

Deklaratív Programozás

Szeredi Péter¹ Kápolnai Richárd²

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

²`kapolnai@iit.bme.hu`
BME Irányítástechnika és Informatika Tanszék

2015 ős

Az előadók köszönetüket fejezik ki Hanák Péternek, a tárgy alapítójának

I. rész

Bevezetés

- 1 Bevezetés
- 2 Cékla: deklaratív programozás C++-ban
- 3 Erlang alapok
- 4 Prolog alapok
- 5 Keresési feladat pontos megoldása
- 6 Haladó Prolog
- 7 Haladó Erlang

Bevezetés

A tárgy témája

- Deklaratív programozási nyelvek – gyakorlati megközelítésben
- Két fő irány:
 - funkcionális programozás **Erlang** nyelven
 - logikai programozás **Prolog** nyelven
- Bevezetesként foglalkozunk a C++ egy deklaratív résznyelvével, a Cékla nyelvvel – C(É) deKLAratív része
- A **két fő nyelv**ként az **Erlang** és **Prolog** nyelvekre hivatkozunk majd (lásd követelmények)

Bevezetés

Követelmények, tudnivalók

Tartalom

- 1 Bevezetés
 - Követelmények, tudnivalók
 - Egy kedvcsináló példa Prolog nyelven
 - A példa Erlang változata

Honlap, ETS, levelezési lista

- Honlap: <http://dp.iit.bme.hu>
a jelen félév honlapja: <http://dp.iit.bme.hu/dp-current>
- ETS, az Elektronikus TanárSegéd
<http://dp.iit.bme.hu/ets>
- Levelezési lista:
<http://www.iit.bme.hu/mailman/listinfo/dp-1>
- A listára automatikusan felvesszük a tárgy hallgatóit az ETS-beli címükkel. Címet módosítani csak az ETS-ben lehet.
- A listára levelet küldeni a dp-1@iit.bme.hu címre lehet.
- Csak a feliratkozási címről küldött levelek jutnak el moderátori jóváhagyás nélkül a listatagokhoz.

Prolog-jegyzet

- Szeredi Péter, Benkő Tamás: Deklaratív programozás. Bevezetés a logikai programozásba. Budapest, 2005
 - Elektronikus változata letölthető a honlapról (ps, pdf)
 - Nyomtatott változata kifogyott
 - Kellő számú további igény esetén megszervezzük az újranyomtatást
- A SICStus Prolog kézikönyve (angol):
<http://www.sics.se/isl/sicstuswww/site/documentation.html>

Magyar nyelvű Prolog szakirodalom

- Farkas Zsuzsa, Futó Iván, Langer Tamás, Szeredi Péter:
Az MProlog programozási nyelv.
Műszaki Könyvkiadó, 1989
jó bevezetés, sajnos az MProlog beépített eljárásai nem szabványosak.
- Márkus Zsuzsa: Prologban programozni könnyű.
Novotrade, 1988
mint fent
- Futó Iván (szerk.): Mesterséges intelligencia. (9.2 fejezet, Szeredi Péter)
Aula Kiadó, 1999
csak egy rövid fejezet a Prologról
- Peter Flach: Logikai Programozás. Az intelligens következtetés példákon keresztül.
Panem — John Wiley & Sons, 2001
jó áttekintés, inkább elméleti érdeklődésű olvasók számára

Angol nyelvű Prolog szakirodalom

- Logic, Programming and Prolog, 2nd Ed., by Ulf Nilsson and Jan Maluszynski, Previously published by John Wiley & Sons Ltd. (1995)
Letölthető a <http://www.ida.liu.se/~ulfni/lpp> címről.
- Prolog Programming for Artificial Intelligence, 3rd Ed., Ivan Bratko, Longman, Paperback - March 2000
- The Art of PROLOG: Advanced Programming Techniques, Leon Sterling, Ehud Shapiro, The MIT Press, Paperback - April 1994
- Programming in PROLOG: Using the ISO Standard, C.S. Mellish, W.F. Clocksin, Springer-Verlag Berlin, Paperback - July 2003

Erlang-szakirodalom (angolul)

- Joe Armstrong: Programming Erlang. Software for a Concurrent World. The Pragmatic Bookshelf, 2007.
<http://www.pragprog.com/titles/jaerlang/programming-erlang>
- Joe Armstrong, Robert Virding, Claes Wikström, Mike Williams: Concurrent Programming in Erlang. Second Edition. Prentice Hall, 1996. Az első rész szabadon letölthető PDF-ben:
<http://erlang.org/download/erlang-book-part1.pdf>

További irodalom:

- On-line Erlang documentation
<http://erlang.org/doc.html> vagy `erl -man <module>`
- Learn You Some Erlang for great good!
<http://learnyousomeerlang.com>
- ERLANG összefoglaló magyarul
<http://nyelvek.inf.elte.hu/leirasok/Erlang/>
- Wikibooks on Erlang Programming
http://en.wikibooks.org/wiki/Erlang_Programming
- Francesco Cesarini, Simon Thompson: Erlang Programming. O'Reilly, 2009. <http://oreilly.com/catalog/9780596518189/>

Deklaratív programozás: követelmények

Nagy házi feladat (NHF)

- Programozás mindkét fő nyelven (Prolog, Erlang)
- Mindenkinek önállóan kell kódolnia (programoznia)!
- Hatékony (időlimit!), jól dokumentált („kommentezett”) programok
- A két programhoz közös, 5–10 oldalas fejlesztői dokumentáció (TXT, HTML, PDF, PS; de nem DOC vagy RTF)
- Kiadás legkésőbb a 4. héten, a honlapon, letölthető keretprogrammal
- Beadás a 9. héten; elektronikus úton (ld. honlap)
- A beadáskor és a pontozáskor külön-külön teszt sorozatot használunk (nehézségben hasonlókat, de nem azonosakat)
- Azok a programok, amelyek megoldják a tesztesetek 80%-át *létraversenyen* vesznek részt (hatékonyság, gyorsaság plusz pontokért)

Fordító- és értelmezőprogramok

- SICStus Prolog – 4.3 verzió (licenz az ETS-en keresztül kérhető)
- Más Prolog rendszer is használható (pl. SWI Prolog <http://www.swi-prolog.org/>, Gnu Prolog <http://www.gprolog.org/>), de a házi feladatokat csak akkor fogadjuk el, ha azok a SICStus rendszerben (is) helyesen működnek.
- Erlang (szabad szoftver)
- Letöltési információ a honlapon (Linux, Windows)
- Webes Prolog gyakorló felület az ETS-ben (ld. honlap)
- Kézikönyvek HTML-, ill. PDF-változatban
- Emacs szövegszerkesztő Erlang-, ill. Prolog-módban (Linux, Win95/98/NT/XP/Vista/7)
- Eclipse fejlesztői környezet (SPIDER, erlide)

Deklaratív programozás: követelmények (folyt.)

Nagy házi feladat (folyt.)

- A beadási határidőig többször is beadható, csak az utolsót értékeljük
- Pontozása mindkét fő nyelvből:
 - helyes (azaz jó eredményt időkorláton belül adó) futás esetén a 10 teszteset mindegyikére 0,5-0,5 pont, összesen max. 5 pont
 - a dokumentációra, a kód olvashatóságára, kommentezettségére max. 2,5 pont
 - tehát nyelvenként összesen max. 7,5 pont szerezhető
- A NHF súlya az osztályzatban: 15% (a 100 pontból 15)
- A megajánlott jegy előfeltétele, hogy a hallgató nagy házi feladata mindkét fő nyelvből bejusson a létraversenybe (minimum 80%-os teljesítmény)
- A NHF beadása **nem kötelező, de ajánlott!**

Deklaratív programozás: követelmények (folyt.)

Kis házi feladatok (KHF)

- 3 feladat Prologból, 3 Erlangból, 1 Céklából
- Beadás elektronikus úton (ld. honlap)
- Egy KHF beadása érvényes, ha minden tesztesetre lefut
- **Kötelező** a KHF-ek legalább 50%-ának érvényes beadása, és legalább egy érvényes KHF beadása Prologból is és Erlangból is. Azaz kötelező 1 Prolog, 1 Erlang, és 1 bármilyen (összesen 3) KHF érvényes beadása.
- Minden feladat jó megoldásáért 1-1 jutalompont (azaz a 100 alappont feletti pont) jár
- Minden KHF-nak külön határideje van, pótlási lehetőség nincs
- A házi feladatot önállóan kell elkészíteni! Másolás esetén kötelesek vagyunk fegyelmi eljárást indítani: http://www.kth.bme.hu/document/189/original/bme_rektori_utasitas_05.pdf ("Beadandó feladat ... elkészíttetése mással")

Deklaratív programozás: követelmények (folyt.)

Nagyzárthelyi, pótzárthelyi (NZH, PZH, PPZH)

- A zárthelyi **kötelező**, semmilyen jegyzet, segédlet nem használható!
- 40%-os szabály (nyelvenként a maximális részpontszám 40%-a kell az eredményességhez)
- NZH: 2015. október 21. 8:00, PZH: november 4. 8:00
- A PPZH-ra indokolt esetben a pótlási időszakban egyetlen alkalommal adunk lehetőséget
- Az NZH anyaga az addig előadott tananyag
- A PZH, ill. a PPZH anyaga azonos az NZH anyagával
- A zárthelyi súlya az osztályzatban: 15% (a 100 pontból 15)

Deklaratív programozás: követelmények (folyt.)

Gyakorlatok

- 2–9. heteken 2 órás gyakorlatok, időpontok a Neptunban
- Laptop használata megengedett
- **Kötelező** részvétel a gyakorlatok 70 %-án (pontosabban n gyakorlat esetén legalább $\lfloor 0.7n \rfloor$ gyakorlaton)
- További Prolog gyakorlási lehetőség az ETS rendszerben (gyakorló feladatok, lásd honlap)

Deklaratív programozás: követelmények (folyt.)

Beszámoló (korábbi elnevezése *írásbelivel kombinált szóbeli vizsga*)

- A beszámoló szóbeli, felkészülés írásban
- Prolog, Erlang: több kisebb feladat (programírás, -elemzés) kétszer 35 pontért
- A beszámolón szereshető max. 70 ponthoz adjuk hozzá a korábbi munkával szerzett pontokat: ZH: max. 15 pont, NHF: max. 15 pont, továbbá a pluszpontokat (KHF, létraverseny)
- A beszámolón semmilyen jegyzet, segédlet nem használható, de lehet segítséget kérni
- Nyelvenként a max. részpontszám 40%-a kell az eredményességhez
- **Kötelező**, de van megajánlott jegy = beszámoló alóli mentesség
 - Alapfeltételek: KHF, ZH, Gyakorlat teljesítése; NHF „megvédése”
 - Jó (4): a nagy házi feladat mindkét fő nyelvből bejut a létraversenybe
 - Jeles (5): legalább 40%-os eredmény a létraversenyen, mindkét fő nyelvből

Tartalom

Bevezető példa: adott értékű kifejezés előállítása

1 Bevezetés

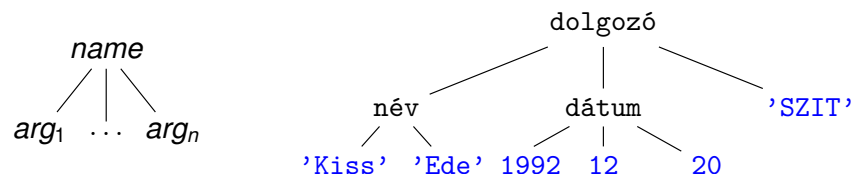
- Követelmények, tudnivalók
- Egy kedvcsináló példa Prolog nyelven
- A példa Erlang változata

- A feladat: írjunk Prolog programot a következő feladvány megoldására:
 - Adott számokból a négy alapművelet (+, -, *, /) segítségével építsünk egy megadott értékű kifejezést!
 - A számok nem „tapaszthatók” össze hosszabb számokká
 - Mindegyik adott számot pontosan egyszer kell felhasználni, sorrendjük tetszőleges lehet
 - Nem minden alapműveletet kell felhasználni, egyfajta alapművelet többször is előfordulhat
 - Zárójelek tetszőlegesen használhatók
- Példák a fenti szabályoknak megfelelő, az 1, 3, 4, 6 számokból felépített kifejezésekre: $1 + 6 * (3 + 4)$, $(1 + 3)/4 + 6$
- Viszonylag nehéz megtalálni egy olyan kifejezést, amely az 1, 3, 4, 6 számokból áll, és értéke 24

A Prolog nyelv adatfogalma

A Prolog adatokat **Prolog kifejezés**nek hívjuk (angolul: **term**). Fajtái:

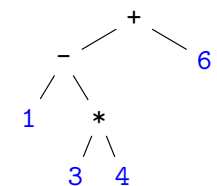
- egyszerű kifejezés: számkonstans (pl. 3), névkonstans (pl. alma, 'SZIT'), vagy változó (pl. X)
- összetett kifejezés (rekord, struktúra): $name(arg_1, \dots, arg_n)$
 - *name* egy névkonstans, az arg_i mezők tetsz. Prolog kifejezések
 - példa: `dolgozó(név('Kiss', 'Ede'), dátum(1992, 12, 20), 'SZIT')`.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



Szintaktikus „édesítőszerek” Prologban

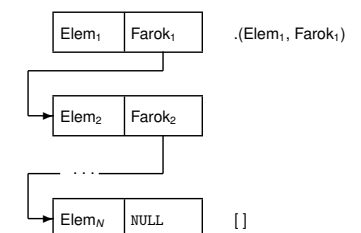
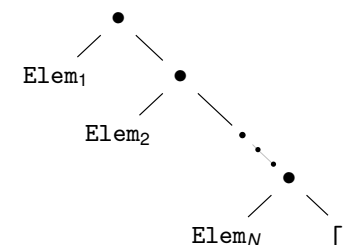
- Egy- és kétargumenetumú struktúrák operátoros (infix. prefix stb.) írásmódját: $1+2 \equiv +(1, 2)$

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



- Listák, mint speciális struktúrák

```
| ?- write_canonical([a,b,c]).
'. '(a, '. '(b, '. '(c, []))
```



Aritmetikai kifejezések kezelése Prologban – ellenőrzés

Írjunk egy `kif` nevű egyargumentumú Prolog eljárást!

