás-blokk (jegyzetben a 4. fejezet) célja: olyan
  - beépített eljárások,
  - programozási technikákbemutatása, amelyekkel
  - hatékony Prolog programok készíthetők,
  - esetleg a tiszta logikán túlmutató eszközök alkalmazásával.

## Prolog programozási módszerek: tartalomjegyzék

---

- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Algoritmusok Prologban
- Megoldások gyűjtése és felsorolása
- Megoldásgyűjtő eljárások
- Meta-logikai eljárások
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezeelés
- „Hagyományos” beépített eljárások
- Nyelvtani elemzés

# A KERESÉSI TÉR SZŰKÍTÉSE



## Prolog nyelvi eszközök a keresési tér szűkítésére

---

### Eszközök

- a vágó beépített eljárás: !
- feltételes diszjunktív szerkezet:  
( felt -> akkor ; egyébként )

Miért vágunk le ágakat a keresési térben?

- mert mi tudjuk, hogy ott nincs megoldás, de a Prolog megvalósítás nem — zöld vágás, szemantikailag „ártalmatlan”
- (Például, a legtöbb Prolog megvalósítás „nem tudja”, hogy a  $X = 0$  és  $X > 0$  feltételek kizárják egymást, lásd később: indexelés.)
- ténylegesen eldobunk megoldásokat — vörös vágás, a program jelentését megváltoztatja
- (Vörös vágás legtöbbször úgy keletkezik, hogy egy zöld vágót tartalmazó programban a „felesleges” feltételeket elhagyjuk.)

## Példák a vágó eljárás használatára

```

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

last(+L, ?E): az L lista utolsó eleme E. (A lists könyvtárban van!)
ast([E], E) :- !.
 % zöld vágó
ast([_|L], E) :- last(L, E).

pirosak(+L, -Pirosak): Pirosak az L piros/1 funktorú elemeiből áll.
pirosak([], []).
pirosak([E|Ek], [E|Pk]) :-
 E = piros(_), !,
 pirosak(Ek, Pk).
 % vörös vágó
pirosak([_E|Ek], Pk) :-
 /* \+ _E = piros(_), */ pirosak(Ek, Pk).

```

**Figyelem:** a fenti példák nem tökéletesek, hatékonyabb ill. általánosabb átaltoztatukat később ismertetjük!

## A vágó definíciója

---

### Végédfogalom

- Egy cél *szülője* az a cél, amelyik az őt tartalmazó klóz fejével illesztődött.
- Pl. a  $\text{last}([E], E) :- !.$  klózbeli vágó szülője lehet a  $\text{last}([7], X)$  hívás.
- A  $g$  (ancestors) nyomkövetési parancs kiírja a kurrens cél őseit (szülőjét, annak szülőjét, stb.)

### A vágó végrehajtása:

- mindig sikerül; és a végrehajtás adott állapotától visszafelé egészen a szülő célig, azt is beleértve, minden választási pontot megszüntet.

### A vágás kétféle választási pontot szüntet meg:

```
s(X) :- s(X), !. % az s(X)-beli választási pontokat --- a vágót megelőző
 % cél(ok)nak az első megoldására szorítkozunk
 % az r(X) többi klózának választását --- a vágót tartalmazó
s(X) :- t(X). % klóz mellett kötelezzük el magunkat (commit)
```



## A vágó által megszüntetett választási pontok

o vágó nélküli példa

l(X) :- s(X).

l(X) :- t(X).

o ugyanaz a példa vágóval

l(X) :- s(X), !.

l(X) :- t(X).

(a).            s(b).            t(c).

o a vágó nélküli példa futása

- q(X), write(X), fail.

-->

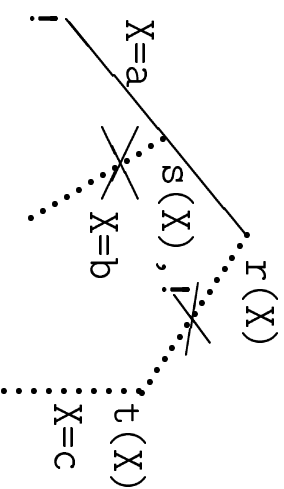
abc

o a vágót tartalmazó példa futása

- r(X), write(X), fail.

-->

a



## A diszjunktív feltételes szerkezet definíciója

- A diszjunktív feltételes szerkezet, a diszjunktíóhoz hasonlóan egy segédeljárással váltható ki:

|                                                                                                                                       |                                                                                                                                                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> p :-     ...     (   felt1 -&gt; akkor1     ;   felt2 -&gt; akkor2     ;   ...     ;   egyébként     )     ... .         </pre> | $\Rightarrow$ <pre> p :-     ...     segéd(...)     ... .     segéd(...) :- felt1, !, akkor1.     segéd(...) :- felt2, !, akkor2.     ... .     segéd(...) :- egyébként.         </pre> |
|---------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- Az egyébként alternatíva elmaradhat, ilyenkor a megfelelő klóz is elmarad.
- SICStus módban a felt részekben vágó nem lehet, ISO módban lehet, de hatásköre (szülője) a felt rész.
- Az akkor részekben lehet vágó. Ennek hatásköre, a  $\rightarrow$  nyílból generált vágóval ellentétben, a teljes p predikátum (távolbaható vágó).

## Példák a diszjunktív feltételes szerkezet használatára

```

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

last(+L, ?E): az L nem üres lista utolsó eleme E.
ast([E|L], Last) :-
 (L = [] -> Last = E
 ; last(L, Last)
).

pirosak(+L, ?Pirosak): Pirosak az L piros/1 funktorú elemeiből áll.
irosak([], []).
irosak([E|Ek], Pk) :-
 (E = piros(_) -> Pk = [E|Pk0]
 ; Pk = Pk0
), pirosak(Ek, Pk0).

```

## A vágás első alapesete — klóz mellett való elkötelezés

---

- A klóz melletti elkötelezés általában egyszerű feltételes szerkezetet jelent.  
szülő :- feltétel, !, akkor.  
szülő :- egyébként.
- A vágó szükségtelenné teszi a feltétel negációjának végrehajtását a többi klózban. A logikailag tiszta, de nem hatékony alak:  
szülő :- feltétel, akkor.  
szülő :- \+ feltétel, egyébként.  
A fenti két alak csak akkor ekvivalens, ha feltétel egyszerű, nincs benne választás.
- Analógia: ha  $a$ ,  $b$  és  $c$  logikai változók (pl. Pascalban), akkor  
 $\text{if } a \text{ then } b \text{ else } c \equiv a \wedge b \vee \neg a \wedge c$
- A vágó által kiváltott negált feltételt célszerű kommentként jelezni:  
szülő :- feltétel, !, akkor.  
szülő :- / \* \+ feltétel, \*/ egyébként.

## Feltételes szerkezetek

### Feltételes szerkezet — példa

```
% abs(X, A): A az X abszolút értéke.
abs(X, A) :- X < 0, !, A is -X.
abs(X, X) /* :- X >= 0 */.
```

### Általános alak

```
p :- felt1, !, akkor1.
p :- felt2, !, akkor2.
...
p :- egyébként.
```

### Diszjunktív feltételes szerkezet

```
abs(X, Y) :-
 (X < 0 -> Y is -X
 ; Y = X
).
```

### Általános alak

```
p :-
 (felt1 -> akkor1
 ; felt2 -> akkor2
 ; ...
 ; egyébként
).
```

## Feltételes szerkezetek és fejillesztés

- Vigyázat: a tényleges feltétel részét képezik a fejbeli egyesítések!

```
% a vágót fej-egyesítés előzi meg % az egyesítés explicitte téve:
abs(X, X) :- X >= 0, !.
abs(X, A) :- A is -X.
abs(X, A) :- A is -X.
abs(X, A) :- A is -X.
```

- A fej-egyesítés gondot okozhat, ha az eljárást ellenőrzésre használjuk:  
| ?- abs(10, -10). ----> yes

- A megoldás a *vágás alapszabálya*:

- A kimenő paraméterek értékadását mindig a vágó után végezzük!

```
abs(X, A) :- X >= 0, !, A = X.
abs(X, A) :- A is -X.
```

- Ez nemcsak általánosabban használható, hanem hatékonyabb kódot is ad!  
• („*kimenő*” *paraméterek* — vágó alkalmazásakor általában nincs többirányú használat :-)

## A bevezető példának a vágás alapszabályát betartó változata

```

fakt(+N, ?F): N! = F.
akt(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.
ast([E], Last) :- !, Last = E.
ast([_|L], E) :- last(L, E).

pirosak(+L, ?Pirosak): Pirosak az L piros/1 funktorú elemeiből áll.
irosak([], []).
irosak([E|Ek], Pk) :-
 E = piros(_), !, Pk = [E|Pk0], pirosak(Ek, Pk0).
irosak([_|Ek], Pk) :-
 /* \+ _E = piros(_), */ pirosak(Ek, Pk).

```

Megjegyzés: a diszjunktív alakban a feltételek eleve explicitek, nincs ejillesztési probléma.

**Példasor:**  $\max(X, Y, Z) : X \text{ és } Y \text{ maximuma } Z.$

---

- 1. változat, tiszta Prolog. Lassú, választási pontot hagy.

$\max(X, Y, X) :- X \geq Y.$

$\max(X, Y, Y) :- Y > X.$

- 2. változat, zöld vágóval. Lassú, nem hagy választási pontot.

$\max(X, Y, X) :- X \geq Y, !.$

$\max(X, Y, Y) :- Y > X.$

- 3. változat, vörös vágóval. Gyors, nem hagy választási pontot, de nem használható ellenőrzésre, pl.  $| ?- \max(10, 1, 1)$  sikerül.

$\max(X, Y, X) :- X \geq Y, !.$

$\max(X, Y, Y).$

- 4. változat, vörös vágóval. Helyes, gyors és nem hagy választási pontot.

$\max(X, Y, Z) :- X \geq Y, !, Z = X.$

$\max(X, Y, Y) /* :- Y > X */.$



## A vágás második alapesete — első megoldásra való megszorítás

---

Mikor használjuk az első megoldásra megszorító vágót?

- behelyettesítést nem okozó, eldöntendő kérdés esetén;
- feladat-specifikus optimalizálásra;
- végtelen választási pontot létrehozó eljárások hasznosítására.

Eldöntendő kérdés: eljáráshívás csupa bemenő paraméterrel

```
van_elég_hosszú_út(+N, +A, +B, +Min):
```

*van*, *A* és *B* között van *N* lépéses út, amelynek összhossza legalább *Min* km.

```
van_elég_hosszú_út(N, A, B, Min) :-
```

```
 út_vonal(N, A, B, Hossz), Hossz >= Min, !.
```

Eldöntendő kérdés esetén általában nincs értelme többszörös választ adni/várni.

## Adatszpecifikus optimalizálás

- Példafeladat: egy nem-negatív számokból álló lista első pozitív eleme
  - Első megoldás, rekurzióval (mérnöki :-)
- ```

első_poz_elem([0|L], EP) :- !, első_poz_elem(L, EP).
első_poz_elem([EP|_], EP) /* :- EP > 0 */.

```
- Második megoldás, visszalépéses kereséssel (matematikusai :-)
- ```

első_poz_elem(L, EP) :-
 append(Nk, [EP|_], L), EP > 0, \+ poz_elem(Nk, _).

poz_elem(L, P) :- member(P, L), P > 0.

```
- Harmadik megoldás, vágóval (Prolog hacker megoldása :-)
- ```

első_poz_elem(L, EP) :- member(EP, L), EP > 0, !.

```
- Figyelem: a harmadik megoldás épít a member/2 felsorolási sorrendjére!

Adatszpecifikus optimalizálás — 2. példa

A feladat: megkeresendő egy lista elején álló „plató” hossza
 platónak hívjuk a csupa azonos elemből álló folytonos részlistát).

Az L lista első eleme H-szor ismétlődik a lista kezdőszelleteként.

```
kezdethossz(L, H) :-
    L = [E|_], append(Ek, Farok, L),
    \+ Farok = [E|_], !,           % vörös vágó
    /* egyformák(Ek, E), */
    length(Ek, H).
```

*

egyformák(Ek, E): Az Ek lista minden eleme E.

```
egyformák([], _).
```

```
egyformák([E|Ek], E) :-
```

```
    egyformák(Ek, E).
```

```
:/
```

```
?- kezdethossz([1,1,1,2,3,5], H).
```

```
H = 3 ? ; no
```

Végtelen választás megszelídítése: `memberchk`

`memberchk(X, L):` "X eleme az L listának" kérdés első megoldása.

```
% 1. változat
memberchk(X, L):-
    member(X, L), !.

% 2. ekvivalens változat
memberchk(X, [_|_]) :- !.
memberchk(X, [_|L]) :-
    memberchk(X, L).
```

`memberchk/2` használata

- Eldöntő kérdésben (visszalépéskor nem keresi végig a lista maradékát.)
`memberchk(1, [1,2,3,4,5,6,7,8,9])`
- Nyílt végű lista elemévé tesz, pl.:
| ?- `memberchk(1,L), memberchk(2,L), memberchk(1,L).`
 `L = [1,2|_A] ?`

Nyílt végű listák kezelése memberchk segítségével: szótárprogram

```
szótaraz(Sz) :-
    read(M-A), !, % kifejezést olvas be és egyesíti az argumentummal
    memberchk(M-A, Sz),
    write(M-A), nl,
    szótaraz(Sz).
szótaraz(_).
```

Egy futása:

```
?- szótaraz(Sz).
   : alma-apple.
   alma-apple
   : korte-pear.
   korte-pear
   : vege.
Sz = [alma-apple,korte-pear|_A] ?
```

VEZÉRLÉSI ELJÁRÁSOK



Vezérlési eljárások: *call/1*

- Vezérlési eljárás: A Prolog végrehajtási mechanizmusához kapcsolódó beépített eljárás (pl. a vágó).
- A vezérlési eljárások többsége *magasabbrendű* eljárás, azaz olyan eljárás, amely egy vagy több argumentumát eljáráshívásként értelmezi. (A magasabbrendű Prolog eljárásokat szokás *meta-eljárásnak* is hívni.)
- A meta-eljárások fő képviselője és alapvető építőeleme a `call/1`:
 - Hívási minta: `call(+Cél)`
 - Argumentumok: `Cél` egy „meghívható kifejezés” (callable), azaz struktúra, vagy atom.
 - Jelentése: `Cél igaz`.
 - Hatása: a `Cél` kifejezést eljáráshívássá alakítja és meghívja
- Ha klóztörzsben célként szerepel egy `X` változó, akkor azt a rendszer egy `call(X)` hívássá alakítja át.
- Cél-ban további vezérlési eljárások is előfordulhatnak, pl. `'/2`, `','/2`, `','/2`. A Cél-beli vágó csak a `call` belsejében vág (szülője a `call(Cél)` hívás).

call/1: példák

```
?- [user].
kétszer(Hívás) :- call(Hívás), Hívás.
~D
consulted user in module user, 0 msec 224 bytes}
es
?- kétszer(write(ba)), nl.
aba
es
?- _Cél = (kétszer(write(ba)), write(' ')), kétszer(_Cél), nl.
aba baba
es
?- listing(kétszer).
kétszer(A) :-
    call(user:A),
    call(user:A).
es
```


call/1 példa: futási időt mérő meta-eljárás

Kiírja Goal első megoldásának előállításához vagy a meghiúsuláshoz

szükséges időt, a Txt szöveg kíséretében (lásd: példak/call_koltsege.pl).

ime(Txt, Goal) :-

```
    statistics(runtime, [T0,_]), % T0 az indítás óta eltelt CPU idő,
                                % msec-ban (szemétyűjtés nélkül).
```

```
    ( call(Goal) -> Res = true
```

```
    ; Res = false
```

```
),
```

```
statistics(runtime, [T1,_]), T is T1-T0,
```

```
format('~w futási idő = ~3d sec\n', [Txt,T]),
```

```
Res = true.
```

A call/1 viszonylag költséges: egy 1414 hosszú lista megfordítása nrev-vel (kb millió append hívás), minden append körül egy felesleges call-lal ill. anélkül:

	call nélkül	call-lal
lefordítva	0.220 sec	9.710 sec
interpretálva	1.850 sec	11.940 sec

Vezérlési szerkezetek mint eljárások

- A `call/1` argumentumában azért szerepelhetnek vezérlési szerkezetek, mert ezek maguk beépített eljárásként is jelen vannak a Prolog rendszerben:
 - `(', ')/2`: konjunkció.
 - `(;)/2`: diszjunkció.
 - `(->)/2`: if-then.
 - `(;)/2`: if-then-else.
- A `call`-ban szereplő vezérlési szerkezetek lényegében ugyanúgy futnak, mint az interpretált (`consult`-tal betöltött) kód.

- **Példa**

```
| ?- ketszer((member(X, [a,b,c,d]), write(X), fail ; nl)).
abcd
abcd
true ?
```

További vezérlési eljárások

- A „nem-bizonyíthatóságot” ellenőrző $\backslash +$ eljárás definíciója:
 $\backslash + X :- \text{call}(X), !, \text{fail}.$
 $\backslash + _X.$
- Az első megoldásra megszorító once beépített eljárás definíciója:
 $\text{once}(X) :- \text{call}(X), !.$
- A true ill. fail beépített eljárás mindig sikerül ill. mindig meghiúsul.
- A repeat beépített eljárás egy végtelen választási pontot hoz létre:
 $\text{repeat}.$
 $\text{repeat} :- \text{repeat}.$
- A repeat eljárást mindig egy vágóval kell semlegesíteni. Példa:
 $\text{bc} :- \text{repeat}, \text{read}(\text{Expr}),$
 $(\quad \text{Expr} = \text{end_of_file} \rightarrow \text{true}$
 $; \quad \text{Res is Expr}, \text{write}(\text{Expr} = \text{Res}), \text{nl}, \text{fail}$
 $), !.$

DETERMINIZMUS ÉS INDEXELÉS



Determinizmus

- Egy eljáráshívás *determinisztikus*, ha (legfeljebb) egyféleképpen sikerülhet.
- Egy eljáráshívásnak egy sikeres végrehajtása *determinisztikusan futott le*:
 - ha nem hagyott választási pontot a híváshoz tartozó részében:
 - választásmentesen futott le, azaz létre sem hozott választási pontot (figyelem: ez a Prolog megvalósítástól függ!); vagy
 - létrehozott ugyan választást, de megszüntette (kimerítette, levágta).

- A SICStus Prolog nyomkövetésében ? jelzi a *nem*determinisztikus lefutást:

p(1, a).	?- p(1, X).		% det. hívás,
p(2, b).		1 1 Exit: p(1,a)	% det. lefutás
p(3, b).	?- 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

- Mi a determinisztikus lefutás haszna?
 - a futás gyorsabb lesz,
 - a tárigény csökken,
 - más optimalizálások (pl. jobbrekurzió) alkalmazható.
- Hogyan ismeri fel a fordító azt, hogy nem kell választási pont?
 - indexelés (indexing)
 - vágók és feltételes szerkezetek

- Az alábbi definíciók esetén a $p(Novar, Y)$ hívás nem hoz létre választási pontot (a 2. definíció esetén a $p(Var, Y)$ sem):

$p(1, a).$	$p(1, Y) :- !,$	$p(X, Y) :-$
$p(2, b).$	$Y = a.$	$(X > 1 \rightarrow Y = a$
	$p(-, b).$	$); Y = b$
		$).$

indexelés

- Mi az indexelés?
- egy 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:
 - Fordításkor a funktorokhoz elkészítjük az illeszthető klózok részahalmazát.
 - Futáskor lényegében konstans idő alatt választunk a részahalmazok közül.
 - *Fontos:* ha egyelemű a részahalmaz, nem hozunk létre választási pontot!

Példa indexelésre

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

• A $p(A, B)$ hívással illesztendő klózhalmaz:

- $\{(1) (2) (3) (4) (5)\}$ ha A változó;
- $\{(1) (2)\}$ ha $A = 0$;
- $\{(2) (3) (4)\}$ ha A fő funktora $s/1$;
- $\{(2) (5)\}$ ha $A = 9$;
- $\{(2)\}$ minden más esetben.

• Példák hívásokra:

- $p(3, Y)$ 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

- Azonos funktorú struktúrák az első fejargumentumban:

- Ha a klózok szétválasztásához szükség van az első (struktúra) argumentum részeire is, akkor érdemes segédeljárást bevezetni.

- Például $p/2$ és $q/2$ ekvivalens, de $q(Nonar, Y)$ determinisztikusan fut le!

$p(0, a).$	$q(0, a).$	$q_seged(0, b).$
$p(s(0), b).$	$q(s(X), Y) :-$	$q_seged(1, c).$
$p(s(1), c).$	$q_seged(X, Y).$	
$p(9, z).$	$q(9, z).$	

- Fejillesztés kiváltása egyenlőséggel (vö. SML rétegelt minta)

- Az indexelés figyelembe veszi a törzs elején szereplő egyenlőséget:
 $p(X, \dots) :- X = Kif, \dots$ 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.
% rhossz([X|L], RH) :- length([X|L], H), RH is 1/H. % nem hatékony!
```

indexelés és aritmetika, indexelés és listák

- Aritmetikai elágazások
- Az indexelés nem foglalkozik aritmetikai vizsgálatokkal.
- Pl. az $N = 0$ és $N > 0$ feltételek nem „zárják ki” egymást.
- Az alábbi $\text{fakt}(N, F)$ lefutása nem-determinisztikus:

$$\text{fakt}(0, 1).$$

$$\text{fakt}(N, F) :- N > 0, N1 \text{ is } N-1, \text{fakt}(N1, F1), F \text{ is } N * F1.$$

- Listakezelő eljárások

- 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 (összefűzésre).
`append([], L, L).`
`append([X|L1], L2, [X|L3]) :-`
`append(L1, L2, L3).`
- A `last/2` közvetlen megfogalmazása nondeterminisztikusan 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!

$p(1, Y) :- !, Y = 2.$	$\% (1)$	$ $	$q(1, 2) :- !.$	$\% (1)$
$p(X, X).$	$\% (2)$	$ $	$q(X, X).$	$\% (2)$
$Arg1=1 \rightarrow (1), Arg1 \neq 1 \rightarrow (2)$		$ $	$Arg1=1 \rightarrow \{(1), (2)\}, Arg1 \neq 1 \rightarrow (2)$	

- Csak akkor tudjuk fordításakor kizárni a vágót követő klózokat, ha garantált, hogy az adott fő funktor esetén a vágót elérjük. Ennek feltételei:

- az első argumentumban konstans, vagy legaltalanosabb struktúra legyen,
- a további argumentumok változók legyenek,
- a fejben az összes változóelőfordulás különböző legyen,
- a törzs első hívása a vágó (megengedve a fejillesztést kiváltó $=-t$).

- Példa: $p(s(A, B, C), D, E) :- !, \dots$

- Ez egy újabb érv a vágás alapszabálya mellett:

A kimenő paraméterek értékadását mindig a vágó után végezzük!

A vágó és az indexelés hatékonysága

- Egy Fibonacci-szerű sorozat: $f_1 = 1$; $f_2 = 2$; $f_n = f_{\lfloor 3n/4 \rfloor} + f_{\lfloor 2n/3 \rfloor}$, $n > 2$

% determinisztikus	% determ. leftutású	% választásmentes
fib(1, 1).	fibc(1, 1) :- !.	fibci(1, F) :- !, F = 1.
fib(2, 2).	fibc(2, 2) :- !.	fibci(2, F) :- !, F = 2.
fib(N, F) :-	fibc(N, F) :-	fibci(N, F) :-
N > 2,	N > 2,	N > 2,
N2 is N*3//4,	N2 is N*3//4,	N2 is N*3//4,
N3 is N*2//3,	N3 is N*2//3,	N3 is N*2//3,
fib(N2, F2),	fibc(N2, F2),	fibci(N2, F2),
fib(N3, F3),	fibc(N3, F3),	fibci(N3, F3),
F is F2+F3.	F is F2+F3.	F is F2+F3.

- Futási idők $N = 2000$ esetén

	fib	fibc	fibci
futási idő	4410 ms	4060 ms	3820 ms
meghívásulási idő	730 ms	0 ms	0 ms
összesen	5140 ms	4060 ms	3820 ms

Választás-mentesség diszjunktív 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 (ld. később, pl. @<, @=<, ==).
- Analóg módon 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:

<pre>fakt(N, F) :- (N == 0 -> % N = 0 nem jó! F = 1 ; N1 is N-1, fakt(N1, F1), F is N*F1)</pre>	<pre>fakt(N, F) :- N == 0, !, F = 1. fakt(N, F) :- N1 is N-1, fakt(N1, F1), F is N*F1.</pre>
--	--

JOBPREKURZIO ÉS AKKUMULÁTOROK

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 tagjának utolsó helyén, stb., és
 - a rekurzív hívás pillanatában nincs választási pont a predikátumban, tehát a rekurzív hívást megelőző célok determinisztikusan futottak le.
- Jobbrekurzió optimalizálás: az utolsó hívás végrehajtása *előtt* a predikátum által lefoglalt hely felszabadul ill. szemégyűjtésre alkalmassá válik.
- Ez az optimalizálás nemcsak rekurzív hívás esetén, hanem minden *utolsó* hívás esetén megvalósul — utolsó hívás optimalizálás (last call optimisation).
- A jobbrekurzió így tehát nem növeli a memória-igényt, korlátlan mélységig futthat — 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:

```
% sum(+L, ?S): Az L számlista elemeinek összege S.  
sum([], 0).  
sum([X|L], S):- sum(L,S0), S is S0+X.
```

- Első jobbrekurzív változat, csak ellenőrzésre használható:

```
% sum1(+L, +S): Az L számlista elemeinek összege S.  
sum1([], 0).  
sum1([X|L], S) :- /* S is S0+X helyett: */ S0 is S-X, sum1(L, S0).
```

- Második jobbrekurzív változat, csak kiírni tudja az eredményt:

```
% sum2(+L): Az L számlista elemeinek összegét kiírja.  
sum2(L):- sum2(L, 0).
```

```
% sum2(+L, +S0): Az L lista S0-lal növelt összegét kiírja.  
sum2([], S) :- write(S), nl.  
sum2([X|L], S0):- S1 is S0+X, sum2(L, S1).
```

Jobbrekurzív listaösszeg — akkumulátorpár segítségével

- Harmadik változat: teljes értékű jobbrekurzív lista-összegző:

```
% sum3(+L, ?S) : Az L számlista elemeinek összege S.
```

```
sum3(L, S) :- sum3(L, 0, S).
```

```
% sum3(+L, +S0, ?S) : Az L lista elemeit hozzáadva S0-hoz kapjuk S-et.
```

```
sum3([], S, S).
```

```
sum3([X|L], S0, S) :-
```

```
    S1 is S0+X, sum3(L, S1, S).
```

- Az *akkumulátor* fogalma:

- A `sum3(L, S0, S)` predikátumban az `S0` és `S` argumentumok egy akkumulátorpárt alkotnak.

- Az akkumulátorpár két része egy változó mennyiség (a példában az összeg) különböző időpontokban vett értékeit mutatja:

- `S0` az összeg értéke a `sum3/3 meghívásakor`;

- `S` az összeg értéke a `sum3/3 lefutása után`.

Akkumulátorok használata

- Az akkumulátorpárok a hagyományos, „változtatható” változók megfelelői.

- Az általános séma:

$p(\dots, A_0, A)$:-

$q_0(\dots, A_0, A_1), \dots,$

$q_1(\dots, A_1, A_2), \dots,$

$q_n(\dots, A_n, A).$

- A `sum3/3` második klóza ilyen alakra hozva:

`sum3([X|L], S0, S) :- plus(X, S0, S1), sum3(L, S1, S).`

`plus(X, S0, S) :- S is S0+X.`

- Akkumulátorváltozók elnevezési konvenciója: kezdőérték: *Válto*; közbűlső értékek: *Válto1*, ..., *Váltn*; végérték: *Vált*.

- A Prolog akkumulátorpár nem más mint a funkcionális programozásból ismert gyűjtőargumentum és a függvény eredményének együttese.

Akkumulátorok használata — folytatás

Három lista összege

```
% sum_3_lists(+L, +LL, +LLL, +S0, ?S): Az L, LL, LLL számlisták
% összegeinek összege S-S0
sum_3_lists(L, LL, LLL, S0, S) :-
    sum3(L, S0, S1), sum3(LL, S1, S2), sum3(LLL, S2, S).
```

Többszörös akkumulálás — listák összege és négyzetösszege

```
% sum12(+L, +S0, ?S, +Q0, ?Q): S-S0 =  $\sum L_i$ , Q-Q0 =  $\sum L_i * L_i$ 
sum12([], S, S, Q).
sum12([X|L], S0, S, Q0, Q) :-
    S1 is S0+X, Q1 is Q0+X*X, sum12(L, S1, S, Q1, Q).
```

Többszörös akkumulátorok összevonása

```
% sum12(+L, +S0/Q0, ?S/Q): S-S0 =  $\sum L_i$ , Q-Q0 =  $\sum L_i * L_i$ 
sum12([], SQ, SQ).
sum12([X|L], S0/Q0, SQ) :-
    S1 is S0+X, Q1 is Q0+X*X, sum12(L, S1/Q1, SQ).
```

Korábbi listakezelő predikátumok

- A revapp mint akkumuláló eljárás

```
% revapp(Xs, L0, L): Xs megfordítását L0 elé fűzve kapjuk L-t.  
% Másképpen: Xs megfordítása L-L0.  
revapp([], L, L).  
revapp([X|Xs], L0, L) :-  
    L1 = [X|L0], revapp(Xs, L1, L).
```
- Az L-L0 jelölés (különbséglista): az a lista amelyet úgy kapunk, hogy L végéről elhagyjuk L0-t (előfeltétel: L0 szuffixuma L-nek).
- Az append is tekinthető akkumuláló eljárásnak (a 2. és 3. arg. felcserélt).
A változtatás: az L0 elejéről sorra elhagyjuk Xs elemeit, végül marad L.