A `kif(X)` hívás sikeresen fut le, ha X egy olyan **(helyes)** kifejezés-e, amely számokból a négy alpművelet (+, -, *, /) segítségével épül fel.

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

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

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

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

                               no
                               | ?-
```

Aritmetikai kifejezés levéllistájának előállítás

- A `kif_levelek` eljárás ellenőrzi a kifejezést és előállítja levéllistáját

```
% kif_levelek(Kif, L): A számokból alpműveletekkel felépülő Kif
% kifejezés leveleiben levő számok listája L.
kif_levelek(Kif, L) :-
    number(Kif), L = [Kif].      % L egyelemű, Kif-ből álló lista
kif_levelek(Kif, L) :-
    alap4(K1, K2, Kif),
    kif_levelek(K1, LX),
    kif_levelek(K2, LY),
    append(LX, LY, L).
```

```
| ?- kif_levelek(2/3-4*(5+6), L).    → L = [2,3,4,5,6]
```

- Az `append` egy beépített eljárás, fejkomentje és példafutása

```
% append(L1, L2, L3): Az L1 és L2 listák összefűzése az L3 lista.
```

```
| ?- append([1,2], [3,4], L).        → L = [1,2,3,4]
```

Aritmetikai kifejezések ellenőrzése – továbbfejlesztett változat

- A `kif` Prolog eljárás segédeljárás használó változata (javasolt formázással):

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

- Az `alap4` segédeljárás:

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

Az `append` eljárás többirányú használata

- Az `append` eljárás a fejkomentje által leírt *relációt* valósítja meg, több választ is adhat (új válasz kérése a `;` karakterrel)

```
% append(L1, L2, L3): Az L1 és L2 listák összefűzése az L3 lista.
```

```
| ?- append(L, [3], [1,2,3]).      % [1,2,3] utolsó eleme-e 3,
L = [1,2] ? ;                      % és milyen L lista van előtte?
no                                  % nincs TÖBB válasz
| ?- append([1,2], L, [1,2,3]).    % [1,2,3,4] prefixuma-e [1,2]?
L = [3] ? ; no
| ?- append(L1, L2, [1,2,3]).      % [1,2,3] hogyan bontható két részre?
L1 = [], L2 = [1,2,3] ? ;
L1 = [1], L2 = [2,3] ? ;
L1 = [1,2], L2 = [3] ? ;
L1 = [1,2,3], L2 = [] ? ; no
| ?- append(L, [2], L2).
L = [], L2 = [2] ? ;
L = [_A], L2 = [_A,2] ? ;
L = [_A,_B], L2 = [_A,_B,2] ? ; % végtelen sok válasz, problémás ...
...
```


Adott levéllistájú aritmetikai kifejezések előállítása

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

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

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

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

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

Adott értékű kifejezés előállítása – a teljes kód

```

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

% levelek_ertek_kif(L, Ertek, Kif): Kif az L listabeli számokból
% a négy alapművelettel felépített, Ertek értékű kifejezés.
levelek_ertek_kif(L, Ertek, Kif) :-
    permutation(L, PL), levelek_kif(PL, Kif), Kif == Ertek.

% levelek_kif(L, Kif): Az alapműveletekkel felépített Kif levéllistája L.
levelek_kif(L, Kif) :-
    L = [Kif], number(Kif).

levelek_kif(L, Kif) :-
    append(L1, L2, L),
    L1 \= [], L2 \= [], levelek_kif(L1, K1), levelek_kif(L2, K2),
    alap4_0(K1, K2, Kif).

% alap4_0(X, Y, K): K X-ből és Y-ből értelmes alapművelettel áll elő.
alap4_0(X, Y, X+Y).
alap4_0(X, Y, X-Y).
alap4_0(X, Y, X*Y).
alap4_0(X, Y, X/Y) :- Y \= 0. % a 0-val való osztás kiküszöbölése

```

Adott értékű kifejezés előállítás

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

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


```
| ?- 4+2 =\= 3*2. → no           | ?- 2.0 == 2. → yes
| ?- 8/3 == 2.6666666666666666. → no
```
- A példa „generál és ellenőriz” (generate-and-test) stílusú megoldása:

*% levelek_ertek_kif(L, Ertek, Kif): Kif az L listabeli számokból
% a négy alapművelet segítségével felépített olyan kifejezés,
% amelynek értéke Ertek.*

```
levek_ertek_kif(L, Ertek, Kif) :-
    permutation(L, PL), levelek_kif(PL, Kif), Kif == Ertek.

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

Tartalom

- ## 1 Bevezetés
- Követelmények, tudnivalók
 - Egy kedvcsináló példa Prolog nyelven
 - A példa Erlang változata

Erlang-kifejezések

- Erlang: nem logikai, hanem *funkcionális* programnyelv
- Összetett Erlang-kifejezéseket, függvényhívásokat értékelünk ki:


```
1> [1-3*4+6, 1-3/4+6].
[-5,6.25]
2> lists:seq(1,3).
[1,2,3]
3> {1/2, '+', 1+1}.
{0.5, '+', 2}
```
- Hármas:** $\{K_1, K_2, K_3\}$, ahol K_i tetszőleges Erlang-kifejezés. **Pár:** $\{K_1, K_2\}$.
- A *listanézet* Erlang-kifejezés a matematikai halmaznézet imitációja:


```
4> [X || X <- [1,2,3]] .           % {x|x ∈ {1,2,3}}
[1,2,3]
5> [X || X <- [1,2,3], X*X > 5] .  % {x|x ∈ {1,2,3}, x² > 5}
[3]
6> [{X,Y} || X <- [1,2,3], Y <- lists:seq(1,X)].
% {(x,y)|x ∈ {1,2,3}, y ∈ {1..x}}
[{1,1},{2,1},{2,2},{3,1},{3,2},{3,3}]
```

Adott méretű fák felsorolása

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


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

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

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

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

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

Erlang-kód:

```
% @type fa() = 1 | {fa(), '+', fa()}.
% fak(N) = az összes N levelű fa listája.
fak(1) ->
[1];
fak(N) ->
[ {BalFa, '+', JobbFa}
  || I <- lists:seq(1, N-1),
   BalFa <- fak(I),
   JobbFa <- fak(N-I)    ].
```

Matematikai nézet

Fa definíciója.

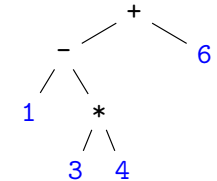
- 1 levelet tartalmazó fák halmaza: $\{1\}$
- n levelet tartalmazók:

$$\{ (b, '+', j) \mid \begin{array}{l} i \in [1 \dots n-1], \\ b \in \text{fak}(i), \\ j \in \text{fak}(n-i) \end{array} \}$$

Aritmetikai kifejezések ábrázolása

- Primitívebb a Prolognál: nem tudja automatikusan sem ábrázolni, se felsorolni az aritmetikai kifejezéseket
- Prolog egy aritmetikai kifejezést fában ábrázol:

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



- Erlangban explicit módon fel kell sorolnunk az összes ilyen fát, és explicit módon ki kell őket értékelni
- A példaprogramunkban a fenti aritmetikai kifejezést (önkényesen) egymásba ágyazott hármasokkal ábrázoljuk:


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

Adott levéllistájú aritmetikai kifejezések felsorolása

- Segédfv: egy lista összes lehetséges kettévágása nem üres listákra


```
1> kif:kettevagasok([1,3,4,6]).
[ {[1],[3,4,6]}, {[1,3],[4,6]}, {[1,3,4],[6]} ]
```
- Kifejezések adott számokból *adott sorrendben*, 4 alapműveletből:

Erlang-kód:

```
% @type kif() = {kif(),muvelet(),kif()}
%               | integer().
% @type muvelet() = '+' | '-' | '*' | '/'.
% kif(L) = L levéllistájú kifek listája.
kifek([H]) ->
[H];
kifek(L) ->
[ {B,M,J}
  || {LB,LJ} <- kettevagasok(L),
   B <- kifek(LB),
   J <- kifek(LJ),
   M <- ['+', '-', '*', '/']
].
```

Matematikai nézet:

Kifejezés (kif) definíciója.
(Az előző általánosítása.)

- Egyetlen h levelet tartalmazó kifek: $\{h\}$
- L levéllistájú kifek:

$$\{(b, m, j) \mid \begin{array}{l} L_B \oplus L_J = L, \\ b \in \text{kifek}(L_B), \\ j \in \text{kifek}(L_J), \\ m \in \{+, -, *, /\} \end{array} \}$$

Utolsó lépés: a kifejezések explicit kiértékelése

```
% ertek(K) = a K kifejezés számértéke.
ertek({B,Muvelet,J}) ->
    erlang:Muvelet(ertek(B), ertek(J));
ertek(I) ->
    I.
```

• Példák:

```
1> erlang:'+'(1,3).
4
2> kif:ertek(3).
3
3> kif:ertek({{1,'-',{3,'*',4}},'+',6}).
-5
4> kif:ertek({1,'/',0}).
** exception error: ...

% permutaciok(L) = az L lista elemeinek minden permutációja.
5> kif:permutaciok([1,3,4]).
[[1,3,4], [1,4,3], [3,1,4], [3,4,1], [4,1,3], [4,3,1]]
```

II. rész

Cékla: deklaratív programozás C++-ban

- 1 Bevezetés
- 2 Cékla: deklaratív programozás C++-ban
- 3 Erlang alapok
- 4 Prolog alapok
- 5 Keresési feladat pontos megoldása
- 6 Haladó Prolog
- 7 Haladó Erlang

Adott értékű kifejezések felsorolása – teljes kód

```
kif:megoldasok([1,3,4,6], 24).
```

```
-module(kif).
-compile([export_all]).
```

```
megoldasok(Szamok, Eredmeny) ->
    [Kif || L <- permutaciok(Szamok),
     Kif <- kifek(L),
     (catch ertek(Kif)) == Eredmeny].
```

• catch: 0-val való osztásnál keletkező kivétel miatt

```
kifek([H]) -> [H];
kifek(L) -> [ {B,M,J} || {LB,LJ} <- kettevagasok(L),
                B <- kifek(LB),
                J <- kifek(LJ),
                M <- ['+', '-', '*', '/'] ].

ertek({B,M,J}) -> erlang:M(ertek(B), ertek(J));
ertek(I) -> I.
kettevagasok(L) -> [ {LB,LJ} || I <- lists:seq(1, length(L)-1),
                        {LB,LJ} <- [lists:split(I, L)] ].

permutaciok([]) -> [[]];
permutaciok(L) -> [[H|T] || H <- L, T <- permutaciok(L--[H])].
```

Tartalom

- 2 Cékla: deklaratív programozás C++-ban
 - Néhány deklaratív paradigma C nyelven
 - Jobbrekurzió
 - A Cékla programozási nyelv
 - Listakezelés Céklában
 - Magasabb rendű függvények
 - Generikus függvények

A deklaratív programozás jelmondata

- MIT és nem HOGYAN (WHAT rather than HOW):
a *megoldás módja* helyett **inkább**
a megoldandó *feladat leírását* kell megadni
- A gyakorlatban mindkét szemponttal foglalkozni kell

Kettős szemantika:

- deklaratív szemantika
MIT (milyen feladatot) old meg a program;
- procedurális szemantika
HOGYAN oldja meg a program a feladatot.
- Új gondolkodási stílus, dekomponálható, verifikálható programok
- Új, magas szintű programozási elemek
 - rekurzió (algoritmus, adatstruktúra)
 - mintaillesztés
 - visszalépéses keresés
 - magasabb rendű függvények

Imperatív és deklaratív programozási nyelvek

- Imperatív program
 - felszólító módú, utasításokból áll
 - változó: változtatható értékű memóriahely
 - C nyelvű példa:


```
int pow(int A, int N) { // pow(A,N) = A^N
    int P = 1;          // Legyen P értéke 1!
    while (N > 0) {      // Amíg N>0 ismételd ezt:
        N = N-1;         // Csökkentsd N-et 1-gyel!
        P = P*A;         // Szorozd P-t A-val!
    }                   // Add vissza P végértékét
    return P;
  }
```
- Deklaratív program
 - kijelentő módú, egyenletekből, állításokból áll
 - változó: egyetlen, fix, a programírás idején ismeretlen értékkel bír
 - Erlang példa:


```
pow(A,N) -> if
              % Elágazás
              N==0 -> 1;      % Ha N == 0, akkor 1
              N>0 -> A * pow(A, N-1) % Ha N>0, akkor A*A^{N-1}
              end.            % Elágazás vége
```