```
% append(Xs, L, L0): L0 elejéről Xs elemeit lehagyva marad L.  
% Másképpen: Xs = L0-L.  
append([], L, L).  
append([X|Xs], L, L0) :-  
    L0 = [X|L1], append(Xs, L, L1).
```

Egy mintafeladat: $a^n b^n$ alakú sorozat előállítás

● Első megoldás, $3n$ lépés

```
anbn(N, L): Az L lista N db a-ból
és azt követő N db b-ből áll.
anbn(N, L) :-
    an(N, a, AN),
    an(N, b, BN),
    append(AN, BN, L).
```

● Második megoldás, $2n$ lépés

```
anbn(N, L) :-
    an(N, b, [], BN),
    an(N, a, BN, L).
```

```
an(N, A, L): L az A elemet N-szer
tartalmazó lista
an(0, _A, L) :- !, L = [].
an(N, A, [A|L]) :-
    N > 0,
    N1 is N-1,
    an(N1, A, L).
```

```
% an(N, A, L0, L): L-L0 az A
% elemet N-szer tartalmazó lista
an(0, _A, L0, L) :- !, L = L0.
an(N, A, L0, [A|L]) :-
    N > 0,
    N1 is N-1,
    an(N1, A, L0, L).
```

$a^n b^n$ alakú sorozatok (folyt.)

Harmadik megoldás, n lépés

```

nbn(N, L) :-
    anbn(N, [], L).

anbn(N, L0, L): Az L-L0 lista N db a-ból és azt követő N db b-ből áll.
nbn(0, L0, L) :- !, L = L0.
nbn(N, L0, [a|L]) :-
    N > 0,
    N1 is N-1,
    anbn(N1, [b|L0], L).

```

A második klóz nem jobbrekurzív változata

```

nbn(N, L0, L) :-
    N > 0, N1 is N-1,
    L1 = [b|L0],      % 1. lépés: L0 elé b => L1
    anbn(N1, L1, L2), % 2. lépés: L1 elé a~N1 b~N1 => L2
    L = [a|L2].       % 3. lépés: L2 elé a => L

```

Összetettebb adatstruktúrák akkumulálása

- Az adatstruktúra:
% :- type bfa --> ures ; bfa(int, bfa).
- A fa csomópontjaiban tároljuk a számértékeket, a levelek nem tárolnak információt.
- Egészek gyűjtése rendezett bináris fában
 - beszur(BFa0, E, BFa): Az E egész számnak a BFa0 fába való beszúrása a BFa bináris fát eredményezi.
 - Itt BFa0 és BFa egy akkumulátor-pár, de az indexelés érdekében BFa0 az első argumentum-pozícióba kerül.

- Példafutás:

| ?- beszur(ures, 3, Fa0), beszur(Fa0, 1, Fa1), beszur(Fa1, 5, Fa2).

Fa0 = bfa(3,ures,ures),

Fa1 = bfa(3,bfa(1,ures,ures),ures),

Fa2 = bfa(3,bfa(1,ures,ures),bfa(5,ures,ures)) ?

Akkumulálás bináris fákkal

Elem beszúrása bináris fába

```

, beszur(BF0, E, BF): E beszúrása BF0 rendezett fába
és a BF rendezett fát adja
, :- pred beszur(bfa::in, int::in, bfa::out).
    beszur(ures, Elem, bfa(Elem, ures, ures)).
    beszur(BF0, Elem, BF):-
        BF0 = bfa(E,B,J), % az indexelés működik!
        ( Elem = E -> BF = BF0
        ; Elem < E ->
            BF = bfa(E,B1,J),
            beszur(B, Elem, B1)
        ; BF = bfa(E,B,J1),
            beszur(J, Elem, J1)
        ).

```

Akkumulálás bináris fákkal — folyt.

Lista konverziója bináris fává

```
lista_bfa(L, BF0, BF) : L element beszúrva BF0-ba kapjuk BF-t.  
:- pred lista_bfa(list(int)::in, bfa::in, bfa::out).  
lista_bfa([], BF, BF).  
lista_bfa([E|L], BF0, BF) :-  
    beszur(BF0, E, BF1),  
    lista_bfa(L, BF1, BF).
```

```
?- lista_bfa([3,1,5], ures, BF).
```

```
F = bfa(3,bfa(1,ures,ures),bfa(5,ures,ures)) ? ;  
0
```

```
?- lista_bfa([3,1,5,1,2,4], ures, BF).
```

```
F = bfa(3,bfa(1,ures,bfa(2,ures,ures)),  
    bfa(5,bfa(4,ures,ures),ures)) ? ;  
0
```

Akkumulálás bináris fákkal — folyt.

Bináris fa konverziója listává

```

bfa_lista(BF, L0, L): A BF fa levelei az L-L0 listát adják.
:- pred bfa_lista(bfa::in, list(int)::in, list(int)::out).
    bfa_lista(ures, L, L).
bfa_lista(bfa(E, B, J), L0, L):-
    bfa_lista(J, L0, L1),
    bfa_lista(B, [E|L1], L).

```

Rendezés bináris fával

```

L lista rendezettje R.
:- pred rendez(list(int)::in, list(int)::out).
    rendez(L, R):-
        lista_bfa(L, ures, BF), bfa_lista(BF, [], R).
?- rendez([1,5,3,1,2,4], R).
R = [1,2,3,4,5] ? ;
no

```

IMPERATÍV PROGRAMOK ÁTÍRÁSA PROLOGBA



Hogyan írjunk át imperatív nyelvű algoritmust Prolog programmá?

- Példafeladat: Hatékony hatványozási algoritmus
- Alaplépés: a kitevő felezése, az alap négyzetre emelése.
- Lényegében 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;
}
```

A `hatv` C függvénynek megfelelő Prolog eljárás

- A függvény eredménye a reláció utolsó arg.-a: `hatv(+A, +H, ?E): AH = E.`
- A ciklusnak segédeljárás felel meg: `hatv(+A0, +H0, +E0, ?E): A0H0 * E0 = E.`
- Az »a« és »h« C változóknak az »+A« és »+H« bemenő *paraméterek*, az »e« C változónak az »+E0, ?E« *akkumulátor-pár* felel meg.

<pre> hatv(A, H, E) :- hatv(A, H, 1, E). hatv(A0, H0, E0, E) :- H0 > 0, !, (H0 /\ 1 == 1 -> E1 is E0*A0 ; E1 = E0), H1 is H0 >> 1, A1 is A0*A0, hatv(A1, H1, E1, E). hatv(_, _, E, E). </pre>	<pre> int hatv(int a, unsigned h) { int e = 1; ism: if (h > 0) { if (h & 1) e *= a; if (h >>= 1; a *= a; goto ism; } else return e; } } </pre>
--	---

A C ciklus és a Prolog eljárás kapcsolata

- A ciklust megvalósító Prolog eljárás minden pontján minden C változónak meg lehet adni egy Prolog változó (pl. h_0, h_1, \dots):
 - 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 ezutáni kódban az új változó felel meg a C változónak.
 - Ha a diszjunkció egyik ága megváltoztat egy változót, akkor a többi ágon is be kell vezetni az új Prolog változót, a réggel azonos értékkel (ld. if ($h \ \& \ 1$) ...).
- A C ciklusmag végén a Prolog eljárást vissza kell hívni, argumentumaiban az egyes C változóknak megadott Prolog változóval.
- A C ciklus *ciklus-invariánsa* nem más mint a Prolog eljárás fejcommentje, a példában:

```
% hatv(+A0, +H0, +E0, ?E): A0H0 * E0 = E.
```

Programhelyesség-bizonyítás

- 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: $x = \text{mfoku_gyok}(a, b, c)$
 - előfeltételek: $b*b-4*a*c \geq 0$, $a \neq 0$
 - 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

- A ciklusokat „fel kell vágni” egy *ciklus-invariáns*-sal, 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) = a0h0 */
{
    int e = 1, a = a0, h = h0;
    while (h > 0)
    {
        /* ciklus-invariáns: a0h0 == e*ah */
        /* indukáláskor a kezdőértékek alapján triviálisan fennáll */
        if (h & 1) e *= a;           /* e' = e * ah&1 */
        h >>= 1;                    /* h' = (h-(h&1))/2 */
        a *= a;                     /* a' = a*a */
    }                               /* indukció: e'*ah' = ... = e*ah */
    return e;
}

/* Az invariánsból h = 0 miatt következik az utófeltétel */
```

Második példa: Fibonacci sorozat tagjainak hatékony számítása

A C függvény

```
unsigned fib(unsigned n)
{ unsigned f = 0, fnxt = 1, t;
  while (n > 0) t = fnxt, fnxt += f, f = t, --n; /* (1) */
  return f;
}
```

Az (1) ciklusnak bemenő változói: n, f, fnxt, kimenő változója: f.

A ciklusnak megfeleltetett Prolog eljárás: fib(N, F0, FNXT, F): az F0 és FNXT kezdőértékű Fibonacci sorozat N-edik tagja F.

<pre>% "betű szerinti" Prolog átírás: fib(N, F0, FNXT, F) :- N > 0, !, T = FNXT, FNXT1 is FNXT+F0, F1 = T, N1 is N-1, fib(N1, F1, FNXT1, F). fib(_, F0, _, F0).</pre>	<pre>% Leegyszerűsített alak: fib(N, F0, FNXT, F) :- N > 0, !, FNXT1 is FNXT+F0, N1 is N-1, fib(N1, FNXT, FNXT1, F). fib(_, F0, _, F0).</pre>
--	--

Fibonacci sorozat — Prolog stílusban

- A Fibonacci sorozat teljes Prolog megvalósítása, és az ennek megfeleltethető

C kód:

```
fib(N, F) :-  
    fib(N, 0, 1, F).  
  
fib(N, F0, F1, F) :-  
    N > 0, !,  
    N1 is N-1,  
    F2 is F0+F1,  
    fib(N1, F1, F2, F).  
  
fib(_, F0, _, F0).  
  
% unsigned fib(unsigned N)  
% { unsigned F0 = 0, F1 = 1, F2;  
%  
% ism:  
%   if (N > 0)  
%   {   --N;  
%       F2 = F0+F1;  
%       F0 = F1; F1 = F2; goto ism;  
%   }  
%   return F0;  
% }
```

MEGOLDÁSOK GYŰJTÉSE ÉS FELSOROLÁSA



Keresési feladat Prologban — felsorolás vagy gyűjtés?

- Keresési feladat: bizonyos 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 visszalépés esetén sorra előáll minden megoldás.

- Egyszerű példa: egy lista páros elemeinek megkeresése:

<pre>% páros_elemei(L, Pk) : Pk az L % lista páros elemeinek listája. páros_elemei([], []). páros_elemei([X L], Pk) :- X mod 2 =\= 0, !, páros_elemei(L, Pk). páros_elemei([P L], [P Pk]) :- páros_elemei(L, Pk).</pre>	<pre>% páros_eleme(L, P) : P egy páros % eleme az L listának. páros_eleme([P L], P) :- P mod 2 == 0. páros_eleme([_ L], P) :- páros_eleme(L, P). páros_eleme2(L, P) :- member(P, L), P mod 2 == 0.</pre>
---	--

Mi a közös a felsoroló és gyűjtő megoldásokban?

- Keressük meg a közös részt a `páros_eleme(i)` eljárásokban!
- Mindkettőben át kell lépni a páratlan elemeket:

```
% köv_páros(L0, P, L) :- Az L0 első páros eleme P, a maradék L.
köv_páros([X|L0], P, L) :-
    X mod 2 =\= 0, !, köv_páros(L0, P, L).
köv_páros([P|L], P, L).
```

- A `köv_páros` eljárásra épülő `páros_eleme(i)` eljárások:

<pre>% páros_elemei(L, Pk) : Pk az L % lista páros elemeinek listája. páros_elemei(L0, Pk) :- köv_páros(L0, P, L1), !, Pk = [P Pk1], páros_elemei(L1, Pk1). páros_elemei(_, []).</pre>	<pre>% páros_eleme(L, P) : P egy páros % eleme az L listának. páros_eleme(L0, P) :- köv_páros(L0, P0, L1), (P = P0 ; páros_eleme(L1, P)).</pre>
--	---

A gyűjtő és felsoroló sémák összehasonlítása

- A páros elemeket gyűjtő ill. felsoroló eljárások alapján adjunk meg egy általános sémát a kétféle eljárástípusra!
- Az általános esetben a keresésnek lehet egy vagy több Param paramétere. Például, kereshetjük az Param-mal osztható elemeket.
- A közös építőelem: következő($V0$, Param, E, $V1$): A $V0$ kifejezéssel jellemzett keresési térben az első megoldás E, és a fennmaradó keresési tér $V1$, a Param paraméter-érték mellett.

A gyűjtő séma:

```
% A V0 keresési térben a Param
% paraméterű megoldások listája L.
megoldások(V0, Param, L) :-
    következő(V0, Param, E, V1), !,
    L = [E|L1],
    megoldások(V1, Param, L1).
megoldások(_, _, []).
```

A felsoroló séma:

```
% A V0 keresési térben E egy
% Param paraméterű megoldás.
megoldás(V0, Param, E) :-
    következő(V0, Param, E0, V1),
    ( E = E0
    ; megoldás(V1, Param, E)
    ).
```

Egy összetettebb példa: fennsíkok felsorolása

- Egy listában fennsíknak nevezünk:
 - egy csupa azonos elemből álló, legalább két elemű, folytonos részlistát;
 - amely az ilyenek között maximális (egyik irányba sem kiterjeszthető).
- A feladat: felsorolandók egy lista fennsíkjai és kezdőpozíciójuk.
- Egy gyorsprogramozási módszerrel készült megoldás:

% Az L listában az F pozíción egy H hosszú fennsík van.

<pre>fennsík0(L, F, H) :- Teste = [E,E _], append(Eleje, Teste, L), \+ last(Eleje, E), length(Eleje, F0), F is F0+1, kezdethossz(Teste, H). % kezdethossz/2 definícióját % lásd korábban</pre>	<pre>fennsík1(L, F, H) :- Teste = [E,E _], append(Eleje, Teste, L), \+ last(Eleje, E), length(Eleje, F0), F is F0+1, (append(Ek, Farok, Teste), \+ Farok = [E _] -> length(Ek, H)).</pre>
--	--

Fennsíkok felsorolása — 2., hatékony megoldás

- Használjuk a megoldás-felsoroló sémát: $megoldás(V0, Param, E)!$
- $V0$: »L, P«, a bejárando lista és első elemének pozíciója;
- $Param$: üres;
- E : »F, H«, a megoldás-fennsík kezdőpozíciója és hossza.

Az L listában az F pozíción egy H hosszú fennsík van.

```
fennsík(L, F, H) :-  
    fennsík(L, 1, F, H).
```

A P0-tól számozott L0 listában az F pozíción egy H hosszú fennsík van.
fennsík(L0, P0, F, H) :-

```
    % az első fennsík jellemzői F0 és H0, az utána levő maradék L1:  
    első_fennsík(L0, P0, F0, H0, L1),  
    ( F = F0, H = H0  
    ; % L1 kezdőpoz. ja, P1 = előző megoldás kezdőpoz. ja+hossza:  
      P1 is F0+H0, fennsík(L1, P1, F, H)  
    ).
```

Fennsíkok felsorolása — 2., hatékony megoldás (folyt.)

első_fennsík(+L0, +P0, -F, -H, -L): A P0-tól számozott L0 listában az első fennsík az F. pozíción van és hossza H, a fennsík után fennmaradó rész pedig az L lista.

első_fennsík([E,E|L1], P0, F, H, L) :-

!, F = P0, azonosak(L1, E, 2, H, L).

első_fennsík([_|L1], P0, F, H, L) :-

P1 is P0+1,

első_fennsík(L1, P1, F, H, L).

azonosak(+L0, +E, +H0, -H, -L): Az L0 lista elejéről a maximális számú E-vel azonos elemet hagyva marad L, a hagyott elemek száma H-H0.

azonosak([X|L0], E, H0, H, L) :-

E = X, !,

H1 is H0+1,

azonosak(L0, E, H1, H, L).

azonosak(L, _, H, H, L).

MEGOLDÁSGYŰJTŐ BEÉPÍTETT ELJÁRÁSOK



Gyűjtés és felsorolás kapcsolata

- Korábban láttuk, hogyan lehet egy keresési feladat gyűjtő és felsoroló eljárásait egy közös magból előállítani.
- Most vizsgáljuk meg, hogyan lehet egy felsoroló eljárást visszavezetni a gyűjtőre, és fordítva:
- felsorolás gyűjtésből: a `member/2` könyvtári eljárás segítségével, pl.

$$\text{páros_eleme}(L, P) :-$$

$$\text{páros_elemei}(L, Pk), \text{member}(P, Pk).$$

Természetesen ez így nem hatékony!

- gyűjtés felsorolásból: a megoldásgyűjtő beépített eljárások segítségével, pl.

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

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

- Az eljárás végrehajtása:
- a Cél kifejezést eljáráshívásként értelmezi, meghívja
(A :+ annotáció meta- (azaz eljárás) argumentumot jelez);
- minden egyes megoldáshoz előállítja Gyűjtő egy *másolatát*
(a megoldásbeli változók, ha vannak, szisztematikusan újakkal helyettesítődnek);
- Az összes Gyűjtő értéket egy listába összegyűjti, és ezt egyesíti Lista-val.
- Példák az eljárás használatára:
 - | ?- findall(X, (member(X, [1,7,8,3,2,4]), X>3), L).
 - \Rightarrow L = [7,8,4] ? ; no
 - | ?- findall(X-Y, (between(1, 3, X), between(1, X, Y)), L).
 - \Rightarrow L = [1-1,2-1,2-2,3-1,3-2,3-3] ? ; no
- Az eljárás jelentése: Lista = { Gyűjtő másolat | ($\exists X \dots Z$)Cél igaz }
ahol X, ..., Z a findall hívásban levő szabad változók (azaz olyan, a hívás pillanatában behelyettesítetlen változók, amelyek a Cél-ban előfordulnak de a Gyűjtő-ben nem).

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

Az eljárás végrehajtása:

- a Cél kifejezést eljárashívásként értelmezi, meghívja;
- összegyűjti a megoldásait (a Gyűjtő-t és a szabad változókat);
- a szabad változók összes behelyettesítését *felsorolja* és mindegyikhez a Lista-ban megadja az összes hozzá tartozó Gyűjtő értéket.

Példák az eljárás használatára:

`gráf([a-b, a-c, b-c, c-d, b-d])`.

| ?- `gráf(_G), findall(B, member(A-B, _G), VegP)`.

\implies `VegP = [b,c,c,d,d]` ? ; no

| ?- `gráf(_G), bagof(B, member(A-B, _G), VegP)`.

\implies `A = a, VegP = [b,c]` ? ;

`A = b, VegP = [c,d]` ? ;

`A = c, VegP = [d]` ? ; no

Az eljárás jelentése: `Lista = { Gyűjtő | Cél igaz }, Lista \neq []`.

A bagof megoldásgyűjtő eljárás (folyt.)

Explicit kvantorok

- bagof(Gyűjtő, $V_1 \sim \dots \sim V_n \sim \text{Cél}$, Lista) alakú hívása a V_1, \dots, V_n változókat egzisztenciálisan kötöttnek tekinti, nem sorolja fel.
- jelentése: $\text{Lista} = \{ \text{Gyűjtő} \mid (\exists V_1, \dots, V_n) \text{Cél igaz} \} \neq []$.
 $\mid ?\text{-gráf}(_G), \text{bagof}(B, A \sim \text{member}(A-B, _G), \text{VegP}).$
 $\implies \text{VegP} = [b, c, c, d, d] \text{ ? ; no}$

Egymásba ágyazott gyűjtések

- szabad változók esetén a bagof nemdet. lehet, így skatulyázható:

```
% A G irányított gráf fokszámlistája FL:
% FL = { A-N | N = |{ V | A-V ∈ G }| }
fokszámai(G, FL) :-
    bagof(A-N, V~(bagof(V, member(A-V, G), V_k),
        length(V_k, N)
    ), FL).
| ?- gráf(_G), fokszámai(_G, FL).  $\implies$  FL = [a-2,b-2,c-1] ? ; no
```

A bagof megoldásgyűjtő eljárás (folyt.)

Fokszámlista hatékonyabb előállítás

- a vezérlési szerkezeteket célszerű elkerülni a meta-argumentumokban
- segédeljárás bevezetésével a kvantor is szükségtelemmé válik:

% Az A pont foka a G irányított gráfban N>0.

`pont_foka(A, G, N) :- bagof(V, member(A-V, G), Vks), length(Vks, N).`

% A G irányított gráf fokszámlistája FL:

`fokszamai(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 időigényesebb!

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

- az eljárás végrehajtása:
- ugyanaz mint: bagof(Gyűjtő, Cél, L0), sort(L0, Lista),
- itt sort/2 egy univerzális rendező eljárás (lásd később), amely
- az eredménylistát rendezi (az ismétlődések kiszűrésével).

- Példa a setof/3 eljárás használatára:

```
gráf([a-b,a-c,b-c,c-d,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).  $\implies$  Pk = [a,b,c,d] ? ; no
```

META-LOGIKAI ELJÁRÁSOK



A meta-logikai, azaz a logikán túlmutató eljárások fajtái:

- A Prolog kifejezések pillanatnyi behelyettesíthetőségi állapotát vizsgáló eljárások (értelmszerűen sorrendfüggőek):
 - kifejezések osztályozása (1)
 - | ?- var(X) /* X változó? */ , X = 1. \Rightarrow X = 1
 - | ?- X = 1, var(X) . \Rightarrow no
 - kifejezések rendezése (4)
 - | ?- X @< 3 /* X megelőzi 3-t? */ , X = 4. \Rightarrow X = 4
 - % *a változók megelőzik a nem változó kifejezéseket*
 - | ?- X = 4, X @< 3. \Rightarrow no
- Prolog kifejezéseket szétszedő vagy összerakó eljárások:
 - (struktúra) kifejezés \iff név és argumentumok (2)
 - | ?- X = f(alma,körte) , X = . . L \Rightarrow L = [f,alma,körte]
 - atomok és számok \iff karaktereik (3)
 - | ?- atom_codes(A, [0'a,0'b,0'a] \Rightarrow A = aba

Kifejezések osztályozása

- Kifejezés-osztályok faststruktúrája — osztályozó beépített eljárások

<i>Kifejezés</i>		
var	nonvar	var(X) X változó
atomic compound		nonvar(X) X nem változó
atom	number	atomic(X) X konstans
integer	float	compound(X) X struktúra
		atom(X) X atom
		number(X) X szám
		integer(X) X egész szám
		float(X) X lebegőpontos szám

- SICStus-specifikus osztályozó eljárások:

- simple(X): X nem összetett (konstans vagy változó);
- ground(X): X tömör, azaz nem tartalmaz behelyettesíthető változót.
- Az osztályozó eljárások használata — példák
 - var, nonvar — többirányú eljárásokban a különböző irányok elágaztatása
 - number, atom, ... — nem-megkülönböztetett úniók feldolgozása

Osztályozó eljárások: elágaztatás behelyettesítettség alapján

● Példa: a length/2 beépített eljárás megvalósítása (SICStus kód!)

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

<pre>% 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).</pre>	<pre>% dlength(?L, +I0, +I): % Az L lista I-I0 hosszú. dlength([], I, I) :- !. dlength([_ L], I0, I) :- IO<I, I1 is IO+1, dlength(L, I1, I).</pre>
---	---

<pre> ?- length([1,2], Len). ?- length([1,2], 3). ?- length(L, 3). ?- length(L, Len).</pre>	<pre>(length/3) => Len = 2 ? ; no (dlength/3) => no (dlength/3) => L = [_A,_B,_C] ? ; no (length/3) => L = [], Len = 0 ? ; L = [_A], Len = 1 ? ; L = [_A,_B], Len = 2 ?</pre>
--	---

Osztályozó eljárások: nem-megkülönböztetett úniók kezelése

- Példa: egy formula természetes módon ábrázolható Prologban, pl. $x*y+y+1$.
A formula típusának leírásához nem-megkülönböztetett úniót kell használni:

```
% :- type form == atom \\/ int \\/ { form+form } \\/ { form-form } ...
```

- Példa formulakezelésre: szimbolikus derivált előállítás

```
% deriv(+Kif, +X, ?D): Kif-nek az X atom szerinti deriváltja D.
deriv(X, X, D) :- !, D = 1.
deriv(C, _X, D) :- atomic(C), !, D = 0.
deriv(U+V, X, DU+DV) :-          deriv(U, X, DU), deriv(V, X, DV).
deriv(U-V, X, DU-DV) :-          deriv(U, X, DU), deriv(V, X, DV).
deriv(U*V, X, DU*V + U*DV) :-    deriv(U, X, DU), deriv(V, X, DV).
deriv(U/V, X, (DU*V - U*DV)/(V*V)) :- deriv(U, X, DU), deriv(V, X, DV).

| ?- deriv(x*y+1, x, DX), deriv(x*y+1, y, DY).
    => DX = 1*y+x*0+0, DY = 0*y+x*1+0 ? ; no

| ?- deriv((x+y)*(2+x), x, D).
    => D = (1+0)*(2+x)+(x+y)*(0+1) ? ; no
```

Struktúrák szétszedése és összerakása: az *univ* eljárás

- Az *univ* eljárás hívási mintái:
 - $+Kif = .. \quad ?Lista$
 - $-Kif = .. \quad +Lista$
- Az eljárás jelentése: Igaz, ha
 - $Kif = Fun(A_1, \dots, A_n)$ és $Lista = [Fun, A_1, \dots, A_n]$, ahol Fun egy atom és A_1, \dots, A_n tetszőleges kifejezések; vagy
 - $Kif = C$ és $Lista = [C]$, ahol C egy konstans.

Példák

?- el(a,b,10) =.. L.	\Rightarrow	L = [el,a,b,10]
?- Kif =.. [el,a,b,10].	\Rightarrow	Kif = el(a,b,10)
?- alma =.. L.	\Rightarrow	L = [alma]
?- Kif =.. [1234].	\Rightarrow	Kif = 1234
?- Kif =.. L.	\Rightarrow	hiba
?- f(a,g(10,20)) =.. L.	\Rightarrow	L = [f,a,g(10,20)]
?- Kif =.. [/ ,X,2+X].	\Rightarrow	Kif = X/(2+X)
?- [a,b,c] =.. L.	\Rightarrow	L = [?,a,[b,c]]

Struktúrák szétszedése és összerakása: a functor eljárás

- functor/3: kifejezés funktorának, adott funktorú kifejezésnek az előállítás
- Hívási minták: `functor(-Kif, +Név, +Argszám)`
`functor(+Kif, ?Név, ?Argszám)`
- Jelentése: igaz, ha 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(e1(a,b,1), F, N).	\implies	F = e1, N = 3
?- functor(E, e1, 3).	\implies	E = e1(_A,_B,_C)
?- functor(alma, F, N).	\implies	F = alma, N = 0
?- functor(Kif, 122, 0).	\implies	Kif = 122
?- functor(Kif, e1, N).	\implies	hiba
?- functor(Kif, 122, 1).	\implies	hiba
?- functor([1,2,3], F, N).	\implies	F = '._', N = 2
?- functor(Kif, ., 2).	\implies	Kif = [_A _B]

Struktúrák szétszedése és összerakása: az `arg` eljárás

- `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* (kétirányúság!).

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)
```

Az *univ* alkalmazása: ismétlődő sémák összevonása

- A feladat: egy szimbolikus aritmetikai kifejezésben a kiértékelhető (infix) rész kifejezések helyettesítése az értékükkel.

- 1. megoldás, *univ* nélkül:

```
% Az X szimbolikus kifejezés egyszerűsítése EX.
egysz0(X, EX) :- atomic(X), !, EX = X.
egysz0(U+V, EKif) :-
    egysz0(U, EU), egysz0(V, EV), kiszamol(EU+EV, EU, EV, EKif).
egysz0(U*V, EKif) :-
    egysz0(U, EU), egysz0(V, EV), kiszamol(EU*EV, EU, EV, EKif).
%...
% EU és EV részekből képzett EUV egyszerűsítése EKif.
kiszamol(EUV, EU, EV, EKif) :-
    number(EU), number(EV), !, EKif is EUV.
kiszamol(EUV, _, _, EUV).

| ?- deriv((x+y)*(2+x), x, D), egysz0(D, ED).
=> D = (1+0)*(2+x)+(x+y)*(0+1), ED = 1*(2+x)+(x+y)*1 ? ; no
```

Az *univ* alkalmazása: ismétlődő sémák összevonása (folyt.)

• Kifejezés-egyszerűsítés, 2. megoldás, *univ* segítségével

```
egysz(X, EX) :- atomic(X), !, EX = X.
egysz(Kif, EKif) :-
    Kif =.. [Muv,U,V],    % Kif = Muv(U,V)
    egysz(U, EU), egysz(V, EV),
    EUV =.. [Muv,EU,EV], % EUV = Muv(EU,EV)
    kiszamol(EUV, EU, EV, EKif).
```

• Kifejezés-egyszerűsítés, általánosítás tetszőleges tömör kifejezésre:

```
egysz1(Kif, EKif) :-
    Kif =.. [M|ArgL], egysz1_lista(ArgL, EArgL), EKif0 =.. [M|EArgL],
    % catch(:+Cél,?Kiv,:+KCél): ha Cél kivételt dob, KCél-t futtatja:
    catch(EKif is EKif0, _, EKif = EKif0).

egysz1_lista([], []).
egysz1_lista([K|Kk], [E|Ek]) :- egysz1(K, E), egysz1_lista(Kk, Ek).

| ?- egysz1(f(1+2+a, exp(3,2), a+1+2), E) => E = f(3+a,9.0,a+1+2)
```

Unio alkalmazása általános kifejezés-bejárásra: kiiratás

- A feladat: egy tetszőleges kifejezés kiiratása úgy, hogy
- a kétargumentumú operátorok zárójelezett infix formában,
- minden más alap-struktúra alakban jelenjék meg.

```

ki(Kif) :- compound(Kif), !, /* Kif itt biztosan nem változó: */
    Kif =.. [Func, A1|ArgL],
    ( current_op(_, Kind, Func), (Kind=xfy;Kind=yfx;Kind=xff),
      ArgL = [A2] -> % kétargumentumú operátor
        write('('), ki(A1), format('~w ', [Func]), ki(A2), write(')')
      ; write(Func), write('('), ki(A1), arglistaki(ArgL), write(')')
    ).
ki(Kif) :- write(Kif).

% Az [A1,...,An] listát ",A1,...,An" alakban kiírja.
arglistaki([]).
arglistaki([A|AL]) :- write(', '), ki(A), arglistaki(AL).

| ?- ki(f(+a, X*c*X, e)). ==> f(+a),((_117 * c) * _117),e)

```

Univ alkalmazása általános kifejezés-bejárásra: változómentesítés

- A SICStus Prologban beépített `numbervars(?Kif, +NO, ?N)` eljárás hatása:
 - A tetszőleges `Kif` minden változóját `'$VAR'(I)` alakú kifejezéssel helyettesíti, $I = NO, \dots, N-1$ (azaz `Kif`-ben $N-NO$ különböző változó van).
- A `'$VAR'(0), '$VAR'(1), \dots` kifejezések `write`-tal való kiíráskor változónévként $(A, B \dots)$ jelennek meg.
- Ezek speciális opciókkal `write_term`-mel „eredetiben” is megjeleníthetők:


```
| ?- _Kif = [f(_X),g(_),_X], numbervars(_Kif, 0, N), write(_Kif), nl,
      write_term(_Kif, [quoted(true),numbervars(false)]).
      => [f(A),g(B),A]
          [f('$VAR'(0)),g('$VAR'(1)),$'VAR'(0)]
          N = 2
```
- A feladat: elkészítendő egy `numbervars1/3` eljárás, amely `'$VAR'` helyett `'$myvar'` funktort használ.

Általános kifejezés-bejárás *univ*-val : saját változómentesítés

```

% A Term kifejezésben levő változókat 'myvar(I)' stb.
% struktúrákkal helyettesíti be, I = NO, ... N-1.
numbervar1(Term, NO, N) :- var(Term), !,
    Term = 'myvar'(NO), N is NO+1.
numbervar1(Term, NO, N) :-
    Term =.. [_|Args], numbervar1_list(Args, NO, N).

% numbervar1_list(L, NO, N): Az L listában levő változókat
% 'myvar(I)' stb. struktúrákkal helyettesíti be, I = NO, ... N-1.
numbervar1_list([], N, N).
numbervar1_list([A|As], NO, N) :-
    numbervar1(A, NO, N1), numbervar1_list(As, N1, N).

?- Kif = [f(_X),g(_),_X], numbervar1(Kif, 0, N).
    => N = 2,
    Kif = [f('myvar'(0)),g('myvar'(1)),'myvar'(0)]

```

Unio alkalmazása: részkiejezések keresése

- A feladat: egy tetszőleges kifejezéshez soroljuk fel a benne levő számokat, és minden szám esetén adjuk meg annak a *kiválasztóját!*
 - Egy részkiejezés kiválasztója egy olyan lista, amely megadja, mely argumentumpozíciók mentén juthatunk el hozzá.
 - Az $[i_1, i_2, \dots, i_k]$ lista egy Kif-ből az i_1 -edik argumentum i_2 -edik argumentumának, \dots i_k -adik argumentumát választja ki.
 - Pl. $a*b+f(1, 2, 3)/c$ -ben b kiválasztója $[1, 2]$, 3 kiválasztója $[2, 1, 3]$.
- ```
% kif_szám(?Kif, ?N, ?Kiv): Kif Kiv kiválasztójú része az N szám.
kif_szám(X, N, Kiv) :- number(X), !, N = X, Kiv = [].
kif_szám(X, N, [I|Kiv]) :-
 compound(X), % a változó kizárása miatt fontos!
 functor(X, F, N), between(1, N, I), arg(I, X, X1),
 kif_szám(X1, N, Kiv).
```
- ```
| ?- kif_szám(f(1, [b, 2]), N, K). => K=[1], N=1? ; K=[2, 2, 1], N=2? ; no
```

Atomok szétszedése és összerakása

- atom_codes/2: atom és karakterkód-lista közötti átalakítás
- Hívási minták: atom_codes(+Atom, ?Kódlista)
atom_codes(-Atom, +Kódlista)
- Jelentése: Igaz, ha Atom karakterkódjainak a listája Kódlista.
- Végrehajtása:
 - Ha Atom bemenő, és a $c_1c_2...c_n$ karakterekből áll, akkor Kódlista = $[k_1, k_2, ..., k_n]$, ahol k_i a c_i karakter kódja.
 - Ha Atom kimenő, akkor a Kódlista karakterkód-listából összerak egy atomot, és azt egyesíti Atom-mal.

• Példák:

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

Atomok szétszedése és összerakása — alkalmazási példák

• Keresés atomokban

% Atom-ban a Rész nem üres részatom kétszer ismétlődik.

dadogó_rész(Atom, Rész) :-

atom_codes(Atom, Cs), dadogó(Cs, Ds), atom_codes(Rész, Ds).

% L-ben a D nem üres részlista kétszer ismétlődik (lásd korábban).

dadogó(L, D) :- D = [_|_],

append(_, Farok, L), append(D, Vég, Farok), append(D, -, Vég).

| ?- dadogó_rész(babaruhaha, R). \implies R = ba ? ; R = ha ? ; no

• Atomok összefűzése

% atom_concat(+A, +B, ?C): A és B atomok 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). \implies 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 tízes számrendszerbeli alakja a Kódlista karakterkód-listának felel meg.
- Végrehajtása:
 - Ha Szám bemenő, és tízes számrendszerben a $c_1c_2\dots c_n$ karakterekből áll, akkor Kódlista = $[k_1, k_2, \dots, k_n]$, ahol k_i a c_i karakter kódja.
 - Ha Szám kimenő, akkor a Kódlista karakterkód-listából összerak egy számot, és azt egyesíti Szám-mal.

Példák:

?- <code>number_codes(12, Cs).</code>	\Rightarrow <code>Cs = [49, 50]</code>
?- <code>number_codes(0123, [0'1 L]).</code>	\Rightarrow <code>L = [50, 51]</code>
?- <code>number_codes(N, " - 12.0e1").</code>	\Rightarrow <code>N = -120.0</code>
?- <code>number_codes(N, "12e1").</code>	\Rightarrow hiba (nincs .0)

Kifejezések rendezése: szabványos sorrend

Legyen X és Y két tetszőleges Prolog kifejezés, ha X megelőzi Y -t, azt írjuk, hogy $X \prec Y$.

1. Ha X és Y azonos, akkor sem $X \prec Y$ sem $Y \prec X$ nem igaz és fordítva.
 2. Ha X és Y különböző kifejezésosztályba tartozik, akkor az osztály dönt: *változó* \prec *lebegőpontos szám* \prec *egész szám* \prec *név* \prec *struktúra*.
 3. Ha X és Y változó, akkor az eredmény rendszerfüggő.
 4. Ha X és Y lebegőpontos vagy egész szám, akkor $X \prec Y \Leftrightarrow X < Y$.
 5. Ha X és Y név, akkor sorrendjük megegyezik az abc sorrenddel.
 6. Ha X és Y struktúrák:
 - 6.1. Ha X és Y aritása különböző, $X \prec Y \Leftrightarrow X$ aritása kisebb mint Y aritása.
 - 6.2. Egyébként, ha a rekordok neve különböző, $X \prec Y \Leftrightarrow X$ neve $\prec Y$ neve.
 - 6.3. Egyébként balról az első nem azonos argumentum dönt.
- Végtelen (ciklikus) kifejezésekre a fenti rendezés nem érvényes.

Kifejezések összehasonlítása — beépített eljárások

- Két tetszőleges kifejezés összehasonlítását végző eljárások:

hívás	igaz, ha
Kif1 == Kif2	Kif1 \neq Kif2 \wedge Kif2 \neq Kif1
Kif1 \== Kif2	Kif1 \prec Kif2 \vee Kif2 \prec Kif1
Kif1 @< Kif2	Kif1 \prec Kif2
Kif1 @=< Kif2	Kif2 \neq Kif1
Kif1 @> Kif2	Kif2 \prec Kif1
Kif1 @>= Kif2	Kif1 \neq Kif2

- Az összehasonlító eljárások logikailag nem tiszták:

| ?- X @< 3, X = 4. \implies X = 4
 | ?- X = 4, X @< 3. \implies no

- Az összehasonlítás mindig a belső ábrázolás szerint történik:

| ?- [1, 2, 3, 4] @< struktúra(1, 2, 3). \implies sikerül (6.1 szabály)

A meta-logikai eljárások egy komplex alkalmazása: \prec megvalósítása

```

% T1 megelőzi T2-t a szabványos sorrendben (lényegében T1 @< T2)
precedes(T1, T2) :- \+ \+ (number1(T1-T2, 0, _), prec(T1, T2)).

% T1 megelőzi T2-t, a változók már '$myvar'(n) konstansokra cseréltek.
prec(T1, T2) :- class(T1, C1), class(T2, C2),
    (
        C1 == C2 ->
            (
                C1 == 1 -> T1 < T2    % 4. szabály (lebegőpontos szám)
            ;
                C1 == 2 -> T1 < T2    % 4. szabály (egész szám)
            ;
                struct_prec(T1, T2)   % 3., 5. és 6. szabály
            )
        ;
        C1 < C2
    ).

```

% class(+T, -C): A T kifejezés a C-edik kifejezésosztályba tartozik.

```

class(T, C) :- ( /*vált*/ T='$myvar'(_) -> C=0
    ; /*szám*/ float(T) -> C=1 ; integer(T) -> C=2
    ; /*atom*/ atom(T) -> C=3 ; /*struktúra*/ C=4
    ).

```

\prec reláció megvalósítása (folyt.)

```

% S1 megelőzi S2-t (struktúra-kifejezésekre és atomokra).
struct_prec(S1, S2) :- functor(S1, F1, N1), functor(S2, F2, N2),
    (   N1 < N2 -> true
      ;   N1 = N2, (   F1 = F2 -> args_prec(1, N1, S1, S2)
                      ;   atom_prec(F1, F2)
                )
      ).

% Az S1 struktúra-kifejezés N0, ..., N sorszámú argumentuma
% lexicografikusan megelőzik S2 azonos sorszámú argumentumait.
args_prec(N0, N, S1, S2) :- N0 =< N, arg(N0, S1, A1), arg(N0, S2, A2),
    (   A1 = A2 -> N1 is N0+1, args_prec(N1, N, S1, S2)
      ;   prec(A1, A2)
    ).

% A1 atom megelőzi A2 atomot (előfeltétel: A1 \= A2).
atom_prec(A1, A2) :-
    atom_codes(A1, C1), atom_codes(A2, C2), struct_prec(C1, C2).

```

EGYENLŐSÉGTÁJTÁK — ÖSSZEFOGLALÁS



A Prolog egyenlőség-szerű beépített eljárásai

• $U = V : U$ egyesítendő V -vel. Soha sem jelez hibát.	?- $X = 1+2.$ \implies $X = 1+2$?- $3 = 1+2.$ \implies no
• $U == V : U$ azonos V -vel. Soha sem jelez hibát és soha sem helyettesít be.	?- $X == 1+2.$ \implies no ?- $3 == 1+2.$ \implies no ?- $+(1, 2) == 1+2 \implies$ yes
• $U ::= V : \text{Az } U \text{ és } V \text{ aritmetikai kifejezések értéke megegyezik.}$ Hibát jelez, ha U vagy V nem (tömör) aritmetikai kifejezés.	?- $X ::= 1+2.$ \implies hiba ?- $1+2 ::= X.$ \implies hiba ?- $2+1 ::= 1+2.$ \implies yes ?- $2.0 ::= 1+1.$ \implies yes ?- $2.0 \text{ is } 1+1.$ \implies no
• $U \text{ is } V : U$ egyesítendő a V aritmetikai kifejezés értékével. Hiba, ha V nem (tömör) aritmetikai kifejezés.	?- $X \text{ is } 1+2.$ \implies $X = 3$?- $1+2 \text{ is } X.$ \implies hiba ?- $3 \text{ is } 1+2.$ \implies yes ?- $1+2 \text{ is } 1+2.$ \implies no
• $(U =.. V : U$ „szétszedettje” a V lista)	?- $1+2 =.. X.$ \implies $X = [+ , 1 , 2]$?- $X =.. [f , 1]. \implies$ $X = f(1)$

A Prolog nem-egyenlőség jellegű beépített eljárásai

• A nem-egyenlőség jellegű eljárások sohasem helyettesítenek be változót!

• $U \backslash = V$: U nem egyesíthető V -vel.
Soha sem jelez hibát.

	? -	$X \backslash = 1+2.$	\implies	no
	? -	$+(1, 2) \backslash = 1+2.$	\implies	no

• $U \backslash == V$: U nem azonos V -vel.
Soha sem jelez hibát.

	? -	$X \backslash == 1+2.$	\implies	yes
	? -	$3 \backslash == 1+2.$	\implies	yes
	? -	$+(1, 2) \backslash == 1+2$	\implies	no

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

	? -	$X \backslash = 1+2.$	\implies	hiba
	? -	$1+2 \backslash = X.$	\implies	hiba
	? -	$2+1 \backslash = 1+2.$	\implies	no
	? -	$2. 0 \backslash = 1+1.$	\implies	no

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

		<i>Egyesítés</i>		<i>Azonosság</i>		<i>Aritmetika</i>		
U	V	$U = V$	$U \setminus = V$	$U == V$	$U \setminus == V$	$U := V$	$U \setminus = V$	$U \text{ is } V$
1	2	<i>no</i>	<i>yes</i>	<i>no</i>	<i>yes</i>	<i>no</i>	<i>yes</i>	<i>no</i>
a	b	<i>no</i>	<i>yes</i>	<i>no</i>	<i>yes</i>	error	error	error
1+2	+(1,2)	<i>yes</i>	<i>no</i>	<i>yes</i>	<i>no</i>	<i>yes</i>	<i>no</i>	<i>no</i>
1+2	2+1	<i>no</i>	<i>yes</i>	<i>no</i>	<i>yes</i>	<i>yes</i>	<i>no</i>	<i>no</i>
1+2	3	<i>no</i>	<i>yes</i>	<i>no</i>	<i>yes</i>	<i>yes</i>	<i>no</i>	<i>no</i>
3	1+2	<i>no</i>	<i>yes</i>	<i>no</i>	<i>yes</i>	<i>yes</i>	<i>no</i>	<i>yes</i>
X	1+2	$X=1+2$	<i>no</i>	<i>no</i>	<i>yes</i>	error	error	$X=3$
X	Y	$X=Y$	<i>no</i>	<i>no</i>	<i>yes</i>	error	error	error
X	X	<i>yes</i>	<i>no</i>	<i>yes</i>	<i>no</i>	error	error	error

elmagyarázat: *yes* — siker; *no* — meghiusulás, **error** — hiba.

MODULARITĀS



Modulok definiálása SICStus Prolog nyelven

- A SICStus Prolog modulfogalmának jellemzői:
- Minden modul külön állományba kell kerüljön.
- Az állomány első programeleme egy modul-parancs kell legyen:

```
:- module( ModulNév, [ExpFunktor1, ExpFunktor2, ...] ).
```

ExpFunktor = az exportálandó eljárás funktora (név/argumentumszám)
- Pl. :- module(platók, [fennsík/3]). % *plato állomány első sora*
- Modul-betöltésre szolgáló beépített eljárások:
 - use_module(*ÁllományNév*)
 - use_module(*ÁllományNév*, [*ImpFunktor1*, *ImpFunktor2*, ...])
ImpFunktor — az importálandó eljárás funktora
 - *ÁllományNév* lehet atom, vagy pl. library(*KönyvtárNév*):
 - :- use_module(plato). % *a fenti modul betöltése*
 - :- use_module(library(lists), [last/2]). % *csak last/2 importált*
- A modulfogalom nem szigorú: platók:első_fennsík(...) **meghívható!**
- Modulvalifikált hívási forma: *ModulNév:EljárásNév(Argumentumok ...)*.

Meta-eljárások modularizált programban

- Eljárások átadása paraméterként modulközi hívásban gondot okozhat:

- `m1.pl` állomány:

```
:- module(m1, [kétyszer/1]).
kétyszer(X) :- X, X.

p :- write(bu).
```

- `m2.pl` állomány:

```
:- module(m2, [q/0,r/0]).
:- use_module(m1).
q :- kétyszer(p).  r :- kétyszer(m2:p).

p :- write(ba).
```

- Futtatás:

```
| ?- [m1,m2].
| ?- q.      => bubu
| ?- r.      => baba
```

- Automatikus modul-kvalifikáció meta-predikátum deklarációval:

```
Hu m1.pl-be beszúrjuk: :- meta_predicate kétyszer(:), akkor
| ?- q.      => baba!
```

Meta-predikátum deklaráció, modulnév-kiterjesztés

- Meta-predikátum deklaráció
 - Formája:
 - $\text{:- meta_predicate } \langle \text{eljárásnév} \rangle (\langle \text{módspec}_1 \rangle, \dots, \langle \text{módspec}_n \rangle), \dots$
 - $\langle \text{módspec}_i \rangle$ lehet '.' , '+' , '-' , vagy '?' .
 - A '.' mód azt jelzi, hogy az adott argumentumot betöltéskor ún. modulnév-kiterjesztésnek kell alávetni.
 - Egy Kif kifejezés modulnév-kiterjesztése:
 - ha $M:X$ alakú (vagy egy olyan változó, amely az adott eljárás fejében meta-argumentum pozíción szerepelt) akkor változatlanul hagyjuk;
 - egyébként helyettesítjük $CurMod:Kif$ -fel ($CurMod$ a kurrens modul).
 - Példa folyt. (az m1-beli kétszer meta-predikátumnak deklarált!)
- ```
:- module(m2, [négyyszer/1, q/0]). :- use_module(m1).
q :- kétszer(p). \implies q :- kétszer(m2:p).
:- meta_predicate négyyszer(:).
négyyszer(X) :- kétszer(X), kétszer(X). \implies változatlan
```

# MAGASABBRENDŰ ELJÁRÁSOK



## Magasabbrendű eljárások — listakezelés

- Magasabbrendű (vagy meta-eljárás) egy eljárás,
  - ha eljárásként értelmezi egy vagy több argumentumát
  - pl. `call/1`, `findall/3`, `\+ /1`, stb.

- Listafeldolgozás `findall` segítségével, példák:

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

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

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

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



## Listakezelő meta-eljárások megoldásgyűjtő eszközökkel

- Lista szűrése (vö. a filter SML függvénnyel!)

```
% Az L lista X elemeinek Pred szerinti szűrése FL.
```

```
:- meta_predicate filter(+, ?, :, -).
```

```
filter(L, X, Pred, FL) :-
```

```
 findall(X, (member(X, L), call(Pred)), FL).
```

```
| ?- filter([1,2,3,4], X, X mod 2 == 0, Pk). \implies Pk = [2,4]
```

- Lista leképezése (vö. a map SML függvénnyel!)

```
% Az L lista X elemeit Pred-del Y-ba képezve kapjuk az ML listát.
```

```
:- meta_predicate map(+, ?, :, ?, -).
```

```
map(L, X, Pred, Y, ML) :-
```

```
 findall(Y, (member(X, L), Pred), ML).
```

```
| ?- map([1,2,3,4], X, Y is X*X, Y, Nk). \implies Nk = [1,4,9,16]
```

- A példákban a szűrést az  $\langle X, \text{Pred} \rangle$  argumentumpár, a leképezést az  $\langle X, \text{Pred}, Y \rangle$  hármas határozza meg. Ezek egy-egy- ill. kétargumentumú predikátumot adnak meg (vö. a funkcionális nyelvek  $\lambda$ -kifejezéseivel).