Deklaratív programozás imperatív nyelven

Lehet pl. C-ben is deklaratívan programozni

ha nem használunk: értékadó utasítást, ciklust, ugrást stb.,
amit használhatunk: csak konstans változók, (rekurzív) függvények,
if-then-else

- Példa (a `pow` függvény deklaratív változata a `powd`):

```
// powd(A,N) = A^N
int powd(const int A, const int N) {
    if (N > 0)           // Ha N > 0
        return A * powd(A,N-1); // akkor A^N = A*A^{N-1}
    else
        return 1;        // egyébként A^N = 1
}
```

- A (fenti típusú) rekurzió költséges, nem valósítható meg konstans tárigénnyel :-(
 $\text{powd}(10,3) : 10 * \text{powd}(10,2) : 10 * (10 * \text{powd}(10,1)) : \underbrace{10 * (10 * (10 * 1))}_{\text{tárolni kell}}$

Tartalom

- 2 Cékla: deklaratív programozás C++-ban
 - Néhány deklaratív paradigma C nyelven
 - Jobbrekurzió
 - A Cékla programozási nyelv
 - Listakezelés Céklában
 - Magasabb rendű függvények
 - Generikus függvények

Hatékony deklaratív programozás

- A rekurciónak van egy hatékonyan megvalósítható változata
- Példa: döntsük el, hogy egy A szám előáll-e egy B szám hatványaként:

```
/* ispow(A,B) = létezik i, melyre B^i = A.
 * Előfeltétel: A > 0, B > 1 */
```

```
int ispow(int A, int B) {
    if (A == 1) return true;
    if (A%B==0) return ispow(A/B, B);
    return false;
}

int ispow(int A, int B) {
    again:
    if (A == 1) return true;
    if (A%B==0) {A=A/B; goto again;}
    return false;
}
```

- Itt a **színezett** rekurzív hívás átírható iteratív kódra: értékadással és ugrással helyettesíthető!
- Ez azért tehető meg, mert a rekurzióból való visszatérés után *azonnal* kilépünk az adott függvényhívásból.
- Az ilyen függvényhívást **jobbrekurzió**nak vagy **terminális rekurzió**nak vagy **farok rekurzió**nak nevezzük („tail recursion”)
- A Gnu C fordító (GCC) megfelelő optimalizálási szint mellett a rekurzív definícióból is a nem-rekurzív (jobboldali) kóddal azonos kódot generál!

Jobbrekurzív függvények

- Lehet-e jobbrekurzív kódot írni a hatványozási ($\text{pow}(A, N)$) feladatra?
 - A gond az, hogy a rekurzióból „kifelé jövet” már nem csinálhatunk semmit
 - Tehát a végeredménynek az utolsó hívás belsejében elő kell állnia!
 - A megoldás: segédfüggvény definiálása, amelyben egy vagy több ún. gyűjtőargumentumot (*akkumulátort*) helyezünk el.

- A $\text{pow}(A, N)$ jobbrekurzív (iteratív) megvalósítása:

```
// Segédfüggvény: powi(A, N, P) = P * A^N
int powi(const int A, const int N, const int P) {
    if (N > 0)
        return powi(A, N-1, P*A);
    else
        return P;
}

int powi(const int A, const int N){
    return powi(A, N, 1);
}
```

Imperatív program átírása jobbrekurzív, deklaratív programmá

- Minden ciklusnak egy segédfüggvényt feleltetünk meg (Az alábbi példában: $\text{while ciklus} \Rightarrow \text{powi}(A, N, P)$ függvény)
- A segédfüggvény argumentumai a ciklus által tartalmazott változóknak felelnek meg (powi argumentumai az A, N, P értékek)
- A segédfüggvény eredménye a ciklus által az őt követő kódnak továbbadott változó(k) értéke (powi eredménye a P végértéke)
- Példa: a hatványszámító program

```
int pow(int A, int N) {
    int P = 1;

    while (N > 0) {
        N = N-1;
        P = P*A;
    }

    return P;
}

int powi(int A, int N) {
    return powi(A, N, 1);
}

int powi(int A, int N, int P) {
    if (N > 0) return powi(A, N-1, P*A);
    else
        return P;
}
```

Példák: jobbrekurzióra átirított rekurziók

Általános rekurzió

```
// fact(N) = N!
int fact(const int N) {
    if (N==0) return 1;
    return N * fact(N-1);
}
```

```
// fib(N) =
// N. Fibonacci szám
int fib(const int N) {
    if (N<2) return N;
    return fib(N-1) +
        fib(N-2);
}
```

Jobbrekurzió

```
// facti(N, I) = N! * I
int facti(const int N, const int I) {
    if (N==0) return I;
    return facti(N-1, I*N);
}

int facti(const int N) {
    return facti(N, 1);
}
```

```
int fibi(const int N, // Segédfv
        const int Prev, const int Cur)
{
    if (N==0) return 0;
    if (N==1) return Cur;
    return fibi(N-1, Cur, Prev + Cur);
}

int fibi(const int N) {
    return fibi(N, 0, 1);
}
```

Tartalom

- 2 Cékla: deklaratív programozás C++-ban
 - Néhány deklaratív paradigma C nyelven
 - Jobbrekurzió
 - A Cékla programozási nyelv
 - Listakezelés Céklában
 - Magasabb rendű függvények
 - Generikus függvények

Cékla 2: A „Cé++” nyelv egy deKLAratív része

- Megszorítások:
 - Típusok: csak `int`, lista vagy függvény (lásd később)
 - Utasítások: `if-then-else`, `return`, blokk, kifejezés
 - Változók: csak egyszer, deklarálásukkor kaphatnak értéket (`const`)
 - Kifejezések: változókból és konstansokból kétargumentumú operátorokkal, függvényhívásokkal és feltételes szerkezetekkel épülnek fel
 - $\langle \text{aritmetikai-op} \rangle$: `+` `|` `-` `|` `*` `|` `/` `|` `%` `|`
 - $\langle \text{használati-op} \rangle$: `<` `|` `>` `|` `==` `|` `!=` `|` `>=` `|` `<=`
- C++ fordítóval is fordítható a `cekla.h` fájl birtokában: láncolt lista kezelése, függvénytípusok és kiírás
- Kiíró függvények: főleg nyomkövetéshez, ugyanis *mellékhatásuk* van!
 - `write(X)`; Az `x` kifejezés kiírása a standard kimenetre
 - `writeln(X)`; Az `x` kifejezés kiírása és soremel
- Korábban készült egy csak az `int` típust és a C résznyelvét támogató Cékla 1 fordító (Prologban íródott és Prologra is fordít)
- A (szintén Prologos) Cékla 2.x fordító letölthető a tárgy honlapjáról

Cékla Hello world!

`hello.cpp`

```
#include "cekla.h"           // így C++ fordítóval is fordítható
int main() {                 // bárhogya nevezhetnénk a függvényt
    writeln("Hello World!"); // nem-deklaratív utasítás
}                             // C++ komment megengedett
```

- Fordítás és futtatás a `cekla` programmal:


```
$ cekla hello.cpp           Cékla parancssori indítása
Welcome to Cékla 2.238: a compiler for a declarative C++ sublanguage
* Function 'main' compiled
* Code produced
To get help, type:         |* help;
|* main()                  Kiértékelendő kifejezés
Hello World!               a mellékhatás
|* ^D                      end-of-file (Ctrl+D v Ctrl+Z)
Bye
$ g++ hello.cpp && ./a.out  Szabályos C++ program is
Hello World!
```

A Cékla nyelv szintaxisa

- A szintaxist BNF jelöléssel adjuk meg, kiterjesztés:
 - ismétlés (0, 1, vagy többszöri): `«ismétlendő»...`
 - zárójelezés: `[...]`
 - `< >` jelentése: semmi
- A program szintaxisa

```
<program> ::= <preprocessor_directive>...
               <function_definition>...

<function_definition> ::= <head> <block>
<head> ::= <type> <identifier>(<formal_args>)
<type> ::= [const | < >] [int | list | fun1 | fun2]
<formal_args> ::= <formal_arg>[, <formal_arg>]... | < >
<formal_arg> ::= <type> <identifier>
<block> ::= { [<declaration> | <statement>]... }
<declaration> ::= <type> <declaration_elem>
               [, <declaration_elem>]... ;
<declaration_elem> ::= <identifier> = <expression>
```

Cékla szintaxis folytatás: utasítások, kifejezések

```

<statement> ::=      if (<expression>) <statement> <else_part>
                    | <block>
                    | <expression> ;
                    | return <expression> ;
                    | ;

<else_part> ::=      else <statement> | < >

<expression> ::=     <expression_3> [? <expression> : <expression> | < >]
<expression_3> ::=   <expression_2> [<comp_op> <expression_2>]...
<expression_2> ::=   <expression_1> [<add_op> <expression_1>]...
<expression_1> ::=   <expression_0> [<mul_op> <expression_0>]...
<expression_0> ::=   <identifier>
                    | <constant>
                    | <identifier>(<actual_args>)
                    | (<expression>)

<constant> ::=       <integer> | <string> | '<char>'
<actual_args> ::=     <expression> [, <expression>]... | < >
<comp_op> ::=         < | > | == | != | >= | <=
<add_op>  ::=         + | -
<mul_op>  ::=         * | / | %

```

Tartalom

- 2 Cékla: deklaratív programozás C++-ban
 - Néhány deklaratív paradigma C nyelven
 - Jobbrekurzió
 - A Cékla programozási nyelv
 - **Listakezelés Céklaiban**
 - Magasabb rendű függvények
 - Generikus függvények

Lista építése

- Egészeket tároló láncolt lista
- Üres lista: `nil` (globális konstans)
- Lista építése:

```

// Visszaad egy új listát: első eleme Head, farka a Tail lista.
list cons(int Head, list Tail);

```

pelda.cpp – példaprogram

```

#include "cekla.h"           // így szabályos C++ program is
int main() {                // szabályos függvénydeklaráció
    const list L1 = nil;     // üres lista
    const list L2 = cons(30, nil); // [30]
    const list L3 = cons(10, cons(20, L2)); // [10,20,30]
    writeln(L1);            // kimenet: []
    writeln(L2);            // kimenet: [30]
    writeln(L3);            // kimenet: [10,20,30]
}

```

Futtatás Céklaival

```

$ cekla
Welcome to Cekla 2.xxx: a compiler for a declarative C++ sublanguage
To get help, type:          |* help;
|* load "pelda.cpp";
* Function 'main' compiled
* Code produced
|* main();
[]
[30]
[10,20,30]
|* cons(10,cons(20,cons(30,nil)));
[10,20,30]
|* ^D
Bye
$

```

Lista szétbontása

- Első elem lekérdezése:

```
int hd(list L) // Visszaadja a nem üres L lista fejét.
```
- Többi elem lekérdezése:

```
list tl(list L) // Visszaadja a nem üres L lista farkát.
```
- Egyéb operátorok: = (inicializálás), ==, != (összehasonlítás)
- Példa:

```
int sum(const list L) { // az L lista elemeinek összege
    if (L == nil) return 0; // ha L üres, akkor 0,
    else { // különben hd(L) + sum(tl(L))
        const int X = hd(L); // segédváltozókat használhatunk,
        const list T = tl(L); // de csak konstansokat
        return X + sum(T); // rekurzió (ez nem jobbrekurzió!)
    } // Fejtörő: csak akkor lehet jobbrekurzióvá alakítani, ha
} // a T objektumot nem kell felszabadítani (destruktor)

int main() {
    const int X = sum(cons(10, cons(20, nil))); // sum([10,20]) == 30
    writeln(X); // mellékhatás: kiírjuk a 30-at
}
```

Sztringek Céklában

- Sztring nem önálló típus: karakterkódok listája, „szintaktikus édesítőszer”
- A lista a C nyelvből ismert „lezáró nullát” (`'\0'`) nem tárolja!
- write heurisztikája: ha a lista csak nyomtatható karakterek kódját tartalmazza (32..126), sztring formában íródik ki:

```
int main() {
    const list L4 = "abc"; // abc
    const list L5 = cons(97, cons(98, cons(99, nil))); // abc
    writeln(L4 == L5); // 1
    writeln(97 == 'a'); // 1
    writeln(nil == ""); // 1
    writeln(true); // 1, mint C-ben
    writeln(false); // 0, mint C-ben
    writeln(nil); // []
    writeln(L5); // abc
    writeln(cons(10, L5)); // [10,97,98,99]
    writeln(tl(L4)); // bc
}
```

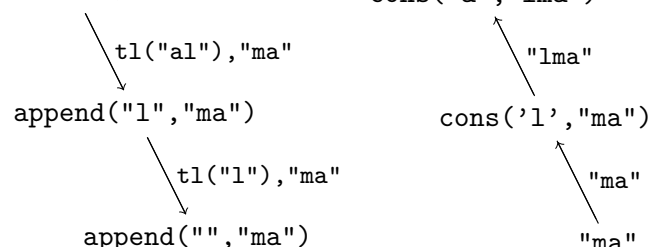
Listák összefűzése: append

- `append(L1, L2)` visszaadja `L1` és `L2` elemeit egymás után fűzve

```
// append(L1, L2) = L1 ⊕ L2 (L1 és L2 összefűzése)
```

```
list append(const list L1, const list L2) {
    if (L1 == nil) return L2;
    return cons(hd(L1), append(tl(L1), L2)); }
```
- Például `append("al", "ma") == "alma"` (vagyis `[97,108,109,97]`).

```
append("al", "ma") cons('a', "lma")
```



- $O(n)$ lépésszáma (`L1` hossza), ha a lista átadása, `cons`, `hd`, `tl` $O(1)$
- Megjegyzés: a fenti megvalósítás nem jobbrekurzív

Lista megfordítása: nrev, reverse

- Naív (négyzetes lépésszáma) megoldás

```
// nrev(L) = az L lista megfordítva
```

```
list nrev(const list L) {
    if (L == nil) return nil;
    return append(nrev(tl(L)), cons(hd(L), nil));
}
```
- Lineáris lépésszáma megoldás

```
// reverse(L) = az L lista megfordítva
```

```
list reverse(const list L) {
    return revapp(L, nil);
}
```

```
// revapp(L, L0) = az L lista megfordítása L0 elé fűzve
```

```
list revapp(const list L, const list L0) {
    if (L == nil) return L0;
    return revapp(tl(L), cons(hd(L), L0));
}
```
- Egy jobbrekurzív `appendi(L1, L2)`: `revapp(revapp(L1, nil), L2)`

További általános listakezelő függvények

- Elem keresése listában

```
// ismember(X, L) = 1, ha az X érték eleme az L listának
int ismember(const int X, const list L) {
    if (L == nil) return false;
    if (hd(L) == X) return true;
    return ismember(X, tl(L));
}
```

- Két, ismétlődésmentes listaként ábrázolt halmaz metszete

```
// intersection(L1, L2) = L1 és L2 közös elemeinek listája.
list intersection(const list L1, const list L2) {
    if (L1 == nil) return nil;
    const list L3 = intersection(tl(L1), L2);
    const int X = hd(L1);
    if (ismember(X, L2))
        return cons(X, L3);
    else return L3;
}
```

Műveletek számlistákkal

- Döntsük el, hogy egy számlista csupa negatív számból áll-e!

```
// allneg(L) = 1, ha az L lista minden eleme negatív.
int allneg(const list L) {
    if (L == nil) return true;
    if (hd(L) >= 0) return false;
    return allneg(tl(L));
}
```

- Állítsuk elő egy számlista negatív elemeiből álló listát!

```
// filterneg(L) = Az L lista negatív elemeinek listája.
list filterneg(const list L) {
    if (L == nil) return nil;
    const int X = hd(L); const list TL = tl(L);
    if (X >= 0) return filterneg(TL);
    else return cons(X, filterneg(TL));
}
```

- Állítsuk elő egy számlista elemeinek négyzeteiből álló listát!

```
// sqllist(L) = az L lista elemeinek négyzeteit tartalmazó lista.
list sqllist(const list L) {
    if (L == nil) return nil;
    const int HD = hd(L);
    const list TL = tl(L);
    return cons(HD*HD, sqllist(TL));
}
```

Imperatív C programok átírása Céklába gyűjtőargumentumokkal

- Példafeladat: Hatékony hatványozási algoritmus
 - Alaplépés: a kitevő felezése, az alap négyzetre emelése.
 - A kitevő kettes számrendszerbeli alakja szerint hatványoz.
- A megoldás imperatív C-ben és Céklában:

```
// hatv(A, H) = A^H
int hatv(int A, int H) {
    int E = 1;
    while (H > 0) {
        if (H % 2) E *= A;
        A *= A; H /= 2;
    }
    return E;
}

// hatv(A, H, E) = E * A^H
int hatv(const int A, const int H, const int E) {
    if (H <= 0) return E; // ciklus vége
    const int E1 = H%2 ? E*A : E;
    return hatv(A * A, H / 2, E1);
}
```

- A jobboldalon a while ciklusnak megfelelő segédfüggvény van, meghívása: `hatv(A, H, 1)`
- A segédfüggvény argumentumai a ciklusban szereplő változók: A, H, E
- Az A és H változók végértékére nincs szükség
- Az E változó végső értéke szükséges (ez a függvény eredménye)

Gyűjtőargumentumok: több kimenő változót tartalmazó ciklus

- A segéd eljárás kimenő változóit egy listába „csomagolva” adjuk vissza
- A segéd eljárás visszatérési értékét [szétszedjük](#).

```
// poznegki(L): kiírja
// L > 0 és < 0 elemeinek
// a számát
int poznegki(list L) {
    int P = 0, N = 0;
    while (L != nil) {
        int Fej = hd(L);
        P += (Fej > 0);
        N += (Fej < 0);
        L = tl(L);
    }
    write(P); write("-");
    writeln(N);
}

// pozneg(L, P0, N0) = [P, N], ahol P-P0, ill.
// N-N0 az L > 0, ill. < 0 elemeinek a száma
list pozneg(list L, int P0, int N0) {
    if (L == nil) return cons(P0, cons(N0, nil));
    const int Fej = hd(L);
    const int P = P0+(Fej>0);
    const int N = N0+(Fej<0);
    return pozneg(tl(L), P, N);
}

int poznegki(const list L) {
    const list PN = pozneg(L, 0, 0);
    const int P = hd(PN), N = hd(tl(PN));
    write(P); write("-");
    writeln(N);
}
```

Gyűjtőargumentumok: append kiküszöbölése

- A feladat: adott N -re N db a , majd N db b karakterből álló sztring előállítás
- Első változat, append felhasználásával

```
list anbn(int N) { // an(A, N) = [A, <-N->, A]
    list LA = nil, LB = nil;
    while (N-- > 0)
        { LA = cons('a', LA);
          LB = cons('b', LB);
        }
    return append(LA, LB); }

list an(const int A, const int N) {
    if (N == 0) return nil;
    else return cons(A, an(A, N-1)); }

list anbn(const int N) {
    return append(an('a', N), an('b', N)); }
```

- Második változat, append nélkül

```
list anbn(int N) { // an(A, N, L) = [A, <-N->, A] ⊕ L
    list L = nil; int M = N;
    while (N-- > 0)
        L = cons('b', L);
    while (M-- > 0)
        L = cons('a', L);
    return L; }

list an(int A, int N, list L) {
    if (N == 0) return L;
    else return an(A, N-1, cons(A, L)); }

list anbn(const int N) {
    return an('a', N, an('b', N, nil)); }
```

- Itt a gyűjtőargumentumok előnye nemcsak a jobbrekurzió, hanem az append kiküszöbölése is.

Tartalom

2 Cékla: deklaratív programozás C++-ban

- Néhány deklaratív paradigma C nyelven
- Jobbrekurzió
- A Cékla programozási nyelv
- Listakezelés Céklában
- Magasabb rendű függvények
- Generikus függvények

Magasabb rendű függvények Céklában

- Magasabb rendű függvény: paramétere vagy eredménye függvény
- A Cékla két függvénytípust támogat:


```
typedef int(* fun1 )(int) // Egy paraméteres egész fv
typedef int(* fun2 )(int, int) // Két paraméteres egész fv
```
- Példa: ellenőrizzük, hogy egy lista számjegyekarakterek listája-e

```
// Igaz, ha L minden X elemére teljesül a P(X) predikátum
int for_all(const fun1 P, const list L) {
    if (L == nil) return true; // triviális
    else {
        if (P(hd(L)) == false) return false; // ellenpélda?
        return for_all(P, tl(L)); // többire is teljesül?
    }
}

int digit(const int X) { // Igaz, ha X egy számjegy kódja
    if (X < '0') return false; // 48 == '0'
    if (X > '9') return false; // 57 == '9'
    return true;
}

int szamjegyek(const list L) { return for_all(digit, L); }
```

Gyakori magasabb rendű függvények: map, filter

- $\text{map}(F, L)$: az $F(X)$ elemekből álló lista, ahol X végigfutja az L lista elemeit


```
list map(const fun1 F, const list L) {
    if (L == nil) return nil;
    return cons(F(hd(L)), map(F, tl(L)));
}
```
- Például az $L=[10,20,30]$ lista elemeit eggyel növelve: $[11,21,31]$

```
int incr(const int X) { return X+1; }
```

 Így a $\text{map}(\text{incr}, L)$ kifejezés értéke $[11,21,31]$.
- $\text{filter}(P, L)$: az L lista lista azon X elemei, melyekre $P(X)$ teljesül


```
list filter(const fun1 P, const list L) {
    if (L == nil) return nil;
    if (P(hd(L))) return cons(hd(L), filter(P, tl(L)));
    else return filter(P, tl(L));
}
```
- Például keressük meg a "X=100;" sztringben a számjegyeket:
 A $\text{filter}(\text{digit}, \text{"X=100;"})$ kifejezés értéke "100" (azaz $[49,48,48]$)

Gyakori magasabb rendű függvények: a fold... család

- Hajtogatás balról – Erlang stílusban

```
// foldl(F, a, [x1,...,xn]) = F(xn, ..., F(x2, F(x1, a))...)
int foldl(const fun2 F, const int Acc, const list L) {
    if (L == nil) return Acc;
    else return foldl(F, F(hd(L),Acc), tl(L)); } // (*)
```

- Hajtogatás balról – Haskell stílusban (csak a (*) sor változik)

```
// foldlH(F, a, [x1,...,xn]) = F(...(F(a, x1), x2), ..., xn)
int foldlH(const fun2 F, const int Acc, const list L) {
    if (L == nil) return Acc;
    else return foldlH(F, F(Acc,hd(L)), tl(L)); }
```

- Futási példák, L = [1,5,3,8]

```
int xmy(int X, int Y) { return X-Y; }
int ymx(int X, int Y) { return Y-X; }

foldl (xmy, 0, L) = (8-(3-(5-(1-0)))) = 9
foldlH(xmy, 0, L) = (((((0-1)-5)-3)-8) = -17
foldl (ymx, 0, L) = (((((0-1)-5)-3)-8) = -17
foldlH(ymx, 0, L) = (8-(3-(5-(1-0)))) = 9
```

Tartalom

2 Cékla: deklaratív programozás C++-ban

- Néhány deklaratív paradigma C nyelven
- Jobbrekurzió
- A Cékla programozási nyelv
- Listakezelés Céklában
- Magasabb rendű függvények
- Generikus függvények

A fold család – folytatás

- Hajtogatás jobbról (Haskell és Erlang egyaránt:-)

```
// foldr(F, a, [x1, ..., xn]) = F(x1, F(x2, ..., F(xn, a)...))
int foldr(const fun2 F, const int Acc, const list L) {
    if (L == nil) return Acc;
    else
        return F(hd(L), foldr(F, Acc, tl(L))); }
```

- Futási példa, L = [1,5,3,8],

```
int xmy(int X, int Y) { return X-Y; }
```

```
foldr(xmy, 0, L) = (1-(5-(3-(8-0)))) = -9
```

- Egyes funkcionális nyelvekben (pl. Haskell-ben) a hajtogató függvényeknek létezik 2-argumentumú változata is: foldl1 ill. foldr1
- Itt a lista első vagy utolsó eleme lesz az Acc akkumulátor-argumentum kezdőértéke, pl.
- foldr1(F, [x1,...,xn]) = F(x1, F(x2, ..., F(xn-1,xn)...))
- (HF: melyiket egyszerűbb visszavezetni: a foldl1 vagy a foldr1 függvényt?)

Generikus függvény szintaxisa

- Cékla részben támogatja a C++ template függvényeket

- Példa:

```
template <class AnyType>
int trace_variable(const AnyType X) {
    write(" *** debug *** : ");
    writeln(X);
}
```

- Kiegészül a Cékla nyelv szintaxisa:

```
<function_definition> ::= <head> <block>
<head> ::= [<template_declaration> | <>]
           <type> <identifier>(<formal_args>)
<type> ::= [const | <>] [<template_id> | int | list | fun1 | fun2]
```

- Megjegyzés: Prologban, Erlangban nincs is típusdeklaráció

Példa: absztrakciós réteg generikus, magasabbrendű fv-nyel

- Tekintsük a sum jobbrekurzív változatát:

```
int sum(list L, int L2) {
    if (L == nil) return L2;
    return sum(tl(L), plus(hd(L), L2));
}

int plus(int A, int B) {
    return A+B;
}

int sum(list L) {
    return sum(L,0);
}
```

- Miben hasonlít a revapp algoritmusára?

```
list revapp(list L, list L2) {
    if (L == nil) return L2;
    return revapp(tl(L), cons(hd(L), L2));
}
```

- Az eltérések: L2 és a visszatérési érték, és plus, cons függvények *típusa*

```
template <class Fun, class AccType>
AccType foldlt(Fun F, AccType Acc, list L) {
    if (L == nil) return Acc;
    return foldlt(F, F(hd(L), Acc), tl(L));
}
```

```
revapp(L, L2) == foldlt(cons, L2, L)
sum(L)       == foldlt(plus, 0, L)
```

III. rész

Erlang alapok

- 1 Bevezetés
- 2 Cékla: deklaratív programozás C++-ban
- 3 Erlang alapok
- 4 Prolog alapok
- 5 Keresési feladat pontos megoldása
- 6 Haladó Prolog
- 7 Haladó Erlang

Deklaratív programozási nyelvek — a Cékla tanulságai

- Mit veszítettünk?