## Részlegesen paraméterezett eljáráshívások

- A listát elemenként négyzetreemelő eljárás egy másik változata:  
 $\text{négyzete}(X, Y) :- Y \text{ is } X * X.$   
 $\text{négyzeteik}(X_k, Y_k) :- \text{map}(X_k, X, \text{négyzete}(X, Y), Y, Y_k).$
- A lista elemeire az  $x \rightarrow x^2 + Px + Q$  hozzárendelést alkalmazó eljárás:  
 $\text{másodfokú_képe}(P, Q, X, Y) :- Y \text{ is } X * X + P * X + Q.$   
 $\text{másodfokú_képeik}(P, Q, X_k, Y_k) :-$   
 $\text{map}(X_k, X, \text{másodfokú_képe}(P, Q, X, Y), Y, Y_k).$
- Konvenció: a meta-alkalmazásban változó paramétereket az eljárás végére tesszük — így egyszerűsíthető a meta-eljárás hívása. Példa: `map/3`:  
 $\text{map}(X_k, \text{RészlPred}, Y_k) :-$   
 $\text{RészlPred} =.. L0, \text{append}(L0, [X, Y], L), \text{Pred} =.. L,$   
 $\text{findall}(Y, (\text{member}(X, X_k), \text{Pred}), Y_k).$   
 $\text{másodfokú_képeik}(P, Q, X_k, Y_k) :- \text{map}(X_k, \text{másodfokú_képe}(P, Q), Y_k).$

## Részlegesen paraméterezett eljáráshívások — segédesszközök

- A `call/1` eljárás általánosítása: a `call/2`, `call/3`, ... eljárások.
- `call(RPred, A1, A2, ...)` végrehajtása: az `RPred` hívást kiegészíti az `A1, A2, ...` argumentumokkal, és meghívja.
- A `call/N` eljárások sok Prologban beépítettek, SICStusban definiálandók:
  - `:- meta_predicate call(:, ?), call(:, ?, ?), ....`
  - `% Pred az A utolsó argumentummal meghívva igaz.`
  - `call(M:Pred, A) :-`
    - `Pred =.. FAS0, append(FAS0, [A], FAS1), Pred1 =.. FAS1,`
    - `call(M:Pred1).`
  - `% Pred az A és B utolsó argumentumokkal meghívva igaz.`
  - `call(M:Pred, A, B) :-`
    - `Pred =.. FAS0, append(FAS0, [A,B], FAS2), Pred2 =.. FAS2,`
    - `call(M:Pred2).`
  - ...

## Részlegesen paraméterezett eljárások — rekurzív map/3

- map/3 rekurzív definíciója:

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

- Példák:

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

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

## Rekurzív meta-eljárások — foldl és foldr

```

% foldl(Xs, Pred, Y0, Y): Az Xs elemeire balról jobbra alkalmazott,
% a Pred által leírt kétargumentumú függvény Y0 kezdőértékre
% alkalmazott eredménye Y.
foldl([X|Xs], Pred, Y0, Y) :-
 call(Pred, X, Y0, Y1), foldl(Xs, Pred, Y1, Y).
foldl([], _, Y, Y).

% foldl([], _, Y, Y).
egyhozzá(A, J, E0, E) :- E is E0*A+J.

?- foldl([1,2,3], jegyhozzá(10), 0, E). => E = 123

% foldr(Xs, Pred, Y0, Y): Az Xs elemeire jobbról balra alkalmazott, a
% Pred által leírt függvény Y0 kezdőértékre alkalmazott eredménye Y.
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

```

# DINAMIKUS ADATBÁZISKEZELÉS



## Dinamikus predikátumok

---

- A dinamikus predikátum jellemzői:
  - a program szövegében lehet 0 vagy több klóza;
  - futási időben hozzáadhatunk és elvehetünk klózokat belőle;
  - végrehajtása mindenképpen interpretált.
- Létrehozása
- programszövegbeli deklarációval:
  - :- dynamic(Eljárásnév/Argumentumszám).(ha van klóza a programban, akkor az első előtt — ilyenkor kötelező);
- futási időben, adatbáziskezelő beépített eljárással
- Adatbáziskezelő eljárások („adatbázis” = a program klózainak összessége):
  - klóz felvétele első, utolsó helyre: asserta/1, assertz/1
  - klóz törlése (illesztéssel, többszörösen sikerülhet): retract/1
  - klóz lekérdezése (illesztéssel, többszörösen sikerülhet): clause/2

## Klóz felvétele: `asserta/1`, `assertz/1`

---

- `asserta(:@Klóz)`
- A Klóz kifejezést klózként értelmezve felveszi a programba az adott predikátum *első* klózaként.
- A ‘@’ mód jelentése: tisztán bemenő paraméter, az eljárás a paraméterbeli változókat nem helyettesíti be (a ‘+’ mód speciális esete).
- A ‘:’ mód modul-kvalifikált paramétert jelez.
- `assertz(:@Klóz)`
- A Klóz kifejezést az adott predikátum *utolsó* klózként veszi fel

- Példa:

```
| ?- assertz((p(1,X):-q(X))), asserta(p(2,0)),
 assertz((p(2,Z):-r(Z))), listing(p).
```

```
p(2, 0).
```



$$\begin{aligned} p(1, A) &:- q(A). \\ p(2, A) &:- r(A). \end{aligned}$$

## Klóz törlése: `retract/1`

- `retract(:@Klóz)`

- A Klóz kifejezésből megállapítja a predikátum funktorát.
- Az adott predikátum klózeit sorra megpróbálja illeszteni Klóz-zal.
- Ha az illesztés sikerült, akkor kitörli a klózt és sikeresen lefut.
- Visszalépés esetén folytatja a keresést (illeszt, töröl, sikerül, stb.)

- Példa (folytatás):

| ?- `listing(p)`, `retract((p(2,_):-_))`, `listing(p)`, `fail`.  $\implies$  no

- A futás kimenete:

|                         |                    |                         |                    |                         |
|-------------------------|--------------------|-------------------------|--------------------|-------------------------|
| <code>p(2, 0).</code>   |                    | <code>p(1, A) :-</code> |                    | <code>p(1, A) :-</code> |
| <code>p(1, A) :-</code> | <code>q(A).</code> |                         | <code>q(A).</code> |                         |
| <code>p(2, A) :-</code> |                    | <code>p(2, A) :-</code> |                    | <code>r(A).</code>      |
| <code>r(A).</code>      |                    |                         |                    |                         |

## Alkalmazási példa — egyszerűsített findall

- A findall1/3 eljárás hatása megegyezik a beépített findall-lal, de
  - nem működik helyesen, ha a Cél-ban újabb findall1 hívás van.
    - dynamic(megoldás/1).
- ```
% findall1(Minta, Cél, L): Cél összes megoldására Minta-k listája L.
findall1(Minta, Cél, _MegoDL) :-
    call(Cél), asserta(megoldás(Minta)), fail.    % fordított sorrend!
findall1(_Minta, _Cél, MegoDL) :-
    megoldás_lista([], MegoDL).
```
- A megoldás/1 tényállításokban tárolt kifejezések fordított listája L-L0.*
- ```
megoldás_lista(L0, L) :-
 retract(megoldás(M)), !, megoldás_lista([M|L0], L).
megoldás_lista(L, L).
```
- ?- findall1(Y, (member(X, [1,2,3]), Y is X\*X), ML).  $\implies$  ML = [1,4,9]

## Klóz lekérdezése: `clause/2`

- `clause(:@Fej, ?Törzs)`
- A Fej alapján megállapítja a predikátum funktorát.
- Az adott predikátum klózeit sorra megpróbálja illeszteni a Fej :- Törzs kifejezéssel (tényállítás esetén Törzs = true).
- Ha az illesztés sikerült, akkor sikeresen left.
- Visszalépés esetén folytatja a keresést (illeszt, sikerül, stb.)

### Példa:

```
:- listing(p), clause(p(2, 0), T).
```

|                         |                           |
|-------------------------|---------------------------|
| <code>p(2, 0).</code>   | <code>T = true ? ;</code> |
| <code>p(1, A) :-</code> | <code>T = r(0) ? ;</code> |
| <code>q(A).</code>      |                           |
| <code>p(2, A) :-</code> | <code>no</code>           |
| <code>r(A).</code>      |                           |

## A clause eljárás alkalmazása: egyszerű nyomkövető interpreter

---

```
% interp(G, D): A G cél futását D bekezdésű nyomkövetéssel mutatja.
interp(true, _) :- !.
interp((G1, G2), D) :- !,
 interp(G1, D), interp(G2, D).
interp(G, D) :-
 (trace(G, D, call)
 ; trace(G, D, fail) % követi a fail kaput, tovább-hiúsul
),
 D2 is D+2, clause(G, B), interp(B, D2),
 (trace(G, D, exit)
 ; trace(G, D, redo), fail % követi a redo kaput, tovább-hiúsul
).
```

*% A G cél áthaladását a Port kapun D bekezdésű nyomkövetéssel mutatja.*

```
trace(G, D, Port) :-
```

```
 /*D szöközt ír ki:*/ tab(D), write(Port), write(' '), write(G), nl.
```

## Nyomkövető interpreter - példafutás

|                               |                                               |
|-------------------------------|-----------------------------------------------|
| - dynamic app/3, app/4.       | ?- interp(app( _, [b,c] , L, [c,b,c,b] ), 0). |
| app( [], L, L).               | call: app( _203, [b,c] , _253, [c,b,c,b] )    |
| app( [X L1] , L2, [X L3] ) :- | call: app( _203, _666, [c,b,c,b] )            |
| app( L1, L2, L3).             | exit: app( [], [c,b,c,b] , [c,b,c,b] )        |
| app( L1, L2, L3, L123) :-     | call: app( [b,c] , _253, [c,b,c,b] )          |
| app( L1, L23, L123),          | fail: app( [b,c] , _253, [c,b,c,b] )          |
| app( L2, L3, L23).            | redo: app( [], [c,b,c,b] , [c,b,c,b] )        |
|                               | call: app( _873, _666, [b,c,b] )              |
|                               | exit: app( [], [b,c,b] , [b,c,b] )            |
|                               | exit: app( [c] , [b,c,b] , [c,b,c,b] )        |
|                               | call: app( [b,c] , _253, [b,c,b] )            |
|                               | call: app( [c] , _253, [c,b] )                |
|                               | call: app( [], _253, [b] )                    |
|                               | exit: app( [], [b] )                          |
|                               | exit: app( [c] , [b] , [c,b] )                |
|                               | exit: app( [b,c] , [b] , [b,c,b] )            |
|                               | exit: app( [c] , [b,c] , [b] , [c,b,c,b] )    |
|                               | L = [b] ?                                     |

# „HAGYOMÁNYOS” BEÉPÍTETT ELJÁRÁSOK



# Aritmetikai beépített eljárások

- $X$  is  $Kif$ :  $Kif$  aritmetikai kifejezés kell legyen, értékét egyesíti  $X$ -szel.
- $Kif1 \ \rho \ Kif2$ :  $Kif1$  és  $Kif2$  aritmetikai kifejezések kell legyenek, értékeik között elvégzi a  $\rho$  összehasonlítást ( $\rho$  lehet =,  $=\backslash$  =,  $<$ ,  $=<$ ,  $>$ ,  $>=$ ).
- Aritmetikai kifejezésekben felhasználható funktorok:

| Infix operátorok   |                    |                               |  |
|--------------------|--------------------|-------------------------------|--|
| + összeadás        | // egész osztás    | \ / bitenkénti és             |  |
| - kivonás          | ** hatványozás     | \ / bitenkénti vagy           |  |
| * szorzás          | mod modulus képzés | << bitenkénti balra léptetés  |  |
| / osztás           | rem maradék képzés | >> bitenkénti jobbra léptetés |  |
| Prefix operátorok: | - negáció          | \ bitenkénti negáció          |  |

| Függvény jelölésűek |                         |             |            |
|---------------------|-------------------------|-------------|------------|
| abs/1               | exp/1                   | floor/1     | sign/1     |
| atan/1              | float/1                 | log/1       | sin/1      |
| ceiling/1           | float_fractional_part/1 | max/2,min/2 | sqrt/1     |
| cos/1               | float_integer_part/1    | round/1     | truncate/1 |



## Listakezelő beépített eljárások

- Lista hossza: `length(?L, ?N)`
- Jelentése: az L lista hossza N.
- `length(-L, +N)` módban adott hosszúságú, csupa különböző változóból álló listát hoz létre.
- `length(-L, -N)` módban rendre felsorolja a 0, 1, ... hosszú listákat.
- Megvalósítását lásd korábban.
- Lista rendezése: `sort(@L, ?S)`
- Jelentése: az L lista @< szerinti rendezése S, (`=`/`/`2 szerint azonos elemek ismétlődését kiszűrve).
- Lista kulcs szerinti rendezése: `keysort(@L, ?S)`
- Az L argumentum Kulcs-Érték alakú kifejezések listája.
- Az eljárás jelentése: az S lista az L lista Kulcs értékei szerinti szabványos (@< általi) rendezése, ismétlődéseket nem szűr.

## Kifejezések kiírása

- `write(@X)`: Kiírja X-et, ha szükséges operátorokat, zárójeleket használva.
- `writetq(@X)`: Mint `write(X)`, csak gondoskodik, hogy szükség esetén az atomok idézőjelek közé legyenek téve.
- `write_canonical(@X)`: Mint `writetq(X)`, csak operátorok nélkül, minden struktúra szabványos alakban jelenik meg.
- `write_term(@X, +Opciók)`: Az Opciók opciólista szerint kiírja X-et.
- `format(@Formatum, @Adatlista)`: A Formatum-nak megfelelő módon kiírja Adatlista-t. A formázójelek alakja:  $\langle \text{szám esetleg} \rangle \langle \text{formázójel} \rangle$ .

```
| ?- write('Helló világ'). \Rightarrow Helló világ
| ?- writetq('Helló világ'). \Rightarrow 'Helló világ'
| ?- write_canonical('*' - '%'). \Rightarrow -(*, '%')
| ?- write_canonical([1,2]). \Rightarrow ', '(1, ', '(2, [])
| ?- write_term([1,2,3], [max_depth(2)]). \Rightarrow [1,2|...]
| ?- format('X=~s --- ~3d s', [0'j,0'ó],3245]). \Rightarrow X=jó --- 3.245 s
```

## Kifejezések kiírása — felhasználó vezérelte formázás

- `print(@X)`: Alapértelmezésben azonos `writeln`-tal. Ha a felhasználó definiál egy `portray/1` eljárást, akkor a rendszer minden a `print`-tel kinyomtatandó rész kifejezésre meghívja `portray`-t. Ennek sikere esetén feltételezi, hogy a kiírás megtörtént, meghívás esetén maga írja ki a rész kifejezést.

A rendszer a `print` eljárást használja a változó-behelyettesítések és a nyomkövetés kiírására!

- `portray(@Kif)` (felhasználó által definiálandó ún. *kampó eljárás*): Igaz, ha `Kif` kifejezést a Prolog rendszernek *nem* kell kiírnia (és ekkor maga a `portray` kell, hogy elvégezze a kiírást).

- Példa:

```
portray(Matrix) :-
 Matrix = [[_|_|_|_|_|],
 (member(Row, Matrix),
 nl, print(Row), fail
 ;
 true
)
].
```

|       |                               |
|-------|-------------------------------|
| X =   | ?- X = [[1,2], [3,4], [5,6]]. |
| [1,2] |                               |
| [3,4] |                               |
| [5,6] | ?                             |

## Karakterek kiírása és beolvasása

---

- `put_code(+Kód):` Kiírja az adott kódú karaktert.
- `tab(+N):` Kiír  $N$  szóközt feltéve, hogy  $N > 0$ .
- `n1:` Kiír egy soremelést.
- `get_code(?Kód):` Beolvas egy karaktert és (karakterkódját) egyesíti Kód-dal. (File végénél  $Kód = -1$ .)
- `peek_code(?Kód):` A soronkövetkező karakter kódját egyesíti Kód-dal. A karaktert nem távolítja el a bemenetről. (File végénél  $Kód = -1$ .)

- **Példa:**

```
% L a következő sor karakterkódjainak listája.
rd_line(L) :- peek_code(0'\n'), !, get_code(_), L = [].
rd_line([C|L]) :- get_code(C), rd_line(L).

| ?- rd_line(L), tab(20), member(X, L), put_code(X), tab(1), fail ; nl.
|: Hello world!
```

```
 H e l l o w o r l d !
```

## Példa: számbeolvasás

```

/ számbe(Szám): a Szám szám következik az input-folyamban.
számbe(Szám) :-
 számjegy(Érték), számbe(Érték, Szám).

/ Az eddig beolvasott Szám0-val együtt az input-folyamban következő
/ szám értéke Szám.
számbe(Szám0, Szám) :-
 számjegy(E), !, Szám1 is Szám0*10+E, számbe(Szám1, Szám).
számbe(Szám, Szám).

/ Érték értékű számjegy következik.
számjegy(Érték) :-
 peek_code(Kar), Kar >= 0'0, Kar =< 0'9, get_code(_),
 Érték is Kar - 0'0.

?- számbe(X), get_code(_), számbe(Y).
: 123 456
 => X = 123, Y = 456

```

## Kifejezések beolvasása

- `read(?Kif)`: Beolvas egy ponttal lezárt kifejezést és egyesíti `Kif`-fel. (File végénél `Kif = end_of_file.`)
- `read_term(?Kif, +Opciók)`: Mint `read/1`, de az Opciók opciólistát is figyelembe veszi.

- Példa — botcsinálta programbeolvasó:

|                                                                                                                                                                                                   |                                                                 |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------|
| <pre>consult_body :-     repeat, read(Term),         (   Term = end_of_file -&gt; true         ;   assertz(Term), fail         ), !.    ?- consult_body.   : p(X) :- q(X), r(X).   : ~D yes</pre> | <pre>  ?- listing([p/1]). p(A) :-     q(A),     r(A). yes</pre> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------|

## Be- és kiviteli csatornák

- Csatornák megnyitása és kezelése:
  - `open(@Filename, @Mód, -Csatorna):` Megnyitja a `Filename` nevű állományt `Mód` módban (`read`, `write` vagy `append`). A `Csatorna` argumentumban visszaadja a megnyitott csatorna „nyelét”.
  - `set_input(@Csatorna)`, `set_output(@Csatorna):` Az ezt követő beviteli/kiviteli eljárások `Csatorna`-t használják majd (jelenlegi csatorna).
  - `current_input(?Csatorna)`, `current_output(?Csatorna):` A jelenlegi beviteli/kiviteli csatornát egyesíti `Csatorna`-val.
  - `close(@Csatorna):` Lezárja a `Csatorna` csatornát.
- Explicit csatornamegadás be- és kiviteli eljárásokban
  - Az eddig ismertett összes be- és kiviteli eljárásnak van egy eggyel több argumentumú változata, amelynek első argumentuma a csatorna. Ezek:  
`write/2`, `writeln/2`, `write_canonical/2`, `write_term/3`, `print/2`, `read/2`, `read_term/3`, `format/3`, `put_code/2`, `tab/2`, `nl/1`, `get_code/2`, `peek_code/2`.

## Egy egyszerűbb be- és kiviteli szervezés: DEC10 I/O

- `see(@Filenév), tell(@Filenév)`: Megnyitja a Filenév file-t olvasásra/írásra és a jelenlegi csatornává teszi. Újabb híváskor csak a jelenlegi csatornává teszi.
- `seeing(?Filenév), telling(?Filenév)`: A jelenlegi beviteli/kiviteli csatorna állománynévét egyesíti Filenév-vel.
- `seen, told`: Lezárja a jelenlegi beviteli/kiviteli csatornát.
- **Példák** — nagyon egyszerű `consult` variánsok:

|                                                                                                                                                                                        |                                                                                                                                                                                   |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>consult_decl0_style(File) :-     seeing(Old), see(File),     repeat, read(Term),     (   Term = end_of_file     -&gt;  seen     ;   assertz(Term), fail     ), !, see(Old).</pre> | <pre>consult_with_streams(File) :-     open(File, read, S),     repeat, read(S, Term),     (   Term = end_of_file     -&gt;  close(S)     ;   assertz(Term), fail     ), !.</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|



## Hibakezelési beépített eljárások

---

- Hibahelyzetet beépített eljárás rossz argumentumokkal való meghívása, vagy a `throw/1` (`raise_exception/1`) eljárás válthat ki.
- Minden hibahelyzetet egy Prolog kifejezés (ún. hiba-kifejezés) jellemez.
- Hiba „dobása”: `throw(@HibaKif), raise_exception(@HibaKif)`
- Hatása: Kiváltja a `HibaKif` hibahelyzetet.
- Hiba „elkapása”: `catch(:+Cél, ?Minta, :+Hibaág), on_exception(?Minta, :+Cél, :+Hibaág)`
- Hatása: Futtatja a `Cél` hívást.
  - Ha `Cél` végrehajtása során hibahelyzet nem fordul elő, futása azonos `Cél`-al.
  - Ha `Cél`-ban hiba van, a hiba-kifejezést egyesíti `Mintá`-val.
  - Ha ez sikeres, meghívja a `Hibaág`-at.
  - Ellenkező esetben továbbdobja a hiba-kifejezést, hogy a további körülvető `catch` eljárások esetleg elkaphassák azt.

## Programfejlesztési beépített eljárások (SICStus specifikusak)

---

- `set_prolog_flag(+Jelző, @Érték):` Jelző értékét Érték-re állítja.
- `current_prolog_flag(?Jelző, ?Érték):` Jelző pillanatnyi értéke Érték.
- Néhány fontos Prolog jelző:
  - `language:` végrehajtási mód (`sicstus`, `iso`).
  - `argv:` csak olvasható, a parancssorbeli argumentumok listája.
  - `unknown:` viselkedés definiálatlan eljárás hívásakor (`trace`, `fail`, `error`).
  - `source_info:` forrásszintű nyomkövetés (`on`, `off`, `emacs`).
- `consult(:@Files), [:@File,...]:` Betölti a `File(ok)`at, interpretált alakban.
- `compile(:@File):` Betölti a `File(ok)`at, lefordított alakot hozva létre.
- `listing:` Kiírja az összes interpretált eljárást az aktuális kimenetre.
- `listing(:@EljárásSpec):` Kiírja a megnevezett interpretált eljárásokat.
- Itt és később: `EljárásSpec` — név vagy funktor, esetleg modul-kvalifikációval ellátva, ill. ezek listája, pl. `listing(p)`, `listing([m:q,p/1])`.

## Programfejlesztési eljárások (folytatás)

---

- `statistics`: Különféle statisztikákat ír ki az aktuális kimenetre.
- `statistics(?Fajta, ?Érték)`: Érték a Fajta fájtájú mennyiség értéke.
- **Példa:** `statistics(runtime, E)  $\implies$  E=[Tdiff, T]`, Tdiff az előző lekérdezés óta, T a rendszerindítás óta eltelt idő, ezredmásodpercben.
- `break`: Egy új interakciós szintet hoz létre.
- `abort`, `halt`: Kilép a legkülső interakciós szintre ill. a Prolog rendszerből.
- `trace`: Elindítja az interaktív nyomkövetést.
- `debug, zip`: Elindítja a szelektív nyomkövetést, csak spion-pontoknál áll meg. (A zip mód gyorsabb, de nem gyűjt annyi információt mint a debug mód.)
- `nodebug, notrace, nozip`: Leállítja a nyomkövetést.
- `spy(:@EljárásSpec)`: Spion-pontot tesz a megadott eljárásokra.
- `nospy(:@EljárásSpec)`: Megszünteti a megadott spion-pontokat.
- `nospyall`: Az összes spion-pontot megszünteti.

# NYELVTANI ELEMZÉS PROLOGBAN



## Egy egyszerű nyelvtani elemzési példa

- Bináris számok nyelvtana
 
$$\begin{aligned} \langle \text{szám} \rangle &::= & \langle \text{számjegy} \rangle \langle \text{számmaradék} \rangle \\ \langle \text{számmaradék} \rangle &::= & \langle \text{számjegy} \rangle \langle \text{számmaradék} \rangle \mid \epsilon \\ \langle \text{számjegy} \rangle &::= & 0 \mid 1 \end{aligned}$$
- Ugyanez DCG (Definite Clause Grammar) jelöléssel:
 
$$\begin{array}{ll} \text{szám} \text{ -->} & \text{számjegy, számmaradék.} \\ \text{számmaradék} \text{ -->} & \text{számjegy, számmaradék} \mid \text{" "}. \\ \text{számjegy} \text{ -->} & \text{"0"} \mid \text{"1"}. \end{array}$$
- A definit klóz nyelvtan (DCG):
  - egy általános nyelvtani formalizmus,
  - amely egyszerűen Prologra fordítható,
  - a legtöbb Prolog rendszer része (bár a szabványnak nem).

## Nyelvtani elemzés „bevetítése” Prologba

- Nyelvtani elemzés: annak eldöntése, hogy egy (Prolog listában tárolt) jelsorozat megfelel-e egy adott nem-terminális nyelvtani fogalomnak.
- A lista lehet karakterkódok listája, lexikai elemek (token-ek) listája stb.
- Egy nem-terminálisnak egy kétargumentumú Prolog szabály felel meg:

```
szám --> számjegy, számmaradék.

szám(L0, L) :- számjegy(L0, L1), számmaradék(L1, L).

% Az L0 kódlistáról "leelemezhető" egy <szám>, marad L ha
% L0-ról leelemezhető egy <számjegy>, marad L1, és
% L1-ről leelemezhető egy <számmaradék>, marad L.
```

- Általánosan: az adott nem-terminálisnak megfelelő jelsorozatot „leelemezve” (lehagyva) egy L0 lista elejéről marad egy L lista.
- Terminális szimbólumok esetén egyetlen elemet kell lehagyni a listáról, erre szolgál a ’C’/3 beépített eljárás. Definíciója: ’C’(L0, X, L)  $\equiv$  L0 = [X|L] (A SICStus fordító a két hívást pontosan ugyanúgy fordítja).