- a megváltoztatható változókat,
- az értékadást, ciklus-utasítást stb.,
- általánosan: a megváltoztatható állapotot

- Hogyan tudunk mégis állapotot kezelni deklaratív módon?

- az állapotot a (segéd)függvény paraméterei tárolják,
- az állapot változása (vagy helybenmaradása) explicit!

- Mit nyertünk?

- Állapotmentes szemantika: egy nyelvi elem értelme nem függ a programállapottól

- Hivatkozási átlátszóság (referential transparency) — pl. ha $f(x) = x^2$, akkor $f(a)$ **helyettesíthető** a^2 -tel.

- Egyszeres értékadás (single assignment) — párhuzamos végrehajthatóság.

- A deklaratív programok **dekomponálhatók**:

- A program részei egymástól **függetlenül** megírhatók, tesztelhetők, verifikálhatók.
- A programon könnyű következtetéseket végezni, pl. helyességét **bizonyítani**.

Tartalom

3 Erlang alapok

- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Ör
- Típus-specifikáció
- Gyakori könyvtári függvények

Funkcionális programozás: mi az?

- Programozás *függvények alkalmazásával*
- Kevésbé elterjedten *applikatív programozásnak* is nevezik (vö. function application)
- A függvény: leképezés – az argumentumából állítja elő az eredményt
A tiszta (matematikai) függvénynek nincs *mellékhatása*.

Példák funkcionális programozási nyelvekre, nyelvcsaládokra:

- Lisp, Scheme
- SML, Caml, Caml Light, OCaml, Alice
- Clean, Haskell
- Erlang
- F#

Az Erlang nyelv

- 1985: megszületik „Ellemtelben” (Ericsson–Televerket labor)
 - Agner Krarup Erlang dán matematikus, ERicsson LANGuage
- 1991: első megvalósítás, első projektek
- 1997: első OTP (Open Telecom Platform)
- 1998-tól: nyílt forráskódú, szabadon használható
- Funkcionális alapú (Functionally based)
- Párhuzamos programozást segítő (Concurrency oriented)
- Hatékony hibakezelés, hibatűrő (Fault tolerance)
- Gyakorlatban használt
[http://en.wikipedia.org/wiki/Erlang_\(programming_language\)#Distribution](http://en.wikipedia.org/wiki/Erlang_(programming_language)#Distribution)

„Programming is fun!”

Erlang emulátor

- Erlang shell (interaktív értelmező) indítása

```
$ erl
Erlang R13B03 (erts-5.7.4) [source] [smp:...

Eshell V5.7.4 (abort with ^G)
1>
1> 3.2 + 2.1 * 2.    % Lezárás és indítás „pont-bevitel”-lel!
7.4
2> atom.
atom
3> 'Atom'.
'Atom'
4> "string".
"string"
5> {ennes, 'A', a, 9.8}.
{ennes,'A',a,9.8}
6> [lista, 'A', a, 9.8].
[lista,'A',a,9.8]
7> q().
ok
```

Erlang shell: parancsok

```
1> help().
** shell internal commands **
b()      -- display all variable bindings
e(N)     -- repeat the expression in query <N>
f()      -- forget all variable bindings
f(X)     -- forget the binding of variable X
h()      -- history
v(N)     -- use the value of query <N>
rr(File) -- read record information from File (wildcards allowed)
...
** commands in module c **
c(File)  -- compile and load code in <File>
cd(Dir)  -- change working directory
help()   -- help info
l(Module) -- load or reload module
lc([File]) -- compile a list of Erlang modules
ls()     -- list files in the current directory
ls(Dir)  -- list files in directory <Dir>
m()      -- which modules are loaded
m(Mod)   -- information about module <Mod>
pwd()    -- print working directory
q()      -- quit - shorthand for init:stop()
...
```

Erlang shell: ^G és ^C

• ^G hatása

User switch command

```
--> h
c [nn] - connect to job
i [nn] - interrupt job
k [nn] - kill job
j      - list all jobs
s      - start local shell
r [node] - start remote shell
q      - quit erlang
? | h  - this message
--> c
```

• ^C hatása

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
(v)ersion (k)ill (D)b-tables (d)istribution
```

Saját program lefordítása

bevezeto.erl – Faktoriális számítása

```
-module(bevezeto). % A modul neve (kötelező; modulnév = fájlnev)
-export([fac/1]). % Látható függvények (praktikusan kötelező)
```

```
% @spec fac(N::integer()) -> F::integer().
```

```
% F = N! (F az N faktoriálisa).
```

```
fac(0) -> 1; % ha az N=0 mintaillesztés sikeres
```

```
fac(N) -> N * fac(N-1). % ha az N=0 mintaillesztés nem volt sikeres
```

• Fordítás, futtatás

```
1> c(bevezeto).
{ok,bevezeto}
2> bevezeto:fac(5).
120
3> fac(5).
** exception error: undefined shell command fac/1
4> bevezeto:fac(5)
4>
4> .
120
```

Listakezelés – rövid példák (1)

```
1> L1 = [10,20,30]. % új változó kötése, '=' a mintaillesztés
[10,20,30]
2> H = hd(L1). % hd: Built-in function (BIF)
10
3> T = tl(L1). % tl: Built-in function
[20,30]
4> b(). % kötött változók kiírása, lásd help().
H = 10
L1 = [10,20,30]
T = [20,30]
ok
5> T == [20|[30|[]]]. % egyenlőségvizsgálat
true
6> tl([]).
** exception error: bad argument
in function tl/1
called as tl([])
7> c(bevezeto).
{ok,bevezeto}
```

Listakezelés – rövid példák (2)

bevezeto.erl – folytatás

```
% sum(L) az L lista összege.
```

```
sum([]) -> 0;
```

```
sum(L) -> H = hd(L), T = tl(L), H + sum(T).
```

```
% append(L1, L2) az L1 lista L2 elé fűzve.
```

```
append([], L2) -> L2;
```

```
append(L1, L2) -> [hd(L1)|append(tl(L1), L2)].
```

```
% revapp(L1, L2) az L1 megfordítása L2 elé fűzve.
```

```
revapp([], L2) -> L2;
```

```
revapp(L1, L2) -> revapp(tl(L1), [hd(L1)|L2]).
```

```
8> bevezeto:sum(L1).
```

```
60
```

```
9> bevezeto:append(L1, [a,b,c,d]).
```

```
[10,20,30,a,b,c,d]
```

```
10> bevezeto:revapp(L1, [a,b,c,d]).
```

```
[30,20,10,a,b,c,d]
```


Tartalom

3 Erlang alapok

- Bevezetés
- **Típusok**
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Őr
- Típusspecifikáció
- Gyakori könyvtári függvények

Típusok

- Az Erlang erősen típusos nyelv, bár nincs típusdeklaráció
- A típusellenőrzés dinamikus és nem statikus
- Alaptípusok

<i>magyarul</i>	<i>angolul</i>
Szám (egész, lebegőpontos)	Number (integer, float)
Atom vagy Névkonstans	Atom
Függvény	Function
Ennes (rekord)	Tuple (record)
Lista	List

- További típusok

Pid	Pid (Process identifier)
Port	Port
Hivatkozás	Reference
Bináris	Binary

Szám (number)

- Egész
 - Pl. 2008, -9, 0
 - Tetszőleges számrendszerben `radix#szám` alakban, pl. `2#101001`, `16#fe`
 - Az egész korlátlan pontosságú, pl. `12345678901234567890123456789012345678901234`
 - Karakterkód
 - Ha nyomtatható: `$z`
 - Ha vezérlő: `$\n`
 - Oktális számmal: `$\012`
- Lebegőpontos
 - Pl. `3.14159`, `0.2e-22`
 - IEEE 754 64-bit

Függvény

(Kicsit később...)

Atom

- Névkonstans (nem fűzér!)
- Kisbetűvel kezdődő, bővített alfanumerikus¹ karaktersorozat, pl. `sicstus`, `erlang_OTP`
- Bármilyen² karaktersorozat is az, ha egyszeres idézőjelbe tesszük, pl. `'SICStus'`, `'erlang OTP'`, `'35 May'`
- Hossza tetszőleges, vezérlőkaraktereket is tartalmazhat, pl. `'ez egy hosszú atom, ékezetes betűkkel spékelve'`
`'formázókarakterekkel \n\\c\\f\\r'`
- Saját magát jelöli
- Hasonló a Prolog névkonstanshoz (atom)
- C++, Java nyelvekben a legközelebbi rokon: enum

¹Bővített alfanumerikus: kis- vagy nagybetű, számjegy, aláhúzás (`_`), kukac (`@`)

²bármilyen latin-1 kódolású karaktert tartalmazó (R14B)

Lista (list)

- Korlátlan számú, tetszőleges kifejezésből álló sorozat
- Lineáris rekurzív adatszerkezet:
 - vagy üres (`[]` jellel jelöljük),
 - vagy egy elemből áll, amelyet egy lista követ: `[Elem|Lista]`
- Első elemét, ha van, a lista *fejének* nevezzük
- Első eleme utáni, esetleg üres részét a lista *farkának* nevezzük
 - Egyelemű lista: `[elem]`
 - Fejből-farokból létrehozott lista: `[elem|[]]`, `['első'|['második']]`
 - Többelemű lista:
 - `[elem,123,3.14,'elem']`
 - `[elem,123,3.14|['elem']]`
 - `[elem,123|[3.14,'elem']]`
- A konkatenáció műveleti jele: `++`

```
11> ['egy'|['két']] ++ [elem,123|[3.14,'elem']]
[egy,két,elem,123,3.14,elem]
```

Ennes (tuple)

- Rögzített számú, tetszőleges kifejezésből álló sorozat
- Példák: `{2008, erlang}`, `{'Joe', 'Armstrong', 16.99}`
- Nullás: `{}`
- Egyelemű ennes \neq ennes eleme, pl. `{elem} \neq elem`

Fűzér (string)

- Csak rövidítés, tkp. karakterkódok listája, pl. `"erl" \equiv [\$e,\$r,\$l] \equiv [101,114,108]`
- Az Eshell a nyomtatható karakterkódok listáját fűzérként írja ki:


```
12> [101,114,108]
"erl"
```
- Ha más érték is van a listában, listaként írja ki:


```
13> [31,101,114,108]
[31,101,114,108]
14> [a,101,114,108]
[a,101,114,108]
```
- Egymás mellé írással helyettesíthető, pl.


```
15> "erl" "ang".
"erlang"
```

Tartalom

3 Erlang alapok

- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Őr
- Típusspecifikáció
- Gyakori könyvtári függvények

Term, változó

Term

- *Közelítő rekurzív definíció:* szám-, atom-, vagy függvénytípusú értékekből vagy *termekből* ennes- és listakonstruktorokkal felépített kifejezés.
- Néhány típussal (ref., port, pid, binary) most nem foglalkozunk
- Tovább nem egyszerűsíthető kifejezés, érték a programban
- Minden termnek van típusa
- Pl. 10 vagy {'Diák Detti', [{khf, [cekla, prolog, erlang, prolog]}]}
- Pl. nem term: 5 + 6, mert műveletet (függvényhívást) tartalmaz
- Termek összehasonlítási sorrendje (v.ö. típusok)
number < atom < ref. < fun < port < pid < tuple < list < binary

Változó

- Nagybetűvel kezdődő, bővített alfanumerikus karaktersorozat, más szóval *név*
- A változó lehet *szabad* vagy *kötött*
- A szabad változónak nincs értéke, típusa
- A kötött változó értéke, típusa valamely konkrét term értéke, típusa
- Minden változóhoz *csak egyszer* köthető érték, azaz kötött változó nem kaphat értéket

Kifejezés

- Lehet
 - term
 - változó
 - minta
 - Minta: term alakú kifejezés, amelyben szabad változó is lehet
 - termék \subset minták³
 - változók \subset minták
- továbbá
 - összetett kifejezés, függvényalkalmazás
 - szekvenciális kifejezés
 - egyéb: if, case, try/catch, catch stb.
- Kifejezés kiértékelése alapvetően: **mohó** (eager vagy strict evaluation).