## A DCG szabályok lefordított alakja

- A DCG mint szintaktikai édesítőszers — példa:

```
szám --> számjegy, számmaradék. % A | B ≡ A ; B
számmaradék --> számjegy, számmaradék | "". % "" ≡ []
számjegy --> "0" | "1". % "0" ≡ [48]
```

- A fenti DCG szabályok betöltésekor a következő Prolog kód keletkezik:

```
szám(L0, L) :- számjegy(L0, L1), számmaradék(L1, L).
számmaradék(L0, L) :- (számjegy(L0, L1), számmaradék(L1, L)
 ; L = L0
).
számjegy(L0, L) :- ('C'(L0, 48, L) ; 'C'(L0, 49, L)).
```

- A DCG elemző futtatása:

```
| ?- szám("101", []). => yes % "101" == [0'1,0'0,0'1]
| ?- szám("102", L). => L = [0'2] ; L = [0'0,0'2] ; no
```

## Vezérlési szerkezetek DCG szabályokban

- DCG szabályokban használható: vágó, diszjunktció, negáció és feltételes diszjunktív szerkezet.
- Ezek változtatás nélkül átkerülnek a Prolog alakba. Példák:

```
% Leelemezhető számjegyek egy MAXIMÁLIS (esetleg üres) listája.
számmaradék --> (számjegy -> számmaradék.
 ; []
).
```

```
% Ugyanez vágóval
számmaradék --> számjegy, !, számmaradék.
számmaradék --> [] .
```

```
% Az utóbbi Prolog alakja:
számmaradék(L0, L) :- számjegy(L0, L1), !, számmaradék(L1, L).
számmaradék(L0, L) :- L = L0.
```

```
| ?- számmaradék("102", L). => L = [0'2] ; no
```



## Prolog hívás beillesztése DCG szabályba

- Általánosabb példa: decimális számjegyek elemzése

```
számjegy --> "0" ; "1" ; "2" ; "3" ; "4" ;
 "5" ; "6" ; "7" ; "8" ; "9" .
```

*% Ugyanez általánosabban és egyszerűbben:*

```
számjegy --> [K], % K a következő terminális
 {decimális_jegy_kódja(K)}.
```

*% K egy számjegy kódja.*

```
decimális_jegy_kódja(K):- K >= 0'0, K =< 0'9.
```

- A fenti DCG szabály Prolog megfelelője:

*% Leelemezhető egy számjegy kódja.*

```
számjegy(L0, L) :-
```

```
 'C'(L0, K, L), % K a következő kód
 decimális_jegy_kódja(K). % megfelelő-e a K?
```

## Az elemző kiegészítése argumentumokkal

- Egy DCG szabály argumentumaiban egy „belső” alakot építhet:

```
% leelemezhető egy Sz értékű decimális számjegy-sorozat
szám(Sz) --> számjegy(J), számmaradék(J, Sz).
```

```
% leelemezhető számjegyek egy esetleg üres listája, amelynek
% az eddig leelemzett Sz0-val együtt vett értéke Sz.
számmaradék(Sz0, Sz) -->
```

```
 számjegy(J), !, {Sz1 is Sz0*10+J}, számmaradék(Sz1, Sz).
```

```
számmaradék(Sz0, Sz0) --> [].
```

```
% leelemezhető egy J értékű számjegy.
```

```
számjegy(J) --> [K], {decimális_jegy_kódja(K), J is K-0'0}.
```

```
| ?- szám(Sz, "102 56", L). \implies L = " 56", Sz = 102; no
```

- A számmaradék DCG szabály Prolog alakja:

```
számmaradék(Sz0, Sz, L0,L) :-
```

```
 számjegy(J, L0,L1), !, Sz1 is Sz0*10+J, számmaradék(Sz1, Sz, L1,L).
```

```
számmaradék(Sz0, Sz0, L0,L) :- L=L0.
```

## A DCG nyelvtani szabályok szerkezete — összefoglalás

- A DCG szabály alakja:  $\langle \textit{Baloldal} \rangle \rightarrow \langle \textit{Jobboldal} \rangle$ .
- $\langle \textit{Baloldal} \rangle$ : egy nem-terminális(, amit esetleg terminálisok listája követ).
- $\langle \textit{Jobboldal} \rangle$ : konjunkció (,), diszjunkció (;), ha-akkor ( $\rightarrow$ ) és negáció ( $\backslash +$ ) segítségével épül terminálisokból, nem-terminálisokból és Prolog hívásokból.
- Nem-terminális: *tetszőleges hívható* kifejezés (atom vagy struktúra).
- Terminális: *tetszőleges* Prolog kifejezés; 0, 1 vagy több terminális jel sorozata *listaként* helyezhető el a DCG szabályokban.
- Prolog hívás:  $\{ \}$  zárójelekbe zárva helyezhető el (vágó köré nem kell zárójel).

- DCG = egy „ingyen” akkumulátor (akkum. lépés: ’C’, egy elem levétele):

$$p(A, \dots) \rightarrow$$

$$q_0(B, \dots), \dots, [X], q_i(C, \dots), \dots, \{\text{Cél}\}, \dots, q_n(D, \dots).$$

$$p(A, \dots, L_0, L) :-$$

$$q_0(B, \dots, L_0, L_1), \dots, 'C'(L_{i-1}, X, L_i), q_i(C, \dots, L_i, L_{i+1}), \dots,$$

$$\text{Cél}, \dots, q_n(D, \dots, L_n, L).$$

## OCG példa: kifejezés kiértékelése

```

% kif(Z, L0, L): L0 elején egy Z értékű aritm. kifejezés áll, marad L.
kif(Z) --> tag(X), "+", kif(Y), {Z is X + Y}.
kif(Z) --> tag(X), "-", kif(Y), {Z is X - Y}.
kif(X) --> tag(X).

% tag(Z, L0, L): L0-ból leelvezhető egy Z értékű tag, marad L.
ag(Z) --> szám(X), "*", tag(Y), {Z is X * Y}.
ag(Z) --> szám(X), "/", tag(Y), {Z is X / Y}.
ag(X) --> szám(X).

?- kif(Z, "10*10-6*6", ""). => Z = 64 ; no
?- kif(Z, "10*10-6*6", L). => L = [], Z = 64 ;
 L = [42,54], Z = 94 ; ...
?- kif(Z, "4-2+1", []). => Z = 1 Jobbról balra elemez!

% Egy lehetséges javítás:
kif(Z) --> tag(X), kifmaradék(X, Z).

kifmaradék(X0, Z) --> "+", tag(X1), {X is X0 + X1}, kifmaradék(X, Z).
. . .

```

## OCG példa: „természetes” nyelvű beszélgetés

```

- use_module(library(lists)).

mondat(Alany, Áll, L0, L): L0-L kielmezhető egy Alany alanyból és Áll
állítmányból álló mondatra. Alany lehet első vagy második személyű
névűs, vagy egyetlen szóból álló (harmadik személyű) alany.
mondat(Alany, Áll) --> {én_te(Alany, Ige)}, én_te_perm(Alany, Ige, Áll).
mondat(Alany, Áll) --> szó(Alany), szavak(Áll).

én_te(Alany, Ige):
 Az Alany első/második személyű névűsnek megfelelő léti Ige.
 én_te("én", "vagyok").

 én_te("te", "vagy").

én_te_perm(Ki, Ige, Áll, L0, L): L0-L kielmezhető egy Ki
névűsből, Ige igealakból és Áll állítmányból álló mondatra.
 én_te_perm(Alany, Ige, Áll) --> szó(Alany), szavak(Áll).
 én_te_perm(Alany, Ige, Áll) --> szó(Alany), szavak(Áll), szó(Ige).
 én_te_perm(Alany, Ige, Áll) --> szavak(Áll), szó(Ige), szó(Alany).
 én_te_perm(_Alany, Ige, Áll) --> szavak(Áll), szó(Ige).

```

## Példa: ‘természetes’ nyelvű beszélgetés — szavak elemzése

```

% szó(Sz, L0, L): L0-L egy Sz betűsorozatból álló (nem üres) szó.
% szó(Sz) --> betű(B), szómaradék(SzM), {illik([B|SzM], Sz)}, köz.
% szómaradék(Sz, L0, L): L0-L egy Sz kódlistából álló (esetleg üres) szó.
% szómaradék([B|Sz]) --> betű(B), !, szómaradék(Sz).
% szómaradék([]) --> [].

% illik(Szó0, Szó): Szó0 = Szó, vagy a kezdő kis-nagy betűben különbözik.
% illik([B0|L], [B|L]) :- (B = B0 -> true ; abs(B-B0) =:= 32).

% köz(L0, L): L0-L nulla, egy vagy több szóköz.
% köz --> (" " -> köz ; "").

% betű(K, L0, L): L0-L egy K kódú "betű" (különbözik a " . ?" jelektől)
% betű(K) --> [K], {non_member(K, " . ?")}.

% szavak(SzL, L0, L): L0-L egy SzL szó-lista.
% szavak([Sz|Szk]) --> szó(Sz), (szavak(Szk)
% ; {Szk = []}
%).

```

## Példa: ‘természetes’ nyelvű beszélgetés — párbeszéd-szervezés

```
% :- type mondás ---> kérdez(szó) ; kijelent(szó, list(szó)) ; un.
% Megvalósít egy párbeszédet.
párbeszéd :-
 repeat, rd_line(L),
 (
 menet(Mondás, L, []) -> feldolgoz(Mondás)
 ; write('Nem értem\n'), fail
),
 Mondás = un, !.

% menet(Mondás, L0, L): Az L0-L kiellenzett alakja Mondás.
menet(kérdez(Alany)) --> {kérdő(Szó)}, mondat(Alany, [Szó]), "?"
menet(kijelent(Alany, Áll)) --> mondat(Alany, Áll), ". "
menet(un) --> szó("unlak"), ". "

% kérdő(Szó): Szó egy kérdőszó.
kérdő("mi"). kérdő("ki"). kérdő("kicsoda").
```

## Példa: ‘természetes’ nyelvű beszélgetés — válaszok előállítás

- dynamic tudom/2.

```
% feldolgoz(Mondás): feldolgozza a felhasználótól érkező Mondás üzenetet.
eldolgoz(un) :- write('Én is.\n').
eldolgoz(kijelent(Alany, Áll)) :-
 assertz(tudom(Alany,Áll)), write('Felfogtam.\n').
eldolgoz(kérdez(Alany)) :-
 tudom(Alany, _), !, válasz(Alany).
eldolgoz(kérdez(_)) :-
 write('Nem tudom.\n').
```

*% Felsorolja az Alany ismert tulajdonságait.*

```
álassz(Alany) :- tudom(Alany, Áll),
 (member(Szó, Áll), format('~s ', [Szó]), fail
 ; nl, fail
).
álassz(_).
```



## Beszélgetős DCG példa — egy párbeszéd

|                                           |                                                              |
|-------------------------------------------|--------------------------------------------------------------|
| ?- párbeszéd.                             | : Én vagyok Jeromos.                                         |
| : Magyar legény vagyok én.<br>:felfogtam. | Felfogtam.<br>  : Te egy Prolog program vagy.<br>:felfogtam. |
| : Ki vagyok én?<br>: Magyar legény        | : Ki vagyok én?<br>Magyar legény                             |
| : Péter kicsoda?<br>: Nem tudom.          | Boldog<br>Jeromos                                            |
| : Péter tanuló.<br>:felfogtam.            | : Okos vagy.<br>Felfogtam.                                   |
| : Péter jó tanuló.<br>:felfogtam.         | : Ki vagy te?<br>egy Prolog program                          |
| : Péter kicsoda?<br>: annuló              | Okos<br>  : Valóban?                                         |
| : Jó tanuló                               | Nem értem                                                    |
| : Boldog vagyok.<br>:felfogtam.           | : Unlak.<br>Én is.                                           |

## A DCG formalizmus felhasználása elemzésen kívül

- DCG szabályok kényelmesen használhatók általános akkumulálásra

- Listák akkumulálása (nemcsak elemzés, építés is:)

```
% anbn(+N, ?L): Az L lista N db a-ból és azt követő N db b-ből áll.
anbn(N, L) :- anbn(N, L, []).
```

```
% anbn(N, L0, L): L0-L N db a-ból és azt követő N db b-ből áll.
anbn(0) --> !.
```

```
anbn(N) --> {N > 0, N1 is N-1}, [a], anbn(N1), [b].
```

```
% a fenti DCG szabály kifejtve:
```

```
anbn(N, L0, L) :-
```

```
 N > 0, N1 is N-1, L0=[a|L1], anbn(N1, L1, L2), L2=[b|L].
```

- Egyébként az elemi akkumulálási lépést DCG-n kívül kell megírni:

|                                                                                                       |                                                                                             |
|-------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| <pre>% sum(L, S0, S): L összege S-S0. sum([]) --&gt; []. sum([X L]) --&gt;     plus(X), sum(L).</pre> | <pre>% L számlista összege S. sum(L, S) :- sum(L, 0, S). plus(X, S0, S) :- S is S0+X.</pre> |
|-------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|

# FEJLETTEBB NYELVI ÉS RENDSZERELEMEK



## Külső nyelvi interfész

---

- Hagymányos (pl. C nyelvű) programrészek meghívásának módja:
- A Prolog rendszer elvégzi az átalakítást a Prolog alak és a külső nyelvi alak között. Kényelmesebb, biztonságosabb mint a másik módszer, de kevésbé hatékony. Többnyire csak egyszerű adatokra (egész, valós, atom). (MProlog)
- A külső nyelvi rutin pontereket kap Prolog adatstruktúrákra, valamint hozzáférési algoritmusokat ezek kezelésére. Nehézkesebb, veszélyesebb, de jóval hatékonyabb mint az előző megoldás. Összetett adatok adásvételére is jó. (SWI, SICStus)

## Külső nyelvi interfész — példa

---

Prologban az `index_keys(Spec, Kif, Kulcs, Szám)` eljárást szeretnénk meghívni, aminek a jelentése:

- Ha *Spec* és *Kif* különböző funktorú kifejezések, akkor *Szám* = -1 és *Kulcs* = [].
- Egyébként, ha *Spec* valamelyik argumentuma + és *Kif* megfelelő argumentuma változó, akkor *Szám* = -2 és *Kulcs* = [].
- Egyébként *Szám* a *Spec* argumentumaként előforduló + atomok száma, *Kulcs* pedig *Kif* megfelelő argumentumok *kivonatából* képzett lista. A kivonat lényegében az argumentum funktora, azzal az eltéréssel, hogy a konstansok kivonata maga a konstans, struktúrák esetén pedig a struktúra neve és az aritása külön elemként kerül a kivonat-listába.

## Külső nyelvi interfész — példa

- A példaeljárás használata

```
| ?- [ixtest].
| ?- index_keys(f(+, -, +, +),
 f(12.3, -, s(1, -, z(2)), t),
 L, X).
L = [12.3,s,3,t], X = 3 ?
yes
```

- Az ixtest.pl file.

```
foreign(ixkeys, index_keys(+term, +term, -term, [-integer])).
foreign_resource(ixkeys, [ixkeys]).
:- load_foreign_resource(ixkeys).
```

- A C programot elő kell készíteni a Prolog számára az splfr eszköz segítségével:

```
splfr ixkeys ixtest.pl +c ixkeys.c
```

## Külső nyelvi interfész — a C kód (ixkeys.c állomány)

```

include <sicstus/sicstus.h>

define NA -1 /* not applicable */
define NI -2 /* instantiatedness */

ong ixkeys(SP_term_ref spec,
 SP_term_ref term, SP_term_ref list)
 unsigned long sname, tname, plus;
 int sarity, tarity, i;
 long ret = 0;
 SP_term_ref arg = SP_new_term_ref(),
 tmp = SP_new_term_ref();
 SP_get_functor(spec, &sname, &sarity);
 SP_get_functor(term, &tname, &tarity);
 if (sname != tname || sarity != tarity)
 return NA;

 plus = SP_atom_from_string("+");

 for (i = sarity; i > 0; --i) {
 unsigned long t;
 SP_get_arg(i, spec, arg);
 SP_get_atom(arg, &t); /* no check */
 if (t != plus) continue;

 SP_get_arg(i, term, arg);
 switch (SP_term_type(arg)) {
 case SP_TYPE_VARIABLE:
 return NI;
 case SP_TYPE_COMPOUND:
 SP_get_functor(arg, &tname, &tarity);
 SP_put_integer(tmp, (long)tarity);
 SP_cons_list(list, tmp, list);
 SP_put_atom(arg, tname);
 break;
 }
 SP_cons_list(list, arg, list); ++ret;
 }
 return ret;
}

```

## Hasznos lehetőségek SICStus Prolog-ban

---

- Tetszőleges nagyságú egész számok

pl.:

| ?- fakt(40,F).

F = 815915283247897734345611269596115894272000000000 ?

- Globális változók (Blackboard)

bb\_put(Kulcs,Érték)

A Kulcs kulcs alatt eltárolja Érték-et, az előző értéket, ha van, törölve.  
(Kulcs egy (kis) egész szám vagy atom lehet.)

bb\_get(Kulcs,Érték)

Előhívja Érték-be a Kulcs értékét.

bb\_delete(Kulcs,Érték)

Előhívja Érték-be a Kulcs értékét, majd kitörli.



## Hasznos lehetőségek SICStus Prolog-ban *(folytatás)*

---

- Visszaléptethető módon változtatható kifejezések

```
create_mutable(Adat, ValttKif)
```

Adat kezdőértékkel létrehoz egy új változtatható kifejezést, ez lesz ValttKif. Adat nem lehet üres változó.

```
get_mutable(Adat, ValttKif)
```

Adat-ba előveszi ValttKif pillanatnyi értékét.

```
update_mutable(Adat, ValttKif)
```

A ValttKif változtatható kifejezés új értéke Adat lesz. Ez a változtatás visszalépéskor visszacsinálódik. Adat nem lehet üres változó.

- Takarító eljárás

```
call_cleanup(Hivas, Tiszito)
```

Meghívja call(Hivas)-t és ha az véglegesen befejezte futását, meghívja Tiszito-t. Egy eljárás akkor fejezte be véglegesen a futását, ha további alternatívák nélkül sikerült, meghíúsult vagy kivételt dobott.

## Fejlett vezérlési lehetőségek SICStusban: Blokk-deklarációk

---

### ● Példa:

```
:- block p(-, ?, -, ?, ?).
```

Jelentése: ha az első és a harmadik argumentum is behelyettesíthetően változó (blokkolási feltétel), akkor a `p` hívás felfüggesztődik.

Ugyanarra az eljárásra több vagylagos feltétel is szerepelhet, pl.

```
:- block p(-, ?), p(?, -).
```

### ● Végtelen választási pontok kiküszöbölése blokk-deklarációval

```
:- block append(-, ?, -).
```

```
append([], L, L).
```

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

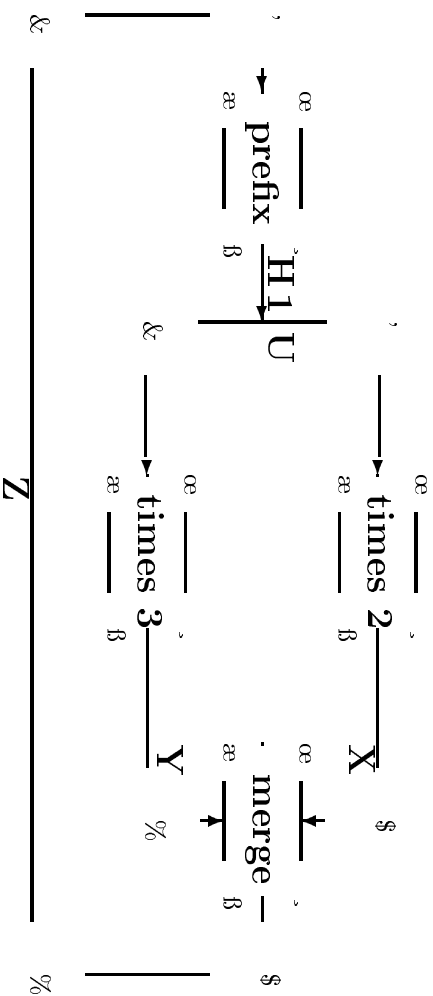
```
append(L1, L2, L3).
```

## Blokk-deklarációk *(folytatás)*

---

- Generál-és-ellenőriz típusú programok gyorsítása
  - általában nem hatékonyak (pl megrajzolja\_1), mert túl sok visszalépést használnak
  - korutinszervezéssel a generáló és ellenőrző rész “automatikusan” összefésülhető
  - ehhez az ellenőrző részt kell előre tenni és megfelelően blokkolni
- Korutinszervezésre épülő programok
  - Példa: egyszerűsített Hamming feladat
    - keressük a  $2^i * 3^j$  ( $i \geq 1, j \geq 1$ ) alakú számok közül az első  $N$  darabot nagyság szerint rendezve.
    - ‘stream-and-parallelism’ közelítésmódot használva korutinszervezéssel egyszerűen lehet megoldani

# Hamming probléma



A H lista az első N, csak a 2 és 3 tényezőkből álló szám.

amming(N, H) :-

U = [1|H], times(U, 2, X), times(U, 3, Y),

merge(X, Y, Z), prefix(N, Z, H).

times(X, M, Z): A Z lista az X elemeinek M-szerese

- block times(-, ?, ?).

times([A|X], M, Z) :- B is M\*A, Z = [B|U], times(X, M, U).

times([], -, []).

## Hamming probléma (folyt.)

---

$\text{merge}(X, Y, Z) : Z$  az  $X$  és  $Y$  összefésülése.

–  $\text{block merge}(-, ?, ?)$ ,  $\text{merge}(?, -, ?)$ .

Csak akkor fusson, ha az első két argumentum ismert

$\text{merge}([A|X], [B|Y], V) :-$

$A < B, !, V = [A|Z], \text{merge}(X, [B|Y], Z).$

$\text{merge}([A|X], [B|Y], V) :-$

$B < A, !, V = [B|Z], \text{merge}([A|X], Y, Z).$

$\text{merge}([A|X], [A|Y], [A|Z]) :-$

$\text{merge}(X, Y, Z).$

$\text{merge}([], X, X) :- !.$

$\text{merge}(-, [], []).$

$\text{prefix}(N, X, Y) : \text{Az } X \text{ lista első } N \text{ eleme } Y.$

$\text{prefix}(0, -, []) :- !.$

$\text{prefix}(N, [A|X], [A|Y]) :-$

$N > 0, N1 \text{ is } N-1, \text{prefix}(N1, X, Y).$

## Korutinszervező eljárások

- `freeze(X, Hivas)`

Hivast felfüggeszti mindaddig, amíg X behelyettesíthetetlen változó.

- `frozen(X, Hivas)`

Az X változó miatt felfüggesztett hívás(oka)t egyesíti Hivas-sal.

- `dif(X, Y)`

X és Y nem egyesíthető. Mindaddig felfüggesztődik, amíg ez el nem dönthető.

- `call_residue(Hivas, Maradék)`

Hivas-t végrehajtja, és ha a sikeres lefutás után maradnak felfüggesztett hívások, akkor azokat visszaadja Maradékban. Pl.

```
| ?- call_residue(dif(X, f(Y)), Maradék).
```

```
 => Maradék = [[X]-(prolog:dif(X, f(Y)))]
```

```
| ?- call_residue((dif(X, f(Y)), X=f(Z)), Maradék).
```

```
 => X = f(Z), Maradék = [[Y,Z]-(prolog:dif(f(Z), f(Y)))]
```

## STCStus könyvtárak

---

- Könyvtár betöltése

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

- A legfontosabb könyvtárak

- arrays Logaritmikus elérési idejű kiterjeszthető tömbök megvalósítását tartalmazza.
- assoc AVL fák segítségével valósítja meg az „asszociációs listák”, azaz véges Prolog kifejezeshalmazokon definiált kiterjeszthető leképezések fogalmát.
- atts tetszőleges attributumokat enged a Prolog változókhoz rendelni, ezeket tárolórekeszként és a Prolog egyesítési mechanizmusának módosítására is engedi használni.
- heaps A bináris kazal (heap) fogalmát valósítja meg, amely főként prioritásos sorok (priority queue) megvalósítására használható.
- lists Biztosítja a listakezelő alapműveleteket.

- **terms** Különböző kifejezéskezelő eljárásokat tartalmaz.
- **ordsets** Halmazműveleteket definiál, ahol a halmazokat a Prolog szabványos rendezése szerint (`compare`) rendezett listákkal ábrázolja.
- **queues** Sorokra (`queue`, FIFO store) vonatkozó műveleteket definiál.
- **random** Egy véletelenszám-generátort tartalmaz.
- **system** Különféle operációsrendszer-szolgáltatások elérését biztosítja.
- **trees** Az arrays könyvtárhoz hasonló, de nem-kiterjeszthető logaritmikus elérési idejű tömbfogalmat valósít meg, bináris fák segítségével (kicsit hatékonyabb mint az arrays könyvtár).
- **ugraphs** Irányított és irányítatlan gráf fogalmat valósít meg, élcímkék nélkül.
- **wgraphs** Olyan irányított és irányítatlan gráf fogalmat valósít meg, ahol minden él egy egészértékű súllyal rendelkezik.
- **sockets** A socket-ek kezelésére szolgáló eljárásokat biztosít.
- **linda/client** és **linda/server** Linda-szerű processzorkommunikációs eszközöket ad.



- bdb Felhasználó által definiált többszörös indexelést lehetővé tevő, Prolog kifejezések lemezen való tárolására szolgáló adatbázis-rendszer.
- clpb Boole-értékekre vonatkozó feltétel-megoldó (constraint solver).
- clpq és clpr Feltétel-megoldó a  $Q$  (racionális számok) ill.  $R$  (valós számok) tartományán.
- clpfd Véges tartományokra vonatkozó feltétel-megoldó (constraint solver).
- tcltk A *Tcl/Tk* nyelv és eszközkészlet elérését biztosítja.
- gauge Prolog programok a profilrozására szolgáló, a tcltk -n alapuló grafikus interfésszel rendelkező eszköz.
- charsio Karktersorozatból olvasó ill. abba író be- és kiviteli eljárások gyűjteménye.
- timeout Lehetőséget ad arra, hogy célok futási idejét korlátozzuk.
- xref A nyomkövetés és a program-analízis segítésére használható keresztreferencia készítő program.

# KÍRÁS, NYOMKÖVETÉS



- `{TextIO.println : string -> unit}`  
`print s = kiírja az s értékét a standard kimenetre, és azonnal kiüríti a puffert.`
- `{Meta.printlnVal : 'a -> 'a}`  
`printVal e = kiírja az e kifejezés értékét a standard kimenetre pontosan úgy, ahogyan az SML értelmező írja ki a „legfelső szinten”, és azonnal kiüríti a puffert. Eredményül visszaadja az e kifejezés értékét. Csak interaktív módban használható.`

### Példák:

|                                                                             |                                                                                              |
|-----------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| <pre>- print("alma"~"Korte\n");   almaKorte   &gt; val it = () : unit</pre> | <pre>- printVal("alma"~"Korte\n");   "almaKorte\n"&gt; val it = "almaKorte\n" : string</pre> |
|-----------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|

*Megjegyzés.* A `k` kapcsolós zárójelek – { és } – között opcionálisan megadható modulnév áll.

Például `{TextIO.println}` azt jelenti, hogy a függvény a `TextIO` modulban van definiálva, de az SML-értelmező a `print` nevet rövid alakban is felismeri.

## Kíírás (folyt.)

---

• `printVal`-lal tetszőleges típusú érték íratható ki. További példák:

```
- printVal (3, 5.0);
(3, 5.0) > val it = (3, 5.0) : int * real

- printVal ["A", "Z", "#"];
["A", "Z", "#"] > val it = ["A", "Z", "#"] : char list

- datatype t = L | B of t * t;
> New type names: =t
datatype t = (t, {con B : t * t -> t, con L : t})
 con B = fn : t * t -> t
 con L = L : t

- val fa = B(B(B(L, B(L, B(B(L, L))), L), B(L, L));
> val fa = B(B(B(L, B(L, B(B(L, L))), L), B(L, L)) : t
- printVal fa;
B(B(B(L, B(L, B(B(L, L))), L), B(L, L)) > val it = B(B(B(L, B(L, B(B(L, L),
```

## Kiírás (folyt.)

---

- Az utolsó példában a kiírt sor túl hosszú lett, jó lenne eltörni a > jel előtt. Hogyan írathatunk ki egy újsor-jellet úgy, hogy az eredmény a fa érték maradjon? Például így, de ez elég körülményes:

```
- let val res = printVal fa;
 val _ = print "\n"
 in
 res
 end;
```

```
B(B(B(L, B(L, B(B(L, L)), L), B(L, L))
> val it = B(B(B(L, B(L, B(B(L, L))), L), B(L, L)) : t
```

- A before operátort az ilyen és hasonló dolgok kezelésére találták ki.

## Szekvenciális kifejezés (before)

- Az  $x$  before  $y$  kifejezés az ún. *szekvenciális kifejezés* egy változata.  
`{General.}before : 'a * 'b -> 'a`  
 $x$  before  $y$  = először az  $x$ -et, majd az  $y$ -t értékeli ki, eredménye az  $x$  értéke.  
 Precedenciaszintje 0.

- Példa before használatára:

```
- printVal fa before print "\n";
B(B(B(L, B(L, B(B(L, L))), L), B(L, L))
> val it = B(B(B(L, B(L, B(B(L, L))), L), B(L, L)) : t
```

- Az  $x$  before  $y$ -hoz hasonló a  $(x; y)$  szekvenciális kifejezés, amely azonban az *utolsó* rész kifejezésének az értékét adja eredményül.

```
- (print "A fa változó értéke =\n"; printVal fa before print "\n");
A fa változó értéke =
B(B(B(L, B(L, B(B(L, L))), L), B(L, L))
> val it = B(B(B(L, B(L, B(B(L, L))), L), B(L, L)) : t
```

## Szekvenciális kifejezés (;)

---

- Az  $(x; y)$  szekvenciális kifejezés, akárcsak az  $x$  before  $y$ , szintaktikai édesítőszér. Az  $(x; y)$  helyett írhatjuk, hogy:

```
let val _ = x in y end;
```

## Kiírás (folyt.)

- Hosszú lista, ill. egymásba skatulyázott adatszerkezetek esetén `printVal` (és maga az SML-értelmező is) alapesetben csak az első 200 listaelemet, ill. legfeljebb 20 szintet ír ki. A hosszát a `printLength`, a szintek számát a `printDepth` *frissíthető* változó szabályozza. Mindkét érték felülírható.

```
printLength : int ref | printLength := 7; !printLength;
printDepth : int ref | printDepth := 3; !printDepth;
```

- Példák:

```
- printVal [1,2,3,4,5,6,7,8,9,10] before print "\n";
[1, 2, 3, 4, 5, 6, 7, ...]
> val it = [1, 2, 3, 4, 5, 6, 7, ...] : int list
- printVal fa before print "\n";
B(B#, B#)
> val it = B(B#, B#) : t
```

- Figyelem:** a `printLength` és a `!printLength` kifejezések különböznek!

```
- printLength; | - !printLength;
> val it = ref 7 : int ref |> val it = 7 : int
```



## Kíírás (folyt.)

---

- Különböző típusú egyszerű értékeket alakítanak át füzérre a toString függvények:

```
Char.toString : char -> string
Int.toString : int -> string
Real.toString : real -> string
Bool.toString : bool -> string
Word.toString : word -> string
```

## Nyomkövetés

- Az MOSML-ben nyomkövetés csak a program szövegébe beírt kiíró függvényekkel lehetséges.
  - Példa: a length függvény két változatának kiértékelése
  - A length „naív” változata
- ```
fun length (_::xs) = 1 + length xs
  | length []      = 0;
```
- A length „naív” változata kiíró függvényekkel
- ```
fun length ((_ : int) :: xs) =
 printVal(1 + (print "&"; printVal(length(printVal xs))
 before print "$ "
)
 before print "#\n"
 | length [] = (print " * "; printVal 0 before print "%\n");
```

## Nyomkövetés (folyt.)

---

- A length iteratív változata

```
fun lengthi xs = let fun len (i, _::xs) = len(i+1, xs)
 | len (i, []) = i
 in len(0, xs)
end;
```

- A length iteratív változata kiíró függvényekkel

```
fun lengthi xs =
 let fun len (i, (_ : int) :: xs) =
 len((print " "; printVal((printVal i before print " $ ") + 1)),
 (print " & "; printVal xs)
)
 before print "#\n"
 | len (i, []) = (print " * "; printVal i before print " %\n")
 in len(0, xs)
end;
```

# Nyomkövetés

## length és egy alkalmazása

```
fun length ((_ : Int) :: xs) =
 printVal(1 + (print " & "; printVal(length(printVal xs))
 before print " $ "
)
)
 before print " #\n"
| length [] = (print " * "; printVal 0 before print " %\n");

length [1,2,3];
& [2, 3] & [3] & [] * 0 %
0 $ 1 #
1 $ 2 #
2 $ 3 #
```

## Nyomkövetés (folyt.)

---

### lengthi és egy alkalmazása

```
fun lengthi xs =
 let fun len (i, (_ : int) :: xs) =
 len((print " "; printVal((printVal i before print " $ ") + 1))),
 (print " & "; printVal xs)
 in
 before print "#\n"
 | len (i, []) = (print " * "; printVal i before print " %\n")
 in len(0, xs)
 end;

lengthi [1,2,3];
0 $ 1 & [2, 3] 1 $ 2 & [3] 2 $ 3 & [] * 3 %

#
```

## Nyomkövetés (folyt.)

- length és lengthi kiértékelésének összehasonlítása

length [1,2,3];                      lengthi [1,2,3];

|                           |   |                                                |
|---------------------------|---|------------------------------------------------|
| & [2, 3] & [3] & [] * 0 % | # | 0 \$ 1 & [2, 3] 1 \$ 2 & [3] 2 \$ 3 & [] * 3 % |
| 0 \$ 1 #                  | # |                                                |
| 1 \$ 2 #                  | # |                                                |
| 2 \$ 3 #                  | # |                                                |

- További példák a 22fp.sml állományban
- nodes és akkumulátort használó nodesa változata
- depth és akkumulátort használó deptha változata

# KIVÉTELKEZELÉS



# Kivételkezelés

- A leggyakoribb belső kivételek (többek között ld. a General könyvtárat)

| <i>Megnevezés</i> | <i>Művelet, amely a kivételt kiválthatja</i>                                     |
|-------------------|----------------------------------------------------------------------------------|
| Bind              | Értékdeklarációban a jobb oldali kifejezés nem illeszkedik a bal oldali mintára. |
| Chr               | chr pred succ                                                                    |
| Div               | / div mod                                                                        |
| Domain            | Az érték kilóg az értelmezési tartományból.                                      |
| Empty             | hd tl last                                                                       |
| Fail              | compile load loadOne                                                             |
| Interrupt         | Megszakítás ctrl/c-vel.                                                          |
| Io                | Ki/beviteli hiba. Io of {function : string, name : string, cause : exn }         |
| Match             | Mintaillesztési hiba case és handle kifejezésben, vagy függvényalkalmazásban.    |
| Option            | Hiba egy Option könyvtárbeli függvény alkalmazásakor.                            |
| Ord               | Pl. NJ93.ord "" váltja ki; elavult.                                              |
| Overflow          | ~ + - * / div mod abs ceil floor round trunc                                     |
| Size              | ~ array concat fromlist implode tabulate translate vector                        |
| Subscript         | copy drop extract nth sub substring take update                                  |



## Kivételkezelés (folyt.)

---

- Kivételt az `exception` kulcsszóval deklarálunk, a `raise` kulcsszóval jelzünk, a `handle` kulcsszóval bevezetett kifejezésben kezelünk.
- A kivételeket leggyakrabban hibák jelzésére használjuk.
- A kivételkonstruktor lehet állandó vagy függvény.
- A kivételkonstruktor állandó, ill. a kivételkonstruktorfüggvény típusa: `exn`.
- Az `exn` speciális típus:

- a kivételkonstruktorok halmaza *bővíthető*,

- az `exn` típust tartalmazó ún. *kivételcsomag* minden típussal kompatibilis:

```
- fun // {den = 0, ...} = raise Domain
 | // {num = n, den = d} = (real n) / (real d);
> val // = fn : {den : int, num : int} -> real
```

`pedig`

```
- Domain;
> val it = Domain : exn
```

## Kivételkezelés (folyt.)

---

- A raise kulcsszó olyan *kivételcsomagot* hoz létre, amelyben  $exn$  típusú érték is van.
- A kivétel kezelése a case-szerkezetre emlékeztet:  
$$E \text{ handle } P_1 \Rightarrow E_1 \mid \dots \mid P_n \Rightarrow E_n.$$
- Ha  $E$  „közöséges” értéket ad eredményül, a kivételkezelő egyszerűen továbbadja az eredményt.
- Ha  $E$  *kivételcsomagot* eredményez, akkor az SML-futtatórendszer megpróbálja a  $P_1 \dots P_n$  mintákra illeszteni.
- Ha az első illeszkedő minta a  $P_i$  ( $i = 1, 2, \dots, n$ ), akkor a kivételkezelő eredménye az  $E_i$  kifejezés eredménye.
- Ha egyetlen minta sem illeszthető a kivételcsomagra, akkor a kivételkezelő továbbpasszolja a kivételcsomagot az előző hívási szintre.

# BINÁRIS FÁK



## Egyszerű műveletek bináris fákon (folyt.)

- `fulltree n` mélységű teljes bináris fát épít, és a fa csomópontjait 1-től  $2^n - 1$ -ig beszámozza. Egy teljes bináris fában minden csomópontból pontosan két él indul ki, és minden levelének ugyanaz a szintje.

```
(* fulltree n = n mélységű teljes fa
 fulltree : int -> 'a tree *)
fun fulltree n =
 let fun ftree (_, 0) = L
 | ftree (k, n) = N(k, ftree(2*k, n-1), ftree(2*k+1, n-1))
 in ftree(1, n)
 end;
```

- `reflect a fát` a függőleges tengelye mentén tükrözi.

```
(* reflect =
 reflect : 'a tree -> 'a tree *)
fun reflect (N(v,t1,t2)) = N(v, reflect t2, reflect t1)
| reflect L = L;
```

# KÍRÁS, NYOMKÖVETÉS



## Nyomkövetés: nodes (akkumulátort nem használ)

```
* tab : string -> string
 tab i = a sorok behúzásához használandó i füzér szóközőkkel kiegészítve
)
un tab i = i ~ " ";

un nodes f =

 let (* nodes0 i f = a csomópontok száma f-ben; i a behúzásához használt füzér
 nodes0 : string -> 'a tree -> int *)
 fun nodes0 i (N(a, t1, t2)) =
 (print("\n" ~ i ~ "<"); printVal a : int; print "> ";
 printVal(1 +
 nodes0 (tab i) (printVal t2 before print " *") +
 nodes0 (tab i) (printVal t1 before print " %"))
 before print "$ "
)
 before print("#\n" ~ i)
)
 | nodes0 i L = (print("\n" ~ i); 0)
in
 nodes0 "" f
end;
```

## Nyomkövetés: nodesa (akkumulátort használ)

```
un nodesa f =
 let (* nodes0 i (f, n) = n + a csomópontok száma f-ben;
 i a behúzáshoz használt füzér
 nodes0 : string -> 'a tree * int -> int
 *)
 fun nodes0 i (N(a, t1, t2), n) =
 (print("\n" ^ i ^ "<"); printVal a : int; print "> ";
 nodes0 (tab i) (printVal t1 before print("%\n" ^ (tab i)),
 nodes0 (tab i) (printVal t2 before print("*\n" ^
 (tab i))),
 printVal(n+1) before print "$"
)
 before print("#" ^ i)
)
 | nodes0 i (L, n) = (* (print("\n" ^ i); n) *) n
 in
 nodes0 "" (f, 0)
 end;
```

# nodes és nodesa alkalmazása hét csomópontból álló teljes fára

7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree

|                                   |  |                                    |   |
|-----------------------------------|--|------------------------------------|---|
| nodes f7;                         |  | - nodesa f7;                       |   |
| 1> N(3, N(6, L, L), N(7, L, L)) * |  | <1> N(2, N(4, L, L), N(5, L, L)) % |   |
| <3> N(7, L, L) *                  |  | N(3, N(6, L, L), N(7, L, L)) *     |   |
| <7> L *                           |  | 1 \$                               |   |
| L %                               |  | <3> N(6, L, L) %                   |   |
| \$ 1 #                            |  | N(7, L, L) *                       |   |
| N(6, L, L) %                      |  | 2 \$                               |   |
| <6> L *                           |  | <7> L %                            |   |
| L %                               |  | L *                                |   |
| \$ 1 #                            |  | 3 \$ #                             |   |
| \$ 3 #                            |  | <6> L %                            |   |
| N(2, N(4, L, L), N(5, L, L)) %    |  | L *                                |   |
|                                   |  | 4 \$ #                             |   |
|                                   |  |                                    | # |

Folytatása a következő lapon.



nodes és nodesa alkalmazása ... (folyt.)

```
7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree
```

```
(nodes f7)
| (nodesa f7)
|
| <2> N(5, L, L) *
| <5> L *
| L %
| $ 1 #
| N(4, L, L) %
| <4> L *
| L %
| $ 1 #
| $ 3 #
| $ 7 #
|
| > val it = 7 : int

val it = 7 : int
```

## Nyomkövetés: depth (akkumulátort nem használ)

```

un depth f =
 let (* depth0 i f = az f fa mélysége; i a behúzáshoz használt füzér
 depth0 : string -> 'a tree -> int
 *)
 fun depth0 i (N(a : int, t1, t2)) =
 (print("\n" ^ i ^ "<"); printVal a : int; print "> ";
 printVal(1 +
 Int.max(depth0 (tab i) (printVal t2 before print " *"),
 depth0 (tab i) (printVal t1 before print " %")
)
)
 in
 depth0 "" f
 end;

```

• *Megjegyzés:* Az itt alkalmazott nodes, nodesa, depth és deptha függvények nyomkövetés nélküli változatát az előző előadásokon ismertettük.

## Nyomkövetés: deptha (akkumulátort használ)

```

un deptha f =
 let (* depth0 i (f, d) = d + az f fa mélysége; i a behúzásához használt füzér
 depth0 : string -> 'a tree * int -> int *)
 fun depth0 i (N(a : int, t1, t2), d) =
 (print("\n" ^ i ^ "<"); printVal a : int; print "> ";
 printVal(Int.max(depth0 (tab i) (printVal t2 before print(" * \n" ^
 (tab i))),
 printVal(d+1) before print " $ "
),
 depth0 (tab i) (printVal t1 before print(" % \n" ^
 (tab i))),
 printVal(d+1) before print " & "
)
)
 before print("# \n" ^ i)
)
 | depth0 i (L, d) = (print(" \n" ^ i) ; d);
 in depth0 "" (f, 0)
end;

```

depth és deptha alkalmazása két csomópontból álló teljes fára

```

7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree

```

depth f7; | - depth f7;

| - depth f7;

$$|1\rangle \otimes N(3, N(6, L, L)) * |1\rangle \otimes N(3, N(6, L, L)) *$$
$$\langle 3 \rangle \otimes N(7, L, L) * \quad | \quad 1 \quad \$$$

$\langle 3 \rangle_{N(7, L, L)} * 1$

$$\langle 7 \rangle_L^* \quad | \quad \langle 3 \rangle_{N(7, L, L)}^*$$

|    |  |   |  |     |
|----|--|---|--|-----|
| L% |  | — |  | \$2 |
|----|--|---|--|-----|

$$1\# \quad \quad \quad | \quad \quad \quad \langle 7 \rangle_{L^*}$$

|                          |  |    |
|--------------------------|--|----|
| $\mathbb{N}(6, L, L) \%$ |  | 33 |
|--------------------------|--|----|

|                     |   |             |
|---------------------|---|-------------|
| $\langle 6 \rangle$ | — | $\% \Gamma$ |
| $\Gamma^*$          | — |             |

|     |   |     |
|-----|---|-----|
| 3 & | — | L % |
|-----|---|-----|

|     |   |     |
|-----|---|-----|
| 1 # | — | 3 # |
|-----|---|-----|

$$2 \# \quad | \quad N(6, L, L) \%$$

|                                   |        |
|-----------------------------------|--------|
| $N(2, N(4, L, L), N(5, L, L)) \%$ | $2 \&$ |
|-----------------------------------|--------|

$$\begin{array}{c} \wedge \\ 9 \\ \vee \\ \text{I} \\ * \end{array}$$
$$\omega$$

%

ω  
Σ

ω  
#

ω  
#

$$N(2, N(4, L, L), N(5, L, L))\%$$

18

**Folytatása a következő lapon.**

## depth és deptha alkalmazása hét csomópontból álló teljes fára

```

7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree

(depth f7)
|
| (deptha f7)
|
|
| <2> N(5, L, L) *
|
| <5> L *
| L %
| 1 #
| N(4, L, L) %
|
| <4> L *
| L %
| 1 #
| 2 #
| 3 #
|
| > val it = 3 : int

val it = 3 : int
```

# BINÁRIS FÁK



## lista előállítás bináris fa elemeiből

---

- preorder, inorder és postorder *bináris fából listát* állít elő. A három függvény abban különbözik egymástól, hogy az egy csomópontból az ott tárolt értéket mikor veszik ki, és milyen sorrendben járják be a bal, ill. a jobb részfát.
- preorder először az értéket veszi ki, majd bejárja a bal, és azután a jobb részfát.
- inorder először bejárja a bal részfát, majd kivessi az értéket, és végül bejárja a jobb részfát.
- postorder először bejárja a bal, majd a jobb részfát, és utoljára veszi ki az értéket.
- A következő megvalósítások egyszerűek, érthetőek, de nem elég hatékonyak a @ operátor használata miatt.

## lista előállítás bináris fa elemeiből (folyt.)

- Akkumulátor nem használó változatok
  - (\* preorder f = az f fa elemeinek preorder sorrendű listája  
preorder : 'a tree -> 'a list \*)  
fun preorder (N(v,t1,t2)) = v :: preorder t1 @ preorder t2  
| preorder L = [];
  - (\* inorder f = az f fa elemeinek inorder sorrendű listája  
inorder : 'a tree -> 'a list \*)  
fun inorder (N(v,t1,t2)) = inorder t1 @ (v :: inorder t2)  
| inorder L = [];
  - (\* postorder f = az f fa elemeinek postorder sorrendű listája  
postorder : 'a tree -> 'a list \*)  
fun postorder (N(v,t1,t2)) = postorder t1 @ postorder t2 @ [v]  
| postorder L = [];
- Az akkumulátort használó változatok nehezebben érthetőek, de *hatékonyabbak*.



## lista előállítás bináris fa elemeiből (folyt.)

- (\* preord(f, vs) = az f fa elemeinek a vs lista elé fűzött,  
preorder sorrendű listája >>> rev postord !  
preord : 'a tree \* 'a list -> 'a list \*)  
fun preord (N(v,t1,t2), vs) = v::preord(t1, preord(t2,vs))  
| preord (L, vs) = vs;
- (\* inord(f, vs) = az f fa elemeinek a vs lista elé fűzött,  
inorder sorrendű listája  
inord : 'a tree \* 'a list -> 'a list \*)  
fun inord (N(v,t1,t2), vs) = inord(t1, v::inord(t2,vs))  
| inord (L, vs) = vs;
- (\* postord(f, vs) = az f fa elemeinek a vs lista elé fűzött,  
postorder sorrendű listája  
postord : 'a tree \* 'a list -> 'a list \*)  
fun postord (N(v,t1,t2), vs) = postord(t1, postord(t2, v::vs))  
| postord (L, vs) = vs;

## Bináris fa előállításá lista elemeiből: `balPreorder`

- Listát *kiegyensúlyozott* (balanced) *bináris fáva* alakítanak a következő függvények: `balPreorder`, `balInorder` és `balPostorder`; a különbség közöttük most is a bejárési sorrendben van.
- (\* `balPreorder xs = az xs lista elemeiből álló, preorder bejárású, kiegyensúlyozott fa balPreorder: 'a list -> 'a tree` \*)  

```

fun balPreorder (x::xs) =
 let val k = length xs div 2
 in
 N(x, balPreorder(List.take(xs, k)),
 balPreorder(List.drop(xs, k)))
 end
 | balPreorder [] = L;

```
- A hatékonyságot kisebb mértékben rontja, hogy `List.take` és `List.drop` egymástól függetlenül *kétszer* mennek végig a lista első felén.

## Bináris fa előállítás a lista elemeiből: `take'ndrop`

- Írjunk `take'ndrop` néven olyan függvényt, amelynek egy `xs` listából és egy `k` egészből álló pár az argumentuma, és egy olyan pár az eredménye, amelynek első tagja a lista első `k` db eleme, második tagja pedig a lista többi eleme.

```
(* take'ndrop(xs, k) = olyan pár, amelynek első tagja xs első k db
 eleme, második tagja pedig xs maradéka
 *)
take'ndrop : 'a list * int -> 'a list * 'a list

fun take'ndrop (xs, k) =
 let fun td (xs, 0, ts) = (rev ts, xs)
 | td (x::xs, k, ts) = td(xs, k-1, x::ts)
 | td ([], _, ts) = (rev ts, [])
 in
 td(xs, k, [])
 end;
```

- `take'ndrop` felhasználása, nevezetesen az eredményül átadott pár miatt módosítani kell balpreorder felépítésén.

## Bináris fa előállítására lista elemeiből: `balPreorder`, újra

### ● Ez volt:

```
fun balPreorder (x::xs) =
 let val k = length xs div 2
 in N(x, balPreorder(List.take(xs, k)), balPreorder(List.drop(xs, k)))
 end
 | balPreorder [] = L;
```

### ● Ez lett:

```
(* balPreorder xs = az xs lista elemeiből álló, preorder bejárású, ...
 balPreorder: 'a list -> 'a tree *)
fun balPreorder (x::xs) =
 let val k = length xs div 2
 val (ts, ds) = take'ndrop(xs, k)
 in N(x, balPreorder ts, balPreorder ds)
 end
 | balPreorder [] = L;
```

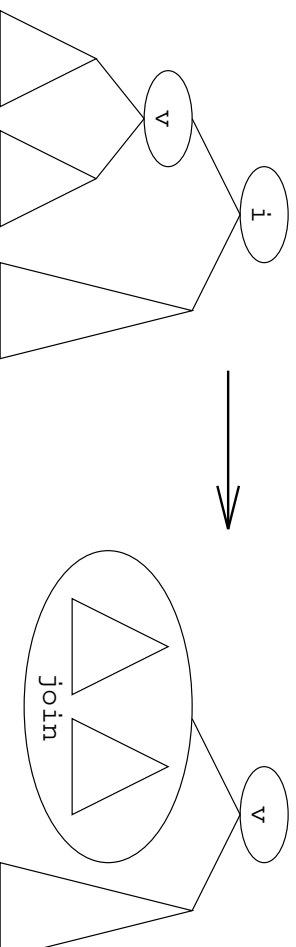
## Bináris fa előállításá lista elemeiből

- (\* balInorder xs = az xs lista elemeiből álló, inorder bejárású, kiegyensúlyozott fa  
balInorder: 'a list -> 'a tree  
\*)  
fun balInorder (xxs as x::xs) =  
 let val k = length xxs div 2  
 val ys = List.drop(xxs, k)  
 in N(hd ys, balInorder(List.take(xxs, k)), balInorder(tl ys))  
 end  
 | balInorder [] = L;  
• (\* balPostorder xs = az xs lista elemeiből álló, postorder bejárású, kiegyensúlyozott fa  
balPostorder: 'a list -> 'a tree  
\*)  
fun balPostorder xs = balPreorder(rev xs);  
• balInorder take'ndrop-pal való definiálását megfigyeljük gyakorló feladatnak.

## Elem törlése bináris fából

- Adott értékű *elemet* rekurzív módszerrel *megkeresni* egyszerű feladat.
- Új elemet beszúrni* sem nehéz: rekurzív módszerrel keresünk egy levelet, és ennek a helyére berakjuk az új értéket. Ha a fa rendezve van, ügyelnünk kell arra, hogy a rendezettség megmaradjon.

- Adott értékű *elemet* vagy *elemeket* rekurzív módszerrel *kitörölni* valamivel nehezebb: ha a törlendő érték az éppen vizsgált részfa gyökerében van, a két részre széteső fa részfáit *egyesíteni* kell, miután a törlést a két részfán már végrehajtottuk.



- Megtehetjük, hogy előbb egyesítjük a két részfát, majd az eredményül kapott fából töröljük az adott értékű elemet.

## Elem törlése bináris fából (folyt.)

- A remove rendezetlen bináris fából törli az  $i$  értékű elem összes előfordulását.

- A join-nal egyesítjük a törlés hatására létrejövő két részfát: a bal részfát lebontja, és közben az elemait egyesével berakja a jobb részfába.

```
(* join(b, j) = a b és a j fák egyesítésével létrehozott fa
 join : 'a tree * 'a tree -> 'a tree *)
fun join (N(v, lt, rt), tr) = N(v, join(lt, rt), tr)
 | join (L, tr) = tr;
```

- (\* remove(i, f) =  $i$  összes előfordulását törli  $f$ -ből  
 remove : 'a \* 'a tree -> 'a tree \*)  
 fun remove (i, N(v, lt, rt)) =  
 if  $i < v$  then N(v, remove(i, lt), remove(i, rt))  
 else join(remove(i, lt), remove(i, rt))  
 | remove (i, L) = L;

# LISTÁK HASZNÁLATA





## A „jó” számok” előállítása SML-függvénnyel

- „Jó” számok: keressük azokat a számokat, amelyek négyzete háromjegyű, és a szám fordítottjával kezdődnek (vö. Prolog-előadások).

```
(* joSzamok i = azoknak az i és 100 közötti kétjegyű számoknak a listája,
 amelyek négyzete háromjegyű, és a szám fordítottjával kezdődnek
 joSzamok : int -> int list
)
fun joSzamok i =
 if i < 100
 then if i * i div 10 = i mod 10 * 10 + i div 10
 then i :: joSzamok (i+1)
 else joSzamok (i+1)
 else [];
joSzamok 10;
```

- Írjunk általánosabb megoldást: emeljük ki a szám jó voltának és a felső határ elérésének a vizsgálatát!