```
16> Nevező = 0.
```

```
0
```

```
17> (Nevező > 0) and (1 / Nevező > 1).
```

```
** exception error: bad argument in an arithmetic expression
```

³néhány nem túl gyakorlatias ellenpéldától eltekintve, például hibás:

```
[X,fun erlang:'+' / 2] = [5,fun erlang:'+' / 2].
```

Kifejezés: összetett és függvényalkalmazás

Függvényalkalmazás

- Szintaxisa
 - fnév(arg₁, arg₂, ..., arg_n)
vagy
 - modul:fnév(arg₁, arg₂, ..., arg_n)

Például

```
18> length([a,b,c]).
```

```
3
```

```
19> erlang:tuple_size({1,a,'A',"1aA"}).
```

```
4
```

```
20> erlang:'+'(1,2).
```

```
3
```

Összetett kifejezés

- Kiértékelhető műveleteket, függvényeket is tartalmazó, kifejezés, pl. 5+6, vagy: [{5+6, math:sqrt(2+fib(X))}, alma]
- Különbözik a termtől, a termben a művelet/függvényhívás tiltott

Kifejezés: szekvenciális

Szekvenciális kifejezés

- Kifejezések sorozata, szintaxisa:
 - `begin exp1, exp2, ..., expn end`
 - `exp1, exp2, ..., expn`
- A `begin` és `end` párt akkor kell kiírni, ha az adott helyen egyetlen kifejezésnek kell állnia
- Értéke az utolsó kifejezés értéke: `expn`
- 21> `L2 = [10,20,30], H2 = hd(L2), T2 = tl(L2),`
 21> `H2 + bevezeto:sum(T2).`
 60
 22> `[begin a, "a", 5, [1,2] end, a].`
`[[1,2],a]`
- Eltérés Prologtól (gyakori hiba):** a vessző itt nem jelent logikai ÉS kapcsolatot, csak egymásutániságot!
 - `expi`-ben ($i < n$) vagy változót kötünk,
 - vagy mellékhatást keltünk (pl. kiírás)

Függvénydeklaráció

- Egy vagy több, pontosvesszővel (;) elválasztott *klózból* állhat.
- Alakja:


```
fnév(A11, ..., A1m) [when ŐrSz1] -> SzekvenciálisKif1;  
...  
fnév(An1, ..., Anm) [when ŐrSzn] -> SzekvenciálisKifn.
```
- A függvényt a neve, az „aritása” (paramétereinek száma), valamint a moduljának a neve azonosítja.
- Az azonos nevű, de eltérő aritású függvények nem azonosak!
- Példák:


```
fac(N) -> fac(N, 1).  
  
fac(0, R) -> R;  
fac(N, R) -> fac(N-1, N*R).
```
- (Őrök bemutatása kicsit később)

Tartalom

3 Erlang alapok

- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés**
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Őr
- Típusspecifikáció
- Gyakori könyvtári függvények

Minta, mintaillesztés (egyesítés)

- Minta (pattern): term alakú kifejezés, amelyben szabad változó is lehet
- Sikeres illesztés esetén a szabad változók kötötté válnak, értékük a megfelelő részkifejezés értéke lesz.
- A mintaillesztés (egyesítés) műveleti jele: =
Alakja: `MintaKif = TömörKif`
- Mintaillesztés \neq értékadás!
- Példák:

<code>Pi = 3.14159</code>	\rightsquigarrow^4	<code>Pi \mapsto^5 3.14159</code>
<code>3 = P</code>	\rightsquigarrow	hiba (jobboldal nem tömör)
<code>[H1 T1] = [1,2,3]</code>	\rightsquigarrow	<code>H1 \mapsto 1, T1 \mapsto [2,3]</code>
<code>[1,2 T2] = [1,2,3]</code>	\rightsquigarrow	<code>T2 \mapsto [3]</code>
<code>[H2 [3]] = [1,2,3]</code>	\rightsquigarrow	meghiúsulás, hiba
<code>{A1,B1} = {{a}, 'Beta'}</code>	\rightsquigarrow	<code>A1 \mapsto {a}, B1 \mapsto 'Beta'</code>
<code>{{a},B2} = {{a}, 'Beta'}</code>	\rightsquigarrow	<code>B2 \mapsto 'Beta'</code>

⁴`Kif \rightsquigarrow` jelentése: „Kif kiértékelése után”.

⁵`x \mapsto v` jelentése: „x a v értékhez van kötve”.

Mintaillesztés függvény klózaira – 1. példa

- Függvény alkalmazásakor a klóz kiválasztása is mintaillesztéssel történik
- Máshol is, pl. a case vezérlési szerkezetnél is történik illesztés

khf.erl – DP kisházik ellenőrzése

```
-module(khf).
-compile(export_all).           % mindent exportál, csak teszteléshez!
%-export([kiadott/1, ...]). % tesztelés után erre kell cserélni

% kiadott(Ny) az Ny nyelven kiadott kisházik száma.
kiadott(cekla) -> 1;           % 1. klóz
kiadott(prolog) -> 3;          % 2. klóz
kiadott(erlang) -> 3.          % 3. klóz

2> khf:kiadott(cekla). % sikeres illesztés az 1. klózra
1
3> khf:kiadott(erlang). % sikertelen: 1-2. klóz, sikeres: 3. klóz
3
4> khf:kiadott(java). % 3 sikertelen illesztés után hiba
** exception error: no function clause matching ...
```

Kitérő: változók elnevezése

- Az előző függvényre figyelmeztetést kapunk:


```
Warning: variable 'E' is unused
Warning: variable 'Fej' is unused
```
- A figyelmeztetés kikapcsolható alulvonással (_) kezdődő változóval

khf.erl – folytatás

```
elofordul1(_E, []) -> 0;
elofordul1(E, [E|Farok]) -> 1 + elofordul1(E, Farok);
elofordul1(E, [_Fej|Farok]) -> elofordul1(E, Farok).
```

- A változó neve akár el is hagyható, de az _ elnevezésű változót tömör kifejezésben nem lehet használni (vagyis nem lehet kiértékelni)
- Több _ változónk is lehet, például:

$$[H, _, _] = [1, 2, 3] \leadsto H \mapsto 1$$
- Találós kérdés: miben különböznek az alábbi mintaillesztések?


```
A=hd(L).
[A|_] = L.
[A, _|_] = L.
```

Mintaillesztés függvény klózaira – 2. példa

- Hányszor szerepel egy elem egy listában? Első megoldásunk:

khf.erl – folytatás

```
% @spec elofordul0(E::term(), L::[term()]) -> N::integer().
% E elem az L listában N-szer fordul elő.
elofordul0(E, []) -> 0; % 1.
elofordul0(E, [E|Farok]) -> 1 + elofordul0(E, Farok); % 2.
elofordul0(E, [_Fej|Farok]) -> elofordul0(E, Farok). % 3.

5> khf:elofordul1(a, [a,b,a,1]). % 2. klóz, majd 3., 2., 3., 1.
2
6> khf:elofordul1(java, [cekla,prolog,prolog]). % 3., 3., 3., 1.
0
```

- A minták összekapcsolhatóak, az E változó több argumentumban is szerepel: `elofordul1(E, [E|Farok]) -> ...`
- Számít a klózek sorrendje, itt pl. a 3. általánosabb, mint a 2.!

Mintaillesztés függvény klózaira – 3. példa

- Teljesítette-e egy hallgató a khf követelményeket?


```
7> Hallgato1 = {'Diák Detti',
                [{khf, [cekla,prolog,erlang,prolog]},
                 {zh, 59}]}.
```

khf.erl – folytatás

```
% @spec megfelelt(K::kovetelmeny(), H::hallgato()) -> true | false.
megfelelt(khf, {_Nev, [{khf, L}|_]}) ->
  C = elofordul1(cekla, L),
  P = elofordul1(prolog, L),
  E = elofordul1(erlang, L),
  (P >= 1) and (E >= 1) and (C + P + E >= 3);
megfelelt(zh, {_Nev, [{zh, Pont}|_]}) ->
  Pont >= 24;
megfelelt(K, {_Nev, [_|F]}) ->
  megfelelt(K, {_Nev, F});
megfelelt(_, {_, []}) ->
  false.
```

„Biztonságos” illesztés: ha egyik mindig sikerül

- Mit kezdünk a kiadott(java) kiértékelésekor keletkező hibával?
- Erlangban gyakori: jelezzük a sikert vagy a hibát az eredményben

khf.erl – folytatás

```
% @spec safe_kiadott(Ny::atom()) -> {ok, Db::integer()} | error.
% Az Ny nyelven Db darab kisházit adtak ki.
safe_kiadott(cekla) -> {ok, 1};
safe_kiadott(prolog) -> {ok, 3};
safe_kiadott(erlang) -> {ok, 3};
safe_kiadott(_Ny) -> error.    % e klóz mindig illeszthető
```

- Az ok és az error atomokat konvenció szerint választottuk
- Kötés: ha a minta egyetlen szabad változó (_Ny), az illesztés sikeres
- De hogy férjünk hozzá az eredményhez?
8> khf:safe_kiadott(cekla).
{ok,1}
9> khf:safe_kiadott(java).
error

case példa

- Az adott nyelvből hány százalékot adtunk be?

khf.erl – folytatás

```
% @spec safe_teljesitmeny(Nyelv::atom(), Beadott_db::integer()) ->
% {ok, Teljesitmeny::float()} | error.
safe_teljesitmeny(Nyelv, Beadott_db) ->
    case safe_kiadott(Nyelv) of
    {ok, Kiadott_db} -> {ok, Beadott_db / Kiadott_db};
    error -> error
    end.
```

- Függvény klózzai összevonhatóak a case segítségével:

```
kiadott(Ny) ->
    case Ny of
    cekla -> 1;
    prolog -> 3;
    erlang -> 3
    end.

kiadott(cekla) -> 1;
kiadott(prolog) -> 3;
kiadott(erlang) -> 3.    helyett írható:
```

Feltételes kifejezés mintaillesztéssel (case)

- `case Kif of`
Minta₁ [when ŰrSz₁] -> SzekvenciálisKif₁;
...
Minta_n [when ŰrSz_n] -> SzekvenciálisKif_n
end.
- Kiértékelés: balról jobbra
- Értéke: az első illeszkedő minta utáni szekvenciális kifejezés
- Ha nincs ilyen minta, hibát jelez
1> X=2, case X of 1 -> "1"; 3 -> "3" end.
** exception error: no case clause matching 2
2> X=2, case X of 1 -> "1"; 2 -> "2" end.
"2"
3> Y=fagylalt, 3 * case Y of fagylalt -> 100; tolcser -> 15 end.
300
4> Z=kisauto, case Z of fagylalt -> 100;
4> tolcser -> 15;
4> Barmi -> 99999 end.
99999

Tartalom

3 Erlang alapok

- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Űr
- Típus-specifikáció
- Gyakori könyvtári függvények

Listanézet (List Comprehensions)

- Listanézet (List Comprehensions): `[Kif | Minta <- Lista, Feltétel]`
Közelítő definíció: Kif kifejezések listája, ahol a Minta a Lista olyan eleme, melyre Feltétel igaz.
- Feltétel tetszőleges logikai kifejezés lehet. A Mintában előforduló változónevek elfedik a listakifejezésen kívüli azonos nevű változókat.
- Kis példák


```
1> [2*X | X <- [1,2,3]].    % { 2·x | x ∈ {1,2,3} }
[2,4,6]
2> [2*X | X <- [1,2,3], X rem 2 /= 0, X > 2].
[6]
3> lists:seq(1,3).          % egészek 1-től 3-ig
[1,2,3]
4> [{X,Y} | X <- [1,2,3,4], Y <- lists:seq(1,X)].
[{1,1},
 {2,1},{2,2},
 {3,1},{3,2},{3,3},
 {4,1},{4,2},{4,3},{4,4}]
```
- Pontos szintaxis: `[X | Q1, Q2, ...]`, ahol X tetszőleges kifejezés, Q_i lehet generátor (`Minta <- Lista`) vagy szűrőfeltétel (predikátum)

Listanézet: érdekes példák

- Quicksort

```
qsort([]) ->
[];
qsort([Pivot|Tail]) ->
  qsort([X | X <- Tail, X < Pivot])
  ++ [Pivot] ++
  qsort([X | X <- Tail, X >= Pivot]).
```

- Permutáció

```
perms([]) ->
[];
perms(L) ->
  [[H|T] | H <- L, T <- perms(L--[H])].
```

- Listák különbsége: `As--Bs` vagy `lists:subtract(As,Bs)`
`As--Bs` az `As` olyan másolata, amelyből ki van hagyva a `Bs`-ben előforduló összes elem balról számított első előfordulása, feltéve, hogy volt ilyen elem `As`-ben

Listanézet: példák

- Pitagoraszai számhármak, melyek összege legfeljebb N

```
pitag(N) ->
  [{A,B,C} |
   A <- lists:seq(1,N),
   B <- lists:seq(1,N),
   C <- lists:seq(1,N),
   A+B+C =:= N,
   A*A+B*B =:= C*C]
  ].
```

- Hányszor fordul elő egy elem egy listában?