## A „jó” számok” előállítása SML-függvénnyel (folyt.)

---

### • A jsz és a lim segédfüggvények

```
(* jsz1 i = igaz, ha a kétjegyű i négyzete háromjegyű és a
 fordítottjával kezdődik
 jsz1 : int -> bool *)
fun jsz1 i = i * i div 10 = i mod 10 * 10 + i div 10;

(* jsz2 i = igaz, ha a háromjegyű i egyes és százaskénti
 jegyei egyenlők
 jsz2 : int -> bool *)
fun jsz2 i = i > 100 andalso
 (i mod 10, i div 100) = (i div 100, i mod 10);

(* lim x i = igaz, ha i kisebb x-nél
 lim : int -> int -> bool *)
fun lim x i = i < x;
```

## A „jó” számok” előállítása SML-függvénnyel (folyt.)

---

### • joSzamok egy szokásos megvalósítása

```
(* joSzamok lim f i = a (lim max)-ot és az f-et kielégítő i egészek
 listája, ahol (lim max) a felső határ elérését,
 f pedig i jó szám voltát vizsgálja
 joSzamok : (int -> bool) -> (int -> bool) -> int -> int list
*)
fun joSzamok lim f i =
 if lim i
 then if f i
 then i :: joSzamok lim f (i+1)
 else joSzamok lim f (i+1)
 else [];

joSzamok (lim 100) jsz1 10;
joSzamok (lim 300) jsz2 10;
```

## A „jó” számok” előállítása SML-függvénnyel (folyt.)

### • joSzamok jobbrekurzív változata

```
(* joSzamok lim f i = a (lim max)-ot és az f-et kielégítő i egészek
 listája, ahol (lim max) a felső határ elérését,
 f pedig i jó szám voltát vizsgálja
 joSzamok : (int -> bool) -> (int -> bool) -> int -> int list *)
fun joSzamok lim f i =
 let fun jsz i zs =
 if lim i
 then jsz (i+1) (if f i then i :: zs else zs)
 else rev zs
 in
 jsz i []
 end;
 joSzamok (lim 100) jsz1 10;
 joSzamok (lim 300) jsz2 10;
```

# LEÁLLÁSI FELTÉTELEK KEZELÉSE



## Leállási feltétel kezelése

---

- Háromféle megoldást mutatunk be:
  - igazságérték – `true`, `false` – visszaadásával,
  - az `'` a `option` típus alkalmazásával,
  - kivételkezeléssel.
- Példa: „jó” számok előállítása
- A következő érték előállítására és a felső határ elérésének vizsgálatára *speciális* függvényt írunk, háromféle változatban:  
kov `x` `i` jelzi, hogy `i` kisebb-e az `x` felső határnál, és ha igen, az `i` után következő értéket adja eredményül, egyébként az eredmény tetszőleges.

## A kov függvény háromféle változatban

- Igazságérték visszaadásával:

```
(* kov11 x i = (i+1, true), ha i < x (felső határ), egyébként (i, false)
 kov11 : int -> int -> int * bool *)
fun kov11 x i = if i < x then (i+1, true) else (i, false);
```

- int option alkalmazásával:

```
(* kov21 x i = SOME(i+1), ha i < x (felső határ), egyébként NONE
 kov21 : int -> int -> int option *)
fun kov21 x i = if i < x then SOME(i+1) else NONE;
```

- Kivételjelzéssel:

```
exception Limit;
(* kov31 x i = i+1, ha i < x (felső határ), egyébként a Limit kivétel
 kov31 : int -> int -> int *)
fun kov31 x i = if i < x then i+1 else raise Limit;
```

## Leállási feltétel kezelése igazságértékkel

```

• kov : 'a -> 'a -> 'a * bool (* nxt = kov x *)

* findAll1 nxt f i = az f i összes megoldásának listája a felső határ elérését
 vizsgáló és a következő értéket eredményező nxt függvény segítségével
 findAll1 : ('a -> 'a * bool) -> ('a -> bool) -> 'a -> 'a list *)
un findAll1 nxt f i =
 let fun fAll f z zs =
 let val (j, b) = nxt z
 in
 if b then fAll f j (if f z then z::zs else zs)
 else rev zs
 end
 in
 fAll f i []
 end;

findAll1 (kov11 100) jsz1 10;
findAll1 (kov11 300) jsz2 100;

```



## Leállási feltétel kezelése az 'a option típus alkalmazásával

```

 kov : 'a -> 'a -> 'a option (* nxt = kov x *)

 * findAll2 nxt f i = az f i összes megoldásának listája a felső határ elérését
 vizsgáló és a következő értéket eredményező nxt függvény segítségével
 findAll2 : ('a -> 'a option) -> ('a -> bool) -> 'a -> 'a list
)

 un findAll2 nxt f i =
 let fun fail f z zs =
 case nxt z of
 SOME j => fail f j (if f z then z::zs else zs)
 | NONE => rev zs
 in
 fail f i []
 end;
 indAll2 (kov21 100) jsz1 10;
 indAll2 (kov21 300) jsz2 100;
```

## Leállási feltétel kezelése kivételkezeléssel

---

```
● kov : 'a -> 'a -> 'a (* nxt = kov x *)

* findAll3 nxt f i = az f i összes megoldásának listája a felső határ elérését
 vizsgáló és a következő értéket eredményező nxt függvény segítségével
 findAll3 : ('a -> 'a) -> ('a -> bool) -> 'a -> 'a list
)

un findAll3 nxt f i =
 let fun fAll f z zs = fAll f (nxt z) (if f z then z::zs else zs)
 handle Limit => rev zs
 in
 fAll f i []
 end;

indAll3 (kov31 100) jsz1 10;
indAll3 (kov31 300) jsz2 100;
```

# LISTÁK RENDEZÉSE



## Listák rendezése

---

- inssort (beszúró rendezés),
- quicksort (gyorsrendezés),
- tmsort (felülről lefelé haladó összefésülő rendezés),
- bmsort (alulról felfelé haladó összefésülő rendezés),
- smsort (simarendezés).

## Beszűrő rendezés

- Az `ins` segédfüggvény az `x` elemet a megfelelő helyre rakja be az `ys` listában:

```
(* ins (x, ys) = az x értékkel a <= reláció szerint bővített ys
 ins : real * real list -> real list
 PRE: ys a <= reláció szerint rendezett *)
fun ins (x, y::ys) = if x <= y then x::y::ys else y::ins(x, ys)
 | ins (x : real, []) = [x];
```

- `insort`-tal rekurzívan rendezzük a lista maradékát; végrehajtási ideje  $O(n^2)$ :

```
(* insort f xs = az xs elemeinek az f függvény segítségével
 rendezett listája
 insort : ('a * 'b list -> 'b list) -> 'a list -> 'b list *)
fun insort f (x::xs) = f(x, insort f xs)
 | insort _ [] = [];
```

- Példa `insort` alkalmazására:

```
insort ins [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0];
```

# LISTÁK RENDEZÉSE



## Beszűrő rendezés, generikus változat

- Az ins függvényt generikussá tesszük:

```
(* ins cmp (x, ys) = az x értékkel a cmp reláció szerint bővített ys
 ins : ('a * 'a -> bool) -> 'a * 'a list -> 'a list
 PRE: ys a cmp reláció szerint rendezve van *)
fun ins cmp (x, ys) =
 let fun ins0 (y::ys) = if cmp(x, y) then x::y::ys else y::ins0 ys
 | ins0 [] = [x]
 in ins0 ys
 end;
```

- Ezzel inssort egy újabb változata:

```
(* inssort cmp xs = az xs elemeinek a cmp reláció szerint rendezett listája
 inssort : ('a * 'a -> bool) -> 'a list -> 'a list *)
fun inssort cmp (x::xs) = ins cmp (x, inssort cmp xs)
 | inssort _ [] = [];
```

## Beszűrő rendezés, generikus változat (folyt.)

---

- `insort` eddigi változatai előbb elemeire szedik szét a rendezendő listát, majd hátulról visszafelé haladva, rendezés közben építik fel az újat.
- A jobbrekurziót és akkumulátort használó változatnak (`insort2`) kisebb veremre van szüksége, mivel a listáról leválasztott elemeket balról jobbra haladva azonnal berakja a helyükre az eredménylistában. (A két megoldás futási idejét később összehasonlítjuk).

```
fun insort2 cmp xs =
 (* sort xs zs = az xs már feldolgozott elemeinek a cmp
 reláció szerint rendezett listája zs
 sort : 'a list -> 'a list *)
 let fun sort (x::xs) zs = sort xs (ins cmp (x, zs))
 | sort [] zs = zs
 in
 sort xs []
 end;
```



## Beszűrő rendezés folder-rel és foldl-lel

---

- A második argumentumát akkumulátorként használó foldl kisebb vermet használ folder-nél, ezért inssortl hosszabb listákat tud rendezni:

```
fun inssortR cmp = foldr (ins cmp) [];
fun inssortL cmp = foldl (ins cmp) [];
```

- Példák insort-tal és insort2-vel:

```
insort op<= [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0];
inssort2 op>= [4, 4, 5, 1, 0, 8];
insort op< (explode "qwerty");
```

- Példák foldr és foldl felhasználásával:

```
fun inssortRi cmp = foldr (ins cmp) [];
fun inssortLr cmp = foldl (ins cmp) ([] : real list);
inssortRi op>= [4, 4, 5, 1, 0, 8];
inssortLr op>= [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0];
```

## A futási idők összehasonlítása

- 2000 elemet tartalmazó, véletlenszerűen előállított, illetve eredetileg éppen fordított sorrendű listák rendezéséhez szükséges futási időt mérünk.
- Véletlen eloszlású egészlistát állít elő a Random könyvtárbeli rangelist függvény:  

```
val xs2000R = Random.rangelist (1, 100000) (2000, Random.newgen());
```
- Növekvő sorrendű egészlistát állít elő a -- operátor:

```
infix --;
fun fm -- to =
 let
 fun upto to zs = if to < fm then zs else upto (to-1) (to::zs)
 in
 upto to []
 end;
val xs2000N = 1 -- 2000;
```

## A futási idők összehasonlítása (folyt.)

- A futási időt az alábbi függvénnyel mérjük meg:

```
fun futIdo (sort, sortFn) (cmp, cmpFn) (xs, kind) =
 let val starttime = Timer.startCPUTimer()
 val zs = sort cmp xs
 val {usr=tim,...} = Timer.checkCPUTimer starttime
 in
 "Int sort with " ^ sortFn ^ ", " ^ cmpFn ^
 ", length = " ^ Int.toString(length xs) ^ " (" ^
 kind ^ "), time = " ^ Time.fmt 2 tim ^ " sec\n"
 end;

val t1N = futIdo (insort, "insort") (op>=, "op>=") (xs2000N, "increasing");
val t2N = futIdo (insort2, "insort2") (op>=, "op>=") (xs2000N, "increasing");
val t1R = futIdo (insort, "insort") (op>=, "op>=") (xs2000R, "random");
val t2R = futIdo (insort2, "insort2") (op>=, "op>=") (xs2000R, "random");
```

## A futási idők összehasonlítása (folyt.)

---

- A 2000 elemű, fordított sorrendű lista rendezése az akkumulátort nem használó inssort-változatokkal több mint 5 s-ig, az akkumulátort használó változatokkal csak 0.01 s-ig tart (linux, 233 MHz-es Pentium).

```
Int sort with inssort, op>=, length = 2000 (increasing), time = 5.18 sec
Int sort with inssort2, op>=, length = 2000 (increasing), time = 0.01 sec
Int sort with inssortRi, op>=, length = 2000 (increasing), time = 5.14 sec
Int sort with inssortli, op>=, length = 2000 (increasing), time = 0.01 sec
```

- Eltűnik a különbség, ha ugyanolyan hosszú, de véletlenszerűen előállított listákat rendezünk.

```
Int sort with inssort, op>=, length = 2000 (random), time = 2.39 sec
Int sort with inssort2, op>=, length = 2000 (random), time = 2.26 sec
Int sort with inssortRi, op>=, length = 2000 (random), time = 2.40 sec
Int sort with inssortli, op>=, length = 2000 (random), time = 2.24 sec
```

## Gyorsrendezés, akkumulátor használata nélkül

```
* quicksort1 cmp xs = az xs elemeinek cmp szerint rendezett listája
quicksort1 : ('a * 'a -> order) -> 'a list -> 'a list *)
un quicksort1 cmp xs =
 let (* qs : 'a list -> 'a list
 qs ys = az ys elemeinek cmp szerint rendezett listája *)
 fun qs (m::ys) =
 let (* partition : 'a list * 'a list * ' list -> 'a list
 partition (xs, ls, rs) = ... *)
 fun partition (x::xs, ls, rs) =
 if cmp(x, m) = LESS then partition(xs, x::ls, rs)
 else partition(xs, ls, x::rs)
 | partition ([], ls, rs) = qs ls @ (m::qs rs)
 in
 partition (ys, [], [])
 end
 end
 | qs [] = []
 in
 qs xs
 end;
end;
```

## Gyorsrendezés, akkumulátor használatával

```
* quicksort2 cmp xs = az xs elemeinek cmp szerint rendezett listája
quicksort2 : ('a * 'a -> order) -> 'a list -> 'a list *)
un quicksort2 cmp xs =
 let (* qs : 'a list -> 'a list
 qs ys = az ys elemeinek cmp szerint rendezett listája *)
 fun qs (m::ys) zs =
 let (* partition : 'a list * a' list * 'a list -> 'a list
 partition (xs, ls, rs) = ... *)
 fun partition (x::xs, ls, rs) =
 if cmp(x, m) = LESS then partition(xs, x::ls, rs)
 else partition(xs, ls, x::rs)
 | partition ([], ls, rs) = qs ls (m :: qs rs zs)
 in
 partition (ys, [], [])
 end
 | qs [] zs = zs
 in
 qs xs []
 end;
end;
```

## A futási idők összehasonlítása

```

al t1 = futIdo (insort2, "insort2") (op>=, "op>=") (xs2000R, "random");
 (* ~ 2 M összehasonlítás! *)

al t3 = futIdo (quicksort2, "quicksort2")
 (Int.compare, "Int.compare") (xs20000R, "random");

al t4 = futIdo (Listsort.sort, "Listsort.sort")
 (Int.compare, "Int.compare") (xs20000R, "random");
 (* ~ 300 E összehasonlítás *)

nt sort with insort2, op>=, length = 2000 (random), time = 2.30 sec

nt sort with quicksort1, Int.compare, length = 20000 (random), time = 2.18 sec
nt sort with quicksort2, Int.compare, length = 20000 (random), time = 1.72 sec
nt sort with Listsort.sort, Int.compare, length = 20000 (random), time = 1.76 sec

nt sort with quicksort2, Int.compare, length = 200000 (random), time = 27.13 sec
nt sort with quicksort1, Int.compare, length = 200000 (random), time = 32.59 sec

al t7 = futIdo (Listsort.sort, "Listsort.sort") (Int.compare, "Int.compare")
 (Random.rangelist (1, 100000) (200000, Random.newgen()), "random");

Uncaught exception:
 Out_of_memory

```

## Összefésülő rendezések

---

- Az összefésülő rendezéshez kell egy olyan függvény, amely két listát növekvő sorrendben egyesít:

```
(* merge(xs, ys) = xs és ys elemeinek <= szerint egyesített listája
 merge : int list * int list -> int list
 *)
fun merge (xxs as x::xs, yys as y::ys)=
 if x <= y
 then x::merge(xs, yys)
 else y::merge(xxs, ys)
| merge ([], ys) = ys
| merge (xs, []) = xs;
```

- Hatékonyságrömlést okoz, hogy a részeredményeket a veremben tároljuk. Iteratív megoldás esetén meg kell fordítani az eredménylistát.



## Föjlülről lefelé haladó összefésülő rendezés

- A föjlülről lefelé haladó összefésülő rendezés (*top-down merge sort*) akkor hatékony, ha közel azonos hosszúságú az a két lista, amelyekre a rendezendő listát szétsszedjük.

```
(* tmsort xs = az xs elemeinek a <= reláció szerint rendezett listája
*)
tmsort : int list -> int list

fun tmsort xs = let val h = length xs
 val k = h div 2
 in
 if h > 1
 then merge(tmsort(List.take(xs, k)),
 tmsort(List.drop(xs, k)))
 else xs
 end;
```

- A legrosszabb esetben  $O(n \cdot \log n)$  lépésre van szükség.

# Aluról felfelé haladó „összefésülő” rendezés

- Az alulról felfelé haladó összefésülő rendezés (*bottom-up merge sort*) legegyszerűbb változata az eredeti  $k$  hosszúságú listát  $k$  darab egyelemű listára bontja, majd a szomszédos listákat összefuttatja, így 2, 4, 8, 16 stb. elemű listákat állít elő.

• R. O'Keefe algoritmus (1982) lépésről lépésre futtatja össze az egyforma hosszú részlistákat, de csak az utolsó lépésben rendezzi az összeset. Az alábbi példában az összefuttatott részlistákat *egymás mellé írással* jelöljük:

|          |    |    |   |   |   |   |   |   |   |
|----------|----|----|---|---|---|---|---|---|---|
| AB       | C  | D  | E | F | G | H | I | J | K |
| AB       | CD | E  | F | G | H | I | J | K |   |
| ABCD     | E  | F  | G | H | I | J | K |   |   |
| ABCD     | EF | G  | H | I | J | K |   |   |   |
| ABCD     | EF | G  | H | I | J | K |   |   |   |
| ABCD     | EF | GH | I | J | K |   |   |   |   |
| ABCD     | EF | GH | I | J | K |   |   |   |   |
| ABCDEFGH | I  | J  | K |   |   |   |   |   |   |
| ABCDEFGH | I  | J  | K |   |   |   |   |   |   |
| ABCDEFGH | I  | J  | K |   |   |   |   |   |   |

- 
- 
-

## Alulról fölfele haladó összefésülő rendezés (folyt.)

---

- `bmsort` a `sorting` segédfüggvényt használja, amelynek
  - első argumentuma a rendezendő lista,
  - második argumentuma a már rendezett részlistákat gyűjti,
  - harmadik argumentuma az adott lépésben összefuttatandó elem sorszám.

```
(* bmsort xs = az xs elemeinek a <= reláció szerint rendezett listája
 bmsort : int list -> int list
 *)
fun bmsort xs = sorting(xs, [], 0);
```

## Alulról fölfele haladó összefésülő rendezés (folyt.)

- Ha a rendezendő lista (`xs`) még nem fogyott el, soron következő eleméből `sorting` egyelemű listát (`[x]`) képez, és ezt a már rendezett részlisták listájára (`lss`) elé fűzve meghívja a `mergepairs` segédfüggvényt. `mergepairs` az argumentumként átadott lista két azonos hosszúságú bal oldali részlistáját fűzi egybe, feltéve persze, hogy vannak ilyenek. `k` az éppen átadott elem sorszámára. Ha a rendezendő lista kiürült, `sorting` a kétszintű lista egyetlen elemét, a rendezett listát adja eredményül.

```
(* sorting(xs, lss, k) = a még rendezetlen xs lista elemeit berakja
 a k elemet tartalmazó, már rendezett lss listába
 sorting : int list * int list list * int -> int list
 PRE: k >= 0
*)

fun sorting (x::xs, lss, k) = sorting(xs, mergepairs([x]::lss, k+1), k+1)
 | sorting ([], lss, k) = hd(mergepairs(lss, 0));
```

## Alulról felfelé haladó összefésülő rendezés (folyt.)

- mergepairs egyetlen listában gyűjti a már összefuttatott részlistákat. Az éppen átadott elem k sorszámából dönti el, hogy mit kell csinálnia a következő részlistával.

```
(* mergepairs(l1ss, n)= az n elemet tartalmazó, már rendezett l1ss lista
 első két részlistáját, ha egyforma a hosszuk, összefuttatja
 mergepairs : int list list -> int list list
 PRE: n >= 0
*)
```

```
fun mergepairs (l1ss as l1s1::l1s2::l1ss, n) = (* legalább kételemű a lista *)
 if n mod 2 = 1 then l1ss
 else mergepairs(merge(l1s1, l1s2)::l1ss, n div 2)
 | mergepairs (l1ss, _) = l1ss (* egyelemű a lista *)
```

- Ha n páratlan, mergepairs a listát változtatás nélkül adja vissza, ha páros, akkor az l1ss lista elején álló két, egyforma hosszú listát egyetlen rendezett listává futtatja össze. n=0-ra mergepairs az összes listák listáját olyan listává futtatja össze, amelynek egyetlen eleme maga is lista.

## Alulról felfelé haladó összefésülő rendezés (folyt.)

---

- A legrosszabb esetben  $O(n \cdot \log n)$  lépésre van szükség.
- A függvények működését egy példán is bemutattuk. A kezdőhívás legyen `bmsort [1,2,3,4,5,6,7,8,9] --> sorting ([1,2,3,4,5,6,7,8,9], [], 0)`
- Amíg `sorting` első argumentuma a nem üres `(x::xs)` lista, `sorting` saját magát hívja meg. A rekurzív hívás
  - első argumentuma a lépésenként egyre rövidülő `xs` lista,
  - második argumentuma a `mergepairs([x]::lss, k+1)` függvényalkalmazás eredménye, ahol kezdetben `lss = []`,
  - harmadik argumentuma `(k+1)` a már feldolgozott listaelemek száma.

```
fun sorting (x::xs, lss, k) = sorting(xs, mergepairs([x]::lss, k+1), k+1)
 | sorting ([], lss, k) = hd(mergepairs(lss, 0));
```

## Alulról fölfelé haladó összefésülő rendezés (folyt.)

---

- A következő táblázatos elrendezés
    - `mergepairs` mindkét argumentumát,
    - a rekurzív `sorting` hívás itt `j`-vel jelölt 3. argumentumát, `k+1`-et, és
    - bináris számként `k`-t mutatja lépésről lépésre.
  - A `sorting` függvény hívja `mergepairs`-t azokban a sorokban, amelyekben a `j` új értéket vesz föl, a többi helyen `mergepairs` hívása rekurzív.
  - Ne feledjük, hogy `mergepairs`-nek listák listája az első argumentuma!
  - A táblázat utolsó oszlopa a vonatkozó magyarázatra hivatkozik.
  - Vegyük észre, hogy kapcsolat van az `lss` első eleme utáni listaelemnek hossza és a `k` bitjei között! Ha `k` valamelyik bitje 1, akkor (balról jobbra haladva) az `lss` megfelelő listaelemének a hossza az adott bit helyiértékével egyenlő. A 0 értékű biteknek megfelelő listaelemnek „hiányoznak” `lss`-ből.
- ```
fun sorting (x::xs, lss, k) = sorting(xs, mergepairs([x]::lss, k+1), k+1)
  | sorting ([], lss, k) = hd(mergepairs(lss, 0));
```

Alulról fölfele haladó összefésülő rendezés (folyt.)

lss	n	j	k		<div><div>fun sorting (x::xs, lss, k) = sorting(xs, mergepairs([x]::lss, k+1), k+1) sorting ([], lss, k) = hd(mergepairs(lss, 0));</div><div>m1: Az argumentumként átadott listának egyetlen eleme van (maga is lista), ezért az argumentumot mergepairs második klóza változtatás nélkül visszaadja az öt hívó sorting-nak.</div><div>m2: n páros, ez azt jelzi, hogy az argumentumként átadott lista első két eleme egyforma hosszú lista, amelyeket merge egyetlen rendezett listává futtat össze, majd az eredménnyel mergepairs első klóza meghívja saját magát.</div><div>m3: n páratlan, ez azt jelzi, hogy az argumentumként átadott lista első két eleme nem egyforma hosszú lista, ezért az argumentumot mergepairs első klóza változtatás nélkül visszaadja az öt hívó sorting-nak.</div><div>m4: n=0, az összes listák listáját olyan listává kell összefuttatni, amelynek egyetlen lista az eleme.</div></div>
[[1]]	1	1	0	m1	
[[2], [1]]	2	2	1	m2	
[[1,2]]	1			m3	
[[3], [1,2]]	3	3	10	m3	
[[4], [3], [1,2]]	4	4	11	m2	
[[3,4], [1,2]]	2			m2	
[[1,2,3,4]]	1			m3	
[[5], [1,2,3,4]]	5	5	100	m3	
[[6], [5], [1,2,3,4]]	6	6	101	m2	
[[5,6], [1,2,3,4]]	3			m3	
[[7], [5,6], [1,2,3,4]]	7	7	110	m3	
[[8], [7], [5,6], [1,2,3,4]]	8	8	111	m2	
[[7,8], [5,6], [1,2,3,4]]	4			m2	
[[5,6,7,8], [1,2,3,4]]	2			m2	
[[1,2,3,4,5,6,7,8]]	1			m3	
[[9], [1,2,3,4,5,6,7,8]]	9	9	1000	m3	
[[9], [1,2,3,4,5,6,7,8]]	0	0		m4	
[[1,2,3,4,5,6,7,8,9]]					

Simarendezés

- Az applikatív simarendezés (*smooth sort*) algoritmus O' Keefe alulról fölfele haladó rendezéséhez hasonló, de nem egyelemű listákat, hanem növekvő *futamokat* állít elő.
 - Ha a futamok száma n -től független, azaz a lista majdnem rendezve van, akkor az algoritmus végrehajtási ideje $O(n)$, és a legrosszabb esetben is legfeljebb csak $O(n \cdot \log n)$.
- ```
(* nextrun : int list * int list -> int list * int list
 nextrun (run, xs) = ... *)
fun nextrun (run, x::xs) =
 if x < hd run then (rev run, x::xs) else nextrun(x::run, xs)
 | nextrun (run, []) = (rev run, []);
```
- nextrun eredménye egy pár, ennek
    - első tagja a futam (egy növekvő számsorozat),
    - a második tagja pedig a rendezendő lista maradéka.

## Simarendezés (folyt.)

- A futam csökkenő sorrendben bővül, kilépéskor a futamot meg kell fordítani. `msorting` a futamokat ismételtlen előállítja és összefuttatja:

```
(* msorting : int list * int list list * int -> int list
 msorting (xs, lss, k) = ... *)
fun msorting (x::xs, lss, k) =
 let val (run, tail) = nextrun([x], xs)
 in
 msorting(tail, mergepairs(run::lss, k+1), k+1)
 end
```

- `| msorting ([], lss, k) = hd(mergepairs(lss, 0))`;

- `(* msort : int list -> int list`  
`msort xs = az xs elemeinek a <= reláció szerint rendezett listája *)`  
`fun msort xs = msorting(xs, [], 0)`;

- A simarendezés egy változata sort néven megtalálható a `Listsort` könyvtárban.

## A futási idők összehasonlítása

```

un futIdo2 (sort, sortFn) (xs, kind) =
 let val starttime = Timer.startCPUTimer()
 val zs = sort xs
 val {usr=tim,...} = Timer.checkCPUTimer starttime
 in "Int sort with " ^ sortFn ^ ", length = " ^ Int.toString(length xs) ^
 " (" ^ kind ^ "), time = " ^ Time.fmt 2 tim ^ " sec\n"
 end;

al t101 = futIdo2 (tmsort, "tmsort")
 ((Random.rangelist (1, 100000) (100000, Random.newgen()))), "random");
al t102 = futIdo2 (bmsort, "bmsort")
 ((Random.rangelist (1, 100000) (100000, Random.newgen()))), "random");
al t103 = futIdo2 (smsort, "smsort")
 ((Random.rangelist (1, 100000) (100000, Random.newgen()))), "random");

nt sort with tmsort, length = 100000 (random), time = 10.96 sec
nt sort with bmsort, length = 100000 (random), time = 7.69 sec
nt sort with smsort, length = 100000 (random), time = 7.70 sec
nt sort with quicksort2, Int.compare, length = 100000 (random), time = 11.98 sec
nt sort with Listsort.sort, Int.compare, length = 100000 (random), time = 14.17 sec

```

# LISTÁK HASZNÁLATA



## $n$ vezér a sakktáblán

- Hányféleképpen rakható  $n$  vezér a sakktáblára úgy, hogy ne üssék egymást?

A vezéreket tartalmazó mezők sorának számát  $A$  sorvektort (egy egyre bővülő) listával az egyes oszlopokon belül egy  $n$  hosszú sorvektor valósítjuk meg. Egy listához balról könnyű új adott oszlophoz rendelt mezőjébe írt  $\leq s < n$  elemeket fűzni, a táblát és a vezérek helyzetét szám adja meg. Példa  $n = 4$  esetén:

```
+---+---+---+---+
| | | | |
+---+---+---+---+
```

```
...+---+---+---+
| | | | |
...+---+---+---+
```

```
0 <---> n-1
+---+---+---+---+
0 | | | | |
+---+---+---+---+
| | | | |
| | | | |
| +---+---+---+---+
V | | | | |
+---+---+---+---+
n-1 | | | | |
+---+---+---+---+
```

$A$  sorvektort (egy egyre bővülő) listával elemeket fűzni, a táblát és a vezérek helyzetét leíró listát hossztengeleje mentén tükrözzük.

```
n-1 <----- 0
...+---+---+---+
0 | | | | |
...+---+---+---+
| | | | |
| ...+---+---+---+
V | | | | |
...+---+---+---+
n-1 | | | | |
...+---+---+---+
```

## 2. vezér a sakktáblán (folyt.)

---

- Azt, hogy az új vezért üti-e a már táblára rakott másik vezér, a sorvektor vizsgálatával dönthetjük el, amely tehát azt adja meg, hogy a listaelemnek indexe által meghatározott oszlopban és a listaelem értéke által meghatározott sorban vezér van.

1. Az új vezér sorának száma, azaz az új listalem értéke nem fordulhat elő a lista már felépített részében.
2. Az új vezér átlós irányban sem lehet egy vonalban más vezérrel a táblán. Ez azt jelenti, hogy ha a sorvektort jelentő lista elejére az  $s$  sorindexet akarjuk rakni, akkor az  $i$ -edik elemének az értéke, ha van ilyen eleme, nem lehet  $s - (i+1)$ , ill.  $s + (i+1)$ .
3. A következő példa segít megvilágítani az esetet.

## 2. vezér a sakktáblán (folyt.)

- Ha a 2-es oszlopba és az  $s=1$ -es sorba akarjuk lerakni az új vezért, akkor az  $x$ -szel jelölt mezőket kell megvizsgálnunk. Az eddig létrehozott listának (sorvektornak) két eleme van, ahol a lista fejének az indexe 0. A listafej értéke nem lehet  $s-1$ , sem  $s+1$ . A lista rekurzív algoritmussal dolgozható fel.

```

...-+-+---+
 s | | |
...-+-+---+

n-1 <----- 0
...-+-+---+---+
0 | | x | |
...-+-+---+---+
| | q | | |
| ...-+-+---+---+
V | | x | |
...-+-+---+---+
n-1 | | | x |
...-+-+---+---+

```

## 2. vezér a sakktáblán (folyt.)

### • „Ütésben van”-vizsgálat

```
(* utesbenVan : int list -> bool
 utesbenVan zs = igaz, ha a (hd zs) vezér nincs ütésben
 egyetlen (tl zs)-beli vezérrel sem
*)
fun utesbenVan [] = false
 | utesbenVan (z::zs) =
 let fun uV _ _ [] = false
 | uV s1 s2 (r::rs) =
 z = r orelse s1 = r orelse s2 = r orelse
 uV (s1-1) (s2+1) rs
 in
 uV (z-1) (z+1) zs
 end;
```



## **n vezér a sakktáblán (folyt.)**

---

- Egy megoldás előállítása

```
exception Zsakutca;

(* vezerek0 : int -> int list
 vezerek0 n = a feladvány egy megoldása n vezér esetén
 *)
fun vezerek0 n =
 let
 fun vez0 z zs =
 if z = 0 andalso utesbenVan zs orelse z = n then
 raise Zsakutca
 else if length zs = n then rev zs
 else vez0 0 (z::zs) handle Zsakutca => vez0 (z+1) zs
 in
 vez0 0 []
 end;
 end;
```

## **a vezér a sakktáblán (folyt.)**

---

- Több megoldás előállításával visszalépéssel

```
(* vezerek : int -> int list list
 vezerek n = a feladvány összes megoldásának listája n vezér esetén
*)
fun vezerek n =
 let
 fun vez0 z zs =
 if z = 0 andalso utesbenVan zs orelse z = n
 then raise Zsakutca
 else if length zs = n then [rev zs]
 else (vez0 0 (z::zs) handle Zsakutca => []) @
 (vez0 (z+1) zs handle Zsakutca => [])
 in
 vez0 0 []
 end;
 end;
```

## ***vezér a sakktáblán (folyt.)***

---

- Több megoldás előállítás listák listájával

```
fun vezerek n =
 let fun vez0 z zs =
 if z = 0 andalso utesbenVan zs orelse z = n then []
 else if length zs = n then [rev zs]
 else vez0 0 (z::zs) @ vez0 (z+1) zs
 in vez0 0 [] end;
```

- Több megoldás előállítása listák listájával, akkumulátor alkalmazásával

```
fun vezerek n =
 let fun vez0 z zs ws =
 if z = 0 andalso utesbenVan zs orelse z = n then ws
 else if length zs = n then rev zs :: ws
 else vez0 0 (z::zs) (vez0 (z+1) zs ws)
 in vez0 0 [] [] end;
```

# BINÁRIS FÁK



## Bináris keresőfák

---

- Rendszerint adott kulcsú elemet keresünk, ehhez értékeket kell összehasonlítanunk egymással: a keresett kulcsnak *egyenlőségi típusúnak* kell lennie.
- A példákban a string típust használjuk.
- A függvények *kivételt* jeleznek, ha a keresett kulcsú elem nincs a keresőfában.  
`exception Bsearch of string;`
- Szebb lenne, ha *generikus függvényeket* írnanék; ezt gyakorló feladatnak hagyjuk.

## Bináris keresőfák: lookup

---

- A lookup függvény adott kulcshoz tartozó értéket ad vissza egy rendezett bináris fából:

```
(* lookup(f, b) = az f fában a b kulcshoz tartozó érték
 lookup : (string * 'a) tree * string -> 'a *)
fun lookup (N((a,x), t1, t2), b) =
 if b < a
 then lookup(t1,b)
 else if a < b
 then lookup(t2, b)
 else x
| lookup (L, b) = raise Bsearch("LOOKUP: " ^ b);
```

## Bináris keresőfák: `bininsert`

A `bininsert` függvény egy új kulcsú elemet rak be egy rendezett bináris fába, ha még nincs benne:

```
* bininsert(f, (b,y)) = az új (b,y) kulcs-érték párral bővített f fa
 bininsert : (string * 'a) tree * (string * 'a) -> (string * 'a) tree *)
 un bininsert (N((a, x), t1, t2), (b,y)) =
 if b < a
 then N((a, x), bininsert(t1, (b,y)), t2)
 else if a < b
 then N((a, x), t1, bininsert(t2, (b,y)))
 else (* a=b *) raise Bsearch("INSERT: " ^ b)
 | bininsert (L, (b,y)) = N((b,y), L, L);
```

## Bináris keresőfák: bupdate

- A bupdate függvény meglévő kulcsú elembe új értéket ír be egy rendezett bináris fában:

```
(* bupdate(f, (b,y)) = az f fa, a b kulcshoz tartozó érték helyén az y értékkel
 bupdate : (string * 'a) tree * (string * 'a) -> (string * 'a) tree *)
fun bupdate (N((a,x), t1, t2), (b,y)) =
 if b < a
 then N((a,x), bupdate(t1, (b,y)), t2)
 else if a < b
 then N((a,x), t1, bupdate(t2, (b,y)))
 else (* a=b *) N((b,y), t1, t2)
| bupdate (L, (b,y)) = raise Bsearch("UPDATE: " ^ b);
```



# LUSTA LISTÁK



## lusta lista

---

- Olyan lista, amelynek a farka függvény, ezáltal késleltetjük a kiértékelését.
- Ily módon *végtelen listákat* hozhatunk létre.
- A lusta listának hátrányai, veszélyei is vannak, pl.
  - egy lusta lista bármely részét megjeleníthetjük, de sohasem az egészet;
  - két lusta lista elemeiből páronként képezhetünk egy harmadikat, de nem számíthatjuk ki egy lusta lista elemeinek az összegét, nem kereshetjük meg benne a legkisebbet, nem fordíthatjuk meg az elemek sorrendjét;
  - úgy kell rekurziót definiálnunk, hogy nincs alapeset;
  - egy program befejeződése helyett csak azt igazolhatjuk, hogy az eredmény tetszőleges véges része véges idő alatt előáll.
- A lusta listát sorozatnak (*sequence*) nevezzük, és a seq típusoperátort használjuk a létrehozására.  
  
datatype 'a seq = Nil | Cons of 'a \* (unit -> 'a seq)

## Lista lista (folyt.)

---

- Egy sorozat fejét adja eredményül a `head` függvény; abortál, ha üres sorozatra alkalmazzuk.

```
(* head : 'a seq -> 'a *)
fun head (Cons(x, _)) = x;
```

- Egy sorozat farkát adja eredményül a `tail` függvény; abortál, ha üres sorozatra alkalmazzák.

```
(* tail : 'a seq -> 'a seq *)
fun tail (Cons(_, xf)) = xf();
```

A sorozat farka `unit -> 'a seq` típusú *függvény*, erre illesztjük az `xf` mintát `tail` fejében; `tail` törzsében `xf-et` a `()` argumentumra kell alkalmazni.

## lusta lista (folyt.)

- `consq(x, xq)` x-et berakja az xq sorozatba:
- (`* consq : 'a * 'a seq -> 'a seq *`)  
`fun consq (x, xq) = Cons(x, fn () => xq);`
- Ha a `consq` függvényt alkalmazzuk, mondjuk, az  $(x, E)$  argumentumra, az SML a `consq(x, E)` kifejezést *nem lustán* értékeli ki, hiszen alapvetően möhó kiértékelésű.
- Ha E kiértékelésének eredményét xq-val jelöljük, akkor `consq(x, E)` kiértékelése a fenti definíció szerint `Cons(x, fn () => xq)`-t eredményez.
- A `consq`-beli `fn () => xq` függvény nem késlelteti a farok (a példában E) kiértékelését `consq` alkalmazásakor.
- A lusta kiértékelés érdekében a híváskor is a `Cons(x, fn () => E)` alakot kell használnunk, `consq(x, E)` nem jó.
- Az `explicit fn () => E` késlelteti a kiértékelést, és ezzel *szükség szerinti hívatkozást* valósít meg.

## lusta lista (folyt.)

---

- Példaként a korábban megismert `from` és `take` függvények lusta változatait mutatjuk be.
- A `fromq k` sorozat egészek  $k$ -tól induló végtelen sorozata.  

```
(* fromq : int -> int seq *)
fun fromq k = Cons(k, fn () => fromq(k+1));
```
- `takeq(xq, n)` az `xq` sorozat első  $n$  eleméből képzett listát adja vissza:  

```
(* takeq : 'a seq * int -> 'a list *)
fun takeq (xq, 0) = []
 | takeq (Cons(x, xf), n) = x :: takeq(xf(), n-1)
 | takeq (Nil, n) = [];
```
- Az `'a` `seq` típus nem egészen lusta kiértékelésű: egy nemüres sorozat fejét a rendszer mindig feldolgozza.

## Egyszerű függvények lista listákra

---

- A kiszámíthatóság érdekében egy függvény eredményének tetszőleges véges része az argumentum véges részétől függhet csak.
- Amikor az eredményre szükség van, akkor ez az igény váltja ki az argumentum feldolgozását.
- Első példánkban egészeket egyesével emelünk négyzetre. Amikor szükség van rá, az eredmény farka (egy függvény) alkalmazza a `squareq` függvényt az argumentum farkára.

```
(* squareq : int seq -> int seq *)
fun squareq Nil: int seq = Nil
 | squareq (Cons (x, xf)) = Cons(x * x, fn () => squareq(xf()));
```

- Két lista hasonlóan adható össze.

```
(* addq : (int seq * int seq) -> int seq *)
fun addq (Cons (x, xf), Cons(y, yf)) = Cons(x+y, fn () => addq(xf(), yf()))
 | addq _ : int seq = Nil;
```

## Egyszerű függvények lista listákra (folyt.)

---

- Az `appendq` függvény addig nem nyúl `yq`-hoz, amíg `xq` ki nem ürül – vagyis csak akkor nyúl hozzá, ha `xq` véges. Véges sorozatot `consq`-val készíthetünk.  

```
(* appendq : 'a seq * 'a seq -> 'a seq *)
fun appendq (Cons (x, xf) , yq) = Cons(x, fn () => appendq (xf() , yq))
 | appendq (Nil , yq) = yq;
```
- Most érthetjük meg, hogy miért kellett a típusdefinícióban a `Nil` konstruktorállandót definiálni.

## Magasabb rendű függvények lista listákra

---

- A map lista változata:

```
(* mapq : ('a -> 'b) -> 'a seq -> 'b seq *)
fun mapq f (Cons (x, xf)) = Cons(f x, fn () => mapq f (xf()))
 | mapq f Nil = Nil;
```

- A filter lista változata:

```
(* filterq : ('a -> bool) -> 'a seq -> 'a seq *)
fun filterq p (Cons (x, xf)) = if p x
 then Cons(x, fn () => filterq p (xf()))
 else filterq p (xf())
 | filterq p Nil = Nil;
```

- squareq a korábban látottnál sokkal egyszerűbben definiálható mapq-val:

```
val squareq = mapq (fn i => i * i);
```



## Magasabb rendű függvények lista listákra (folyt.)

---

- Olyan számsorozatot állítunk elő, amelyben 50-nél nagyobb, 7-esre végződő egészek vannak:

```
filterq (fn n => n mod 10 = 7) (fromq 50);
```

- Az `iterateq` függvény – `a` `fromq` egy általánosítása – `a` következő sorozatot állítja elő (vö. `repeat-tel`):  $[x, f(x), f(f(x)), \dots, f^k(x), \dots]$ .

```
(* iterateq : ('a -> 'a) -> 'a -> 'a seq *)
fun iterateq f x = Cons(x, fn () => iterateq f (f x));
```

- `fromq`-t `iterateq`-val így definiálhatjuk:

```
(* fromq : int -> int seq *)
val fromq = iterateq (fn i => i+1);
```

## Álvéletlen számok

- Hagyományos álvéletlenszám-generátorok: olyan eljárások, amelyek egy *frissíthető változóban* tárolják a *seed* (mag) értéket – ebből állítják elő egy következő hívásnál a következő álvéletlen számot.

- Lusta listaként megvalósítva: a következő álvéletlen szám csak szükség esetén áll elő.

```
(* randseq : int -> real seq *)
local val a = 16807.0 and m = 2147483647.0
 (* nextrandom : real -> real
 *)
 fun nextrandom seed =
 let val t = a * seed
 in t - real(floor(t/m)) * m
 end
in
 fun randseq s = mapq (secr op/ m) (iterateq nextrandom (real s))
end;
```

## Álvéletlen számok (folyt.)

---

- Ha a `nextrandom`-ot 1.0 és 21474836467.0 közötti `seed`-re alkalmazzuk, ugyanebbe a tartományba eső más értéket állít elő az `*` `seed mod m` művelettel. (A valós számokat a túlcsoordulás elkerülésére használjuk.)
- A lusta lista előállítására `iterateq`-t `nextrandom`-ra és `seed` valós számmá alakított kezdőértékére alkalmazzuk. `mapq` gondoskodik arról, hogy a lusta listában minden értéket elosszunk `m`-mel, és így `randseq` 0.0-nál nem kisebb és 1.0-nél kisebb értékeket adjon eredményül. Látható, hogy a lusta lista a megvalósítás részleteit szépen elrejtí a felhasználó elől.
- Az előállított álvéletlen-számok 0.0-nál nem kisebb és 1.0-nél kisebb valós számok; `mapq`-val alakíthatjuk át őket 0 és 1 közötti egészekké:  
`mapq (floor o sec1 10.0 op*) (randseq 1);`

## Prímszámok előállítása *eratoszteniési szitával*

- Az algoritmus:

1. Vegyük az egészek 2-vel kezdődő sorozatát:  $(2, 3, 4, 5, 6, 7, \dots)$ .
2. Töröljük az összes 2-vel osztható számot:  $(3, 5, 7, 9, 11, \dots)$ .
3. Töröljük az összes 3-mal osztható számot:  $(5, 7, 11, 13, 17, 19, \dots)$ .
4. Töröljük az összes 5-tel osztható számot:  $(7, 11, 13, 17, 19, \dots)$ .
5. Töröljük az összes  $\dots$

- A sorozat első eleme mindig a következő prím. A sorozatban azok a számok maradnak benne, amelyek az eddig előállított prímeikkel nem oszthatók.

```
(* sift : int -> int seq -> int seq *)
fun sift p = filterq (fn n => n mod p <> 0);
```

- A `sift a p` argumentum többszöröseit törli egy lista listából.

## Prímszámok előállítása *eratoszteniési szitával* (folyt.)

---

- A sieve-nek már csak ismételtlen alkalmaznia kell sift-et a megfelelő lista listára. Mivel ez a lista sohasem üres, nem kell az üres lista listára illeszkedő változatot írunk.

```
(* sieve : int seq -> int seq *)
fun sieve (Cons (p, nf)) = Cons(p, fn () => sieve(sift p (nf())))
 | sieve Nil = Nil;
```

## Négyzetgyökvonás Newton-Raphson módszerrel

- nextapprox  $x_k$ -ből  $x_{k+1}$ -et számítja ki az  $x_{k+1} = \frac{\frac{d}{x_k} + x_k}{2}$  képlet alapján.
- (\* nextapprox : real -> real -> real \*)  
fun nextapprox a x = (a/x + x)/2.0;

- A befejeződés megállapítására egyszerű tesztet írunk:

```
(* within : real -> real seq -> real *)
fun within (eps: real) (Cons (x, xf)) =
 let val Cons (y, yf) = xf()
 in
 if abs (x-y) <= eps then y else within eps (Cons (y, yf))
 end;
```

A (Cons (y, yf)) és az xf() lista ugyanaz: az else-ágban azért használjuk az első, mert xf() meghívása költségesebb.

## Négyzetgyökvonás Newton-Raphson módszerrel (folyt.)

### Ezzel

```
(* sqrt : real -> real *)
fun sqrt a = within 1E~6 (iterateq (nextapprox a) 1.0);
```

- A példában világosan különválasztjuk a leállásvizsgálatot (termination test) a következő jelölt előállításától.

Most az abszolút különbséget ( $|x - y| < \varepsilon$ ) tesztelejük, de vizsgálhatnánk pl. a relatív különbséget ( $| \frac{x}{y} - 1 | < \varepsilon$ ) vagy az  $\frac{|x-y|}{\frac{|x|+|y|}{2}+1} < \varepsilon$  feltételt.

A feladat többi része független attól, hogy milyen leállásvizsgálatot alkalmazunk, és így is kell megfogalmazni a megoldást.

## Négyzetgyökvonás Newton-Raphson módszerrel (folyt.)

---

- Írjunk függvényt a következő jelölt előállítására, és rejtjük el a részleteket:

```
(* approxq : real -> real seq *)
fun approxq a =
 let (* nextapprox : real -> real
 *)
 fun nextapprox x = (a/x + x) / 2.0
 in iterateq nextapprox 1.0
 end;
```

- Ezzel `groot` egy „tisztább” változata:

```
(* groot : real -> real *)
val groot = within 1E~6 0 approxq;
```



## Keresztszorzatokból álló lista

- Legyen  $xq$  és  $yq$  egy-egy sorozat. Képezzünk új sorozatot az  $(x_i, y_j)$  párokból, ahol  $x_i \in xq$  és  $y_j \in yq$ !
- Először hagyományos listákra oldjuk meg a feladatot `map` és `pair` alkalmazásával.
- $xs$  és  $ys$  egy-egy lista. Képezzünk listát az  $(x_i, y_j)$  párokból, ahol  $x_i \in xs$  és  $y_j \in ys$ !
- `map-et`, `pair-t` és `List.concat`-ot alkalmazva juthatunk el a keresett függvényhez.  

```
(* pair : 'a -> 'b -> ('a * 'b) *)
fun pair x y = (x, y);
```
- A `pair-t` a `map-pel` az  $ys$  lista elemeire alkalmazva olyan párokból álló listát kapunk eredményül, amelyben a párok első tagja a rögzített  $x$  érték, a második tagja pedig az  $ys$  egy-egy eleme.  

```
map (pair x) ys
```

## Keresztszorzatokból álló lista (folyt.)

---

- Hogyan érhetjük el, hogy az  $x$  végigfusson az  $xs$  lista összes elemén? Az eddig szabad  $x$ -et kössük le egy függvény argumentumaként:

```
fn x => map (pair x) xs
```

majd alkalmazzuk újból a `map`-et erre a függvényre és  $xs$ -re:

```
map (fn x => map (pair x) xs) xs
```

- Listák listáját kapjuk eredményül, mert a belső `map` már listát adott vissza, amelynek minden eleméből újabb listát képeztünk a külső `map`-pel.  
`List.concat` elvégzi a szükséges simítást:

```
(* pairs : 'a list -> 'b list -> ('a * 'b) list *)
```

```
fun pairs xs ys = flat (map (fn x => map (pair x) ys) xs);
```

## Keresztszorzatokból álló lusta lista

- A pairss-hez hasonlóan állíthatjuk elő párok lusta listájának lusta listáját:

```
(* pairqq : 'a seq -> 'b seq -> ('a * 'b) seq seq *)
fun pairqq xq yq = mapq (fn x => mapq (pair x) yq) xq;
```

- Az eredmény véges része kíratható takeqq-val, amely a bal felső saroktól számított első  $m$  sorból és  $n$  oszlopból álló téglalapot jeleníti meg az xqq lusta listából:

```
(* 'a takeqq : (int * int) * 'a seq seq -> 'a list list *)
fun takeqq ((m, n), xqq) = map (secl n takeq) (takeq(m, xqq));
```

- Példa: olyan lusta lista, amelyben a párok első tagja az egymás után következő egészek 30-tól kezdve, második tagja pedig a prímszámok 2-től kezdve:

```
- pairqq (fromq 30) primes;
> val it = Cons (Cons ((30, 2), fn), fn): (int * int) seq seq
```

## Keresztszorzatokból álló lusta lista (folyt.)

- `- takeq((3, 5), it);`  
`> val it = [[(30, 2), \ldots, (30, 11)],`  
`[(31, 2), \ldots, (31, 11)],`  
`[(32, 2), \ldots, (32, 11)]] : (int * int) list list`

- Ha ki akarjuk símtítani a lusta listát, egy `List.concat`-hoz hasonló, lusta listákra alkalmazható függvénnyel nem megyünk semmire: ha `xq` végtelen, `appendq (xq, yq) = xq`. Azonban két lusta lista elemei páronként egymásba ékelhetők:

```
(* interleaved : 'a seq * 'a seq -> 'a seq *)
```

```
fun interleaved (Nil, yq) = yq
```

```
 | interleaved (Cons (x, xf), yq) = Cons(x, fn () => interleaved(yq, xf()));
```

- `interleaved` a rekurzív hívásban váltogatja a két lusta listát.

- `- takeq(10, interleaved(fromq 0, fromq 50));`

```
> val it = [0, 50, 1, 51, 2, 52, 3, 53, 4, 54] : int list
```

## Keresztszorzatokból álló lista lista (folyt.)

---

- `enumerate`: lista listák lista listájából egyetlen lista listát állít elő. Legyen a kétszeres mélységű lista lista feje `xq` és a farka `xqf`; alkalmazzuk `enumerate`-et rekurzívan `xqf`-re, majd az eredményt ékeljük `xq`-ba:

```
(* enumerate : 'a seq seq -> 'a seq *)
fun enumerate Nil = Nil
 | enumerate (Cons (xq, xqf)) = interleaved (xq, enumerate(xqf()));
```

Ez a „megoldás” nem jó, mert a „végtelen” lista lista miatt a rekurzió nem ér véget: az SML-ben, amely alapvetően mohó kiértékelésű, a rekurzív hívást késleltetni kell.

- Több esetet kell megkülönböztetnünk:

```
(* 'a enumerate : 'a seq seq -> 'a seq *)
fun enumerate Nil = Nil
 | enumerate (Cons (Nil, xqf)) = enumerate (xqf())
 | enumerate (Cons (Cons (x, xf), xqf)) =
 Cons(x, fn () => interleaved(enumerate(xqf()), xf()));
```

## Keresztszorzatokból álló lista lista (folyt.)

- Ha a bemenő lista lista üres, készen vagyunk. Ha nem üres, meg kell vizsgálni a lista fejét: ha ez üres, akkor folytatni kell a rekurzív hívást, ha nem üres, akkor az explicit `fn () => ...` függvénydefinióval *késleltetni kell* a rekurziót.

- Példa: pozitív egészekből álló párok egy lista listáját!

```
- val posintq = pairq (fromq 1) (fromq 1);
> val posintq = Cons (Cons ((1, 1), fn), fn):(int * int) seq seq
- takeq(15, enumerate posintq);
> val it = [(1,1), (2,1), (1,2), (3,1), (1,3), (2,2),
 (1,4), (4,1), (1,5), (2,3), (1,6), (3,2),
 (1,7), (2,4), (1,8)] : (int * int) list
```