```
elofordul2(Elem, L) ->
  length([X | X <- L, X==Elem]).
```

- A khf követelményeket teljesítő hallgatók

```
L = [{'Diák Detti', [{khf, [...]}]}, {'Lusta Ludvig', []}],
[Nev | {Nev, M} <- L, khf:megfelelt(khf, {Nev, M})].
```

Tartalom

3 Erlang alapok

- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Őr
- Típus-specifikáció
- Gyakori könyvtári függvények

Függvényérték

- A funkcionális nyelvekben a függvény is *érték*:
 - leírható (jelölhető)
 - van típusa
 - névhez (változóhoz) köthető
 - adatszerkezet eleme lehet
 - paraméterként átadható
 - függvényalkalmazás eredménye lehet (zárójelezni kell!)
- Névtelen függvény (függvényjelölés) mint érték


```
fun (A11, ..., A1m) [when ŐrSz1] -> SzekvenciálisKif1;
...;
(An1, ..., Anm) [when ŐrSzn] -> SzekvenciálisKifn
end.
```
- Már deklarált függvény mint érték


```
fun Modul:Fnev/Aritas % például fun bevezeto:sum/1
fun Fnev/Aritas % ha az Fnev „látható”, pl. modulból
```

„Programming is fun!”

Függvényérték: példák

```
2> Area1 = fun ({circle,R}) -> R*R*3.14159;
              ({rectan,A,B}) -> A*B;
              ({square,A}) -> A*A
              end.
#Fun<erl_eval.6.13229925>
3> Area1({circle,2}).
12.56636
4> Osszeg = fun bevezeto:sum/1.
#Fun<bevezeto.sum.1>
5> Osszeg([1,2]).
3
6> fun bevezeto:sum/1([1,2]).
3
7> F1 = [Area1, Osszeg, fun bevezeto:sum/1, 12, area].
[#Fun<erl_eval.6.13229925>,#Fun<bevezeto.sum.1>,...]
8> (hd(F1))({circle, 2}). % külön zárójelezni kell!
12.56636
% hd/1 itt magasabbrendű függvény, zárójelezni kell értékét
```

Magasabb rendű függvények alkalmazása – map, filter

- Magasabb rendű függvény: paramétere vagy eredménye függvény
- Leképzés: `lists:map(Fun, List)` A List lista Fun-nal transzformált elemeiből álló lista

```
9> lists:map(fun erlang:length/1, ["alma", "korte"]).
[4,5] % erlang:length/1: Built-In Function, lista hossza
10> lists:map(Osszeg, [[10,20], [10,20,30]]).
[30,60]
11> L = [{'Diák Detti', [{khf, [...]}]}, {'Lusta Ludvig', []}].
[{'Diák Detti', [{khf, [...]}]}, {'Lusta Ludvig', []}]
12> lists:map(fun(Hallg) -> khf:megfelelt(khf, Hallg) end, L).
[true,false]

• Szűrés: lists:filter(Pred, List)
A List lista Pred-et kielégítő elemeinek listája

13> lists:filter(fun erlang:is_number/1, [x, 10, L, 20, {}]).
[10,20]
14> lists:filter(fun(Hallg) -> khf:megfelelt(khf, Hallg) end, L).
[{'Diák Detti', [{khf, [...]}]}
```

Magasabb rendű függvények alkalmazása – filter példa

- Hányszor szerepel egy elem egy listában? Új megoldásunk:

khf.erl – folytatás

```
% @spec elofordul3(E::term(), L::[term()]) -> N::integer().
% E elem az L listában N-szer fordul elő.
elofordul3(Elem, L) ->
length(lists:filter(fun(X) -> X == Elem end, L)).
```

```
15> khf:elofordul3(prolog, [cekla,prolog,prolog]).
2
```

- A névtelen függvényben felhasználhatjuk az Elem lekötött változót!
- A filter/2 egy lehetséges megvalósítása:


```
filter(_, []) -> [];
filter(P, [Fej|Farok]) -> case P(Fej) of
                            true -> [Fej|filter(P,Farok)];
                            false -> filter(P,Farok)
                        end.
```
- Fejtörő: miért írjuk le kétszer a filter(P, Farok) hívást?

Redukálás a fold függvényekkel

- Jobbról balra haladva: `lists:foldr(Fun, Acc, List)`
 - Balról jobbra haladva: `lists:foldl(Fun, Acc, List)`
 - A List lista elemeiből és az Acc elemből a kétoperandusú Fun-nal képzett érték
- ```
lists:foldr(fun(X, Acc) -> X - Acc end, 0, [1,2,3,4]) ≡ -2
lists:foldl(fun(X, Acc) -> X - Acc end, 0, [1,2,3,4]) ≡ 2
```
- Példa foldr kiértékelési sorrendjére:  $1 - (2 - (3 - (4 - 0))) = -2$   
Példa foldl kiértékelési sorrendjére:  $4 - (3 - (2 - (1 - 0))) = 2$

**% plus(X, Sum) -> X + Sum.**

**R**

```
sum(Acc, []) -> foldr(Fun, Acc, []) ->
 Acc;
 Acc;
sum(Acc, [H|T]) -> foldr(Fun, Acc, [H|T]) ->
 plus(H, sum(Acc, T)). Fun(H, foldr(Fun, Acc, T)).
```

**L**

```
sum(Acc, []) -> foldl(Fun, Acc, []) ->
 Acc;
 Acc;
sum(Acc, [H|T]) -> foldl(Fun, Acc, [H|T]) ->
 sum(plus(H, Acc), T)). foldl(Fun, Fun(H, Acc), T)).
```

## Tartalom

### 3 Erlang alapok

- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- **Műveletek, beépített függvények**
- Ör
- Típusspecifikáció
- Gyakori könyvtári függvények

## Listaműveletek

- Alapműveletek: `hd(L)`, `tl(L)`, `length(L)` (utóbbi lassú:  $O(n)$ !)
- Listák összefűzése ( $As \oplus Bs$ ):  $As++Bs$  vagy `lists:append(As, Bs)`  
 $Cs = As++Bs \leadsto Cs \mapsto$  az As összes eleme a Bs elé fűzve az eredeti sorrendben
- Példa  
`1> [a, 'A', [65]] ++ [1+2, 2/1, 'A'] .`  
`[a, 'A', "A", 3, 2.0, 'A']`
- Listák különbsége:  $As--Bs$  vagy `lists:subtract(As, Bs)`  
 $Cs = As--Bs \leadsto Cs \mapsto$  az As olyan másolata, amelyből ki van hagyva a Bs-ben előforduló összes elem balról számított első előfordulása, feltéve, hogy volt ilyen elem As-ben
- Példa  
`1> [a, 'A', [65], 'A'] -- ["A", 2/1, 'A'] .`  
`[a, 'A']`  
`2> [a, 'A', [65], 'A'] -- ["A", 2/1, 'A', a, a, a] .`  
`['A']`  
`3> [1, 2, 3] -- [1.0, 2] .` **% erős típusosság:  $1 \neq 1.0$**   
`[1, 3]`

## Aritmetikai műveletek

### • Matematikai műveletek

- Előjel:  $+$ ,  $-$  (precedencia: 1)
- Multiplikatív:  $*$ ,  $/$ , `div`, `rem` (precedencia: 2)
- Additív:  $+$ ,  $-$  (precedencia: 3)


### • Bitműveletek

- `bnot`, `band` (precedencia: 2)
- `bor`, `bxor`, `bsl`, `bsr` (precedencia: 3)

### • Megjegyzések

- $+$ ,  $-$ ,  $*$  és  $/$  egész és lebegőpontos operandusokra is alkalmazhatók
- $+$ ,  $-$  és  $*$  eredménye egész, ha mindkét operandusuk egész, egyébként lebegőpontos
- $/$  eredménye mindig lebegőpontos
- `div` és `rem`, valamint a bitműveletek operandusai csak egészek lehetnek

## Relációk

- Termek összehasonlítási sorrendje (v.ö. típusok):  
`number < atom < ref. < fun < port < pid < tuple < list < binary`
- Kisebb-nagyobb reláció  
`<, =<, >=, >`
- Egyenlőségi reláció (aritmetikai egyenlőségre is):  
`==, /=` **ajánlás: helyette azonosan egyenlőt használjunk!**
- Azonosan egyenlő (különbséget tesz integer és float közt):  
`:=, :=/=` Példa: `1> 5.0 := 5.`  
`false`
- Az összehasonlítás eredménye a true vagy a false atom
-  Lebegőpontos értékre kerülendő:  
`1> 10.1 - 9.9 == 0.2.`  
`false`  
`2> 0.0000000000000001 + 1 == 1.`  
`true`  
 Elrettentő példák: `true`
- Kerekítés (`float ↦ integer`), explicit típuskonverzió (`integer ↦ float`):  
`erlang:trunc/1, erlang:round/1, erlang:float/1`

## Logikai műveletek

- Logikai művelet:  
`not, and, or, xor`
- Csak a true és false atomokra alkalmazhatóak
- Lusta kiértékelésű („short-circuit”) logikai művelet:  
`andalso, orelse`
- Példák:  
`1> false and (3 div 0 := 2).`  
`** exception error: bad argument in an arithmetic expression`  
`2> false andalso (3 div 0 := 2).`  
`false`

## Beépített függvények (BIF)

- BIF (Built-in functions)
  - a futtatórendszerbe beépített, rendszerint C-ben írt függvények
  - többségük az **erts**-könyvtár `erlang` moduljának része
  - többször rövid néven (az `erlang:` modulnév nélkül) hívhatók
- Az alaptípusokon alkalmazható leggyakoribb BIF-ek:
  - Számok:  
`abs(Num), trunc(Num), round(Num), float(Num)`
  - Lista:  
`length(List), hd(List), tl(List)`
  - Ennes:  
`tuple_size(Tuple),`  
`element(Index, Tuple),`  
`setelement(Index, Tuple, Value)`  
 Megjegyzés:  $1 \leq \text{Index} \leq \text{tuple\_size}(\text{Tuple})$

## További BIF-ek

- Rendszer:  
`date(), time(), erlang:localtime(), halt()`
- Típusvizsgálat
  - `is_integer(Term), is_float(Term),`
  - `is_number(Term), is_atom(Term),`
  - `is_boolean(Term),`
  - `is_tuple(Term), is_list(Term),`
  - `is_function(Term), is_function(Term, Arity)`
- Típuskonverzió
  - `atom_to_list(Atom), list_to_atom(String),`
  - `integer_to_list(Int), list_to_integer(String),`  
`erlang:list_to_integer(String, Base),`
  - `float_to_list(Float), list_to_float(String),`
  - `tuple_to_list(Tuple), list_to_tuple(List)`
- Érdekesség: a BIF-ek mellett megtalálhatóak az operátorok az `erlang` modulban, lásd az `m(erlang). kimenetét`, pl. `fun erlang:'*/2'(3,4).`

## Tartalom

## Örszekvencia (Guard sequence)

## 3 Erlang alapok

- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Ör
- Típusspecifikáció
- Gyakori könyvtári függvények

- Nézzük újra a következő definíciót:

```
fac(0) -> 1;
fac(N) -> N * fac(N-1).
```

- Mi történik, ha megváltoztatjuk a klózik sorrendjét?
- Mi történik, ha -1-re alkalmazzuk?
- És ha 2.5-re?

A baj az, hogy a `fac(N) -> ...` klóz túl általános.

- Megoldás: korlátozzuk a mintaillesztést örszekvencia alkalmazásával

```
fac(0) ->
1;
fac(N) when is_integer(N), N>0 ->
N*fac(N-1).
```

## Ismétlés: függvénydeklaráció, case

- Függvénydeklaráció:

```
fnév(A11, ..., A1m) [when ÖrSz1] -> SzekvenciálisKif1;
...
fnév(An1, ..., Anm) [when ÖrSzn] -> SzekvenciálisKifn.
```

- Feltételes mintaillesztés (case):

```
case Kif of
 Minta1 [when ÖrSz1] -> SzekvenciálisKif1;
 ...
 Mintan [when ÖrSzn] -> SzekvenciálisKifn
end.
```

## Ör, örszekvencia, örkifejezés

- Az örszekvenciával olyan tulajdonságot írunk elő, amit strukturális mintaillesztéssel nem tudunk leírni
- Az örszekvenciát a `when` kulcsszó vezeti be
- Az örszekvenciában előforduló összes változónak *kötöttnek* kell lennie

Elnevezések:

- **Örkifejezés (Guard expression):** korlátozott Erlang-kifejezés, mellékhatás nélküli
- **Ör (Guard):** egyetlen örkifejezés vagy örkifejezések vesszővel (,) elválasztott sorozata
  - `true`, ha az összes örkifejezés `true` (ÉS-kapcsolat)
  - Ha értéke `true`  $\leadsto$  *sikerül*, bármely más term  $\leadsto$  *meghiúsul*
- **Örszekvencia (Guard sequence):** egyetlen ör vagy örök pontosvesszővel (;) elválasztott sorozata
  - `true` (azaz *sikerül*), ha legalább egy ör `true` (VAGY-kapcsolat)
  - Sokszor helytelenül örnek rövidítik; mentség: az ör tipikusan elegendő, örszekvencia ritka

## Örkifejezés, mintaillesztés

Örkifejezés:

- Örkifejezések  $\subset$  Erlang-kifejezések
- Garantáltan mellékhatás nélküli, hatékonyan kiértékelhető
- Vagy sikerül, vagy megghiúsul
- Hibát (kivételt) **nem** jelezhet; ha hibás az argumentuma, megghiúsul

A mintaillesztés lépései klózválasztásnál, case-nél:

- Strukturális mintaillesztés (hasonló a Prolog illesztésére)
- Örszekvencia kiértékelése

## Örszekvencia: példák

`orok.erl` – *kategoria(V) a V term egy lehetséges osztályozása.*

```
kategoria(V) ->
 case V of
 X when is_atom(X) ->
 atom;
 X when is_number(X), X < 0 ->
 negativ_szam;
 X when is_integer(X) ;
 is_float(X), abs(X-round(X)) < 0.0001 ->
 kerek_szam;
 X when is_list(X), length(X) > 5 ->
 hosszu_lista;
 ...
```

```
2> orok:kategoria(true).
```

```
atom
```

```
3> [{K,orok:kategoria(K)} || K <- [haho, -5, 5.000001, "kokusz"]].
[{haho,atom}, {-5,negativ_szam}, {5.000001,kerek_szam},
 {"kokusz",hosszu_lista}]
```

## Örkifejezés

Örkifejezés lehet:

- Term (vagyis konstans érték)
- Kötött változó
- Örkifejezésekből aritmetikai, összehasonlító és logikai műveletekkel felépített kifejezés
- Örkifejezéseket tartalmazó ennes vagy lista
- Bizonyos BIF-ek örkifejezéssel paraméterezve:
  - Típust vizsgáló predikátumok (`is_TÍPUS`)
  - `abs(Number)` `round(Number)` `trunc(Number)` `float(Term)`
  - `element(N, Tuple)` `tuple_size(Tuple)`
  - `hd(List)` `length(List)` `tl(List)`
  - `bit_size(Bitstring)` `byte_size(Bitstring)` `size(Tuple|Bitstring)`
  - `node()` `node(Pid|Ref|Port)` `self()`

Örkifejezés **nem** lehet:

- Függvényalkalmazás, mert esetleg mellékhatása lehet vagy lassú
- `++ (lists:append/2)`, `-- (lists:subtract/2)`

## Örszekvencia: példák – folytatás

`orok.erl` – *kategoria(V) folytatása*

```
...
{X,Y,Z} when X*X+Y*Y == Z*Z, is_integer(Z) ;
 Z*Z+Y*Y == X*X, is_integer(X) ;
 X*X+Z*Z == Y*Y, is_integer(Y) ->
 pitagoraszi_szamharmas;
{Nev, []} when is_atom(Nev) ->
 talan_hallgato;
{Nev, [{Tipus,_}|_]} when is_atom(Nev), is_atom(Tipus) ->
 talan_hallgato;
[Ny1|_] when Ny1==cekla ; Ny1==prolog ; Ny1==erlang ->
 talan_programozasi_nyelvek_listaja;
{tort, Sz, N} when abs(Sz div N) >= 0 -> % Ha Sz vagy N nem
 racionalis; % egész, vagy ha N:=0, hiba miatt megghiúsul
_ -> egyeb
end.
```

```
4> [orok:kategoria(K) || K <- [{3,5,4}, {'D.D.',[]}, {tort,1,a}]].
[pitagoraszi_szamharmas,talan_hallgato,egyeb]
```



## Feltételes kifejezés őrsekvenciával

- `if`

```

ŐrSz1 -> SzekvenciálisKif1;
...
ŐrSzn -> SzekvenciálisKifn
end.

```
- Kiértékelés: balról jobbra.
- Értéke: az első teljesülő őrsekvencia utáni szekvenciális kifejezés
- Ha nincs ilyen őrsekvencia, futáskor hibát jelez.
- Példák

```

1> X=2.
2> if X<2 -> "<"; X>2 -> ">" end.
** exception error: no true branch...
3> if X<2 -> "<"; X>=2 -> ">=" end.
">="
4> if X<2 -> "<"; true -> ">=" end.
">="

```

### khf.erl – folytatás

```

elofordul4(_, []) -> 0;
elofordul4(E, [Fej|Farok]) ->
 if
 Fej == E -> 1;
 true -> 0
 end
+ elofordul4(E, Farok).

```

## Tartalom

### 3 Erlang alapok

- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Ör
- Típusspecifikáció
- Gyakori könyvtári függvények

## Az if a case speciális esete

- `case`: kifejezést illeszt mintákra őrsekvenciával, `if`: csak őrsekvenciák
- `if` helyettesítése `case`-zel (az Alapértelmezés sora opcionális):

```

case 1 of % _=1 mindig sikeres lenne
_ when ŐrSz1 -> Kif1;
...
_ when ŐrSzn -> Kifn;
-
end
-
-> Alapértelmezés
end

if
 ŐrSz1 -> Kif1;
...
 ŐrSzn -> Kifn;
 true -> Alapért
end

```

- Fordítva: pl. használhatunk-e `case` helyett `if`-et?

```

filter(_, []) -> [];
filter(P, [Fej|Farok]) -> case P(Fej) of
 true -> [Fej|filter(P, Farok)];
 false -> filter(P, Farok)
end.

```

Vigyázat! `if P(Fej) -> Kif...` hibás lenne, őrből nem lehet függvény „illegal guard expression”

## Típusspecifikáció

- Régebben: csak *dokumentációs konvenció*, nem nyelvi elem az Erlangban
  - Mi most ezt tanuljuk
- Újabban: a nyelv része
- Készültek programok a típusspecifikáció és a programkód összevetésére
- A *typeName* típust így jelöljük: `typeName()`.
- Típusok: előre definiált és felhasználó által definiált

## Előre definiált típusok

- `any()`, `term()`: bármely Erlang-típus
- `atom()`, `binary()`, `float()`, `function()`, `integer()`, `pid()`, `port()`, `reference()`: Erlang-alaptípusok
- `bool()`: a `false` és a `true` atomok
- `char()`: az `integer` típus karaktereket ábrázoló része
- `iolist()` = `[char()|binary()|iolist()]`<sup>6</sup>: karakter-io
- `tuple()`: ennestípus
- `list(L)`: `[L]` listatípus szinonimája
- `nil()`: `[]` üreslista-típus szinonimája
- `string()`: `list(char())` szinonimája
- `deep_string()` = `[char()|deep_string()]`
- `none()`: a „nincs típusa” típus; nem befejeződő függvény „eredményének” megjelölésére

<sup>6</sup> ... | ... választási lehetőség a szintaktikai leírásokban.

## Függvénytípus specifikálása

Egy függvény típusát az argumentumainak (formális paramétereinek) és az eredményének (visszatérési értékének) a típusa határozza meg.

- Szintaxis: `@spec funcName(T1,...,Tn) -> Tret.`

- `T1,...,Tn` és `Tret` háromféle lehet:

- `TypeVar`

Típusváltozó, tetszőleges típus jelölésére

- `Type`

Típuskifejezés

- `Var::Type`

Paraméterváltozóval bővítve dokumentációs célra

- Paraméterváltozó: a term részeinek nevet is adhatunk, pl.:

```
% @spec safe_last(Xs::[term()]) -> {ok, X::term()} | error.
% X az Xs lista utolsó eleme.
% @spec split(N::integer(), List::[term()]) ->
% {Prefix::[term()], Suffix::[term()]}
```

## Új (felhasználó által definiált) típusok

- Szintaxis: `@type newType() = Típuskifejezés.`
- Típuskifejezés a `term`, az előre definiált típus, a felhasználó által definiált típus és a típusváltozó
- Uniótípus  
`T1 | T2` típuskifejezés, ha `T1` és `T2` típuskifejezések  
`% @type nyelv() = cekla | prolog | erlang.`
- Listatípus  
`[T]` típuskifejezés, ha `T` típuskifejezés  
`% @type nyelvlista() = [nyelv()].`
- Ennestípus  
`{T1,...,Tn}` típuskifejezés, ha `T1,...,Tn` típuskifejezések  
`% pl. {'Diák Detti', [{khf, [cekla, prolog, prolog]}}] :`  
`% @type hallgato() = {atom(), [{atom(), munka()}]}.`  
`% @type munka() = nyelvlista() | integer() | ...`
- Függvénytípus  
`fun(T1,...,Tn) -> T` típuskifejezés, ha `T1,...,Tn` és `T` típuskifejezések

## Típusspecifikáció: példák

```
@type onOff() = on | off.
@type person() = {person, name(), age()}.
@type people() = [person()].
@type name() = {firstname, string()}.
@type age() = integer().

@spec file:open(FileName, Mode) -> {ok, Handle} | {error, Why}.
@spec file:read_line(Handle) -> {ok, Line} | eof.

@spec lists:map(fun(A) -> B, [A]) -> [B].
@spec lists:filter(fun(X) -> bool(), [X]) -> [X].

@type sspec() = {size(), board()}.
@type size() = integer().
@type board() = [[field()]].
@type field() = [info()].
@type info() = e | o | s | w | integer().
@type ssol() = [[integer()]].
@spec sudoku:sudoku(SSpec::sspec()) -> SSols::[ssol()].
```

## Tartalom

### 3 Erlang alapok

- Bevezetés
- Típusok
- Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabbrendű függvények, függvényérték
- Műveletek, beépített függvények
- Ör
- Típusspecifikáció
- Gyakori könyvtári függvények

## Fűzerkezelő függvények (string modul)

- `len(Str)`, `equal(Str1,Str2)`, `concat(Str1,Str2)`
- `chr(Str,Chr)`, `rchr(Str,Chr)`, `str(Str,SubStr)`, `rstr(Str,SubStr)`  
A karakter / részfüzér első / utolsó előfordulásának indexe, vagy 0, ha nincs benne
- `span(Str,Chrs)`, `cspan(Str,Chrs)`  
Az `Str` ama prefixumának hossza, amelyben kizárólag a `Chars`-beli karakterek fordulnak / nem fordulnak elő
- `substr(Str,Strt,Len)`, `substr(Str,Strt)`  
Az `Str` specifikált részfüzére

## További fűzerkezelő függvények (string modul)

- `tokens(Str,SepList)`  
A `SepList` karakterei mentén fűzerek listájára bontja az `Str`-t
- `join(StrList,Sep)`  
Fűzérré fűzi össze, `Sep`-vel elválasztva, az `StrList` elemeit
- `strip(Str)`, `strip(Str,Dir)`, `strip(Str,Dir,Char)`  
A formázó / `Char` karaktereket levágja a fűzer elejéről / végéről

Részletek és továbbiak: Reference Manual.

## Listakezelő függvények (lists modul)

- `nth(N,Lst)`, `nthtail(N,Lst)`, `last(Lst)`  
A `Lst` `N`-edik karaktere / ott kezdődő farka / utolsó eleme
- `append(Lst1,Lst2) (++)`, `append(LstOfLsts)`  
Az `Lst1` és `Lst2` / `LstOfLsts` elemei egy listába fűzve
- `concat(Lst)`  
Az `Lst` összes eleme fűzérré alakítva és egybefűzve
- `reverse(Lst)`, `reverse(Lst,Tl)`  
Az `Lst` megfordítva / megfordítva a `Tl` elé fűzve (más deklaratív nyelvekben `reverse/2`-nek `revAppend` a neve)
- `flatten(DeepList)`, `flatten(DeepList,Tail)`  
A `DeepList` kisimítva / kisimítva `Tail` elé fűzve
- `max(Lst)`, `min(Lst)`  
Az `Lst` legnagyobb / legkisebb eleme

További listakezelő függvények (`lists` modul)

- `filter(Pred,Lst)`, `delete(Elem,Lst)`  
A `Lst` `Pred`-et kielégítő elemek / `Elem` nélküli másolata
- `takewhile(Pred,Lst)`, `dropwhile(Pred,Lst)`,  
`splitwith(Pred,Lst)`  
Az `Lst` `Pred`-et kielégítő prefixumát tartalmazó / nem tartalmazó másolata; ilyen listákból álló pár
- `partition(Pred,Lst)`, `split(N,Lst)`  
A `Lst` elemei `Pred` / `N` szerint két listába válogatva
- `member(Elem,Lst)`, `all(Pred,Lst)`, `any(Pred,Lst)`  
Igaz, ha `Elem` / `Pred` szerinti minden / `Pred` szerinti legalább egy elem benne van az `Lst`-ben
- `prefix(Lst1,Lst2)`, `suffix(Lst1,Lst2)`  
Igaz, ha az `Lst2` az `Lst1`-gyel kezdődik / végződik

Továbbra is: listakezelő függvények (`lists` modul)

- `sublist(Lst,Len)`, `sublist(Lst,Strt,Len)`  
Az `Lst` 1-től / `Strt`-től kezdődő, `Len` hosszú része
- `subtract(Lst1,Lst2) (--)`  
Az `Lst1` `Lst2` elemeinek első előfordulását nem tartalmazó másolata
- `zip(Lst1,Lst2)`, `unzip(Lst)`  
Az `Lst1` és `Lst2` elemeiből képzett párok listája; az `Lst`-ben lévő párok szétválasztásával létrehozott két lista
- `sort(Lst)`, `sort(Fun,Lst)`  
Az `Lst` alapértelmezés / `Fun` szerint rendezett másolata
- `merge(LstOfLsts)`  
Az `LstOfLsts` listában lévő rendezett listák alapértelmezés szerinti összefuttatása

Még mindig: listakezelő függvények (`lists` modul)

- `merge(Lst1,Lst2)`, `merge(Fun,Lst1,Lst2)`,  
A rendezett `Lst1` és `Lst2` listák alapértelmezés / `Fun` szerinti összefuttatása
- `map(Fun,Lst)`  
Az `Lst` `Fun` szerinti átalakított elemeiből álló lista
- `foreach(Fun,Lst)`  
Az `Lst` elemeire a mellékhatást okozó `Fun` alkalmazása
- `sum(Lst)`  
Az `Lst` elemeinek összege, ha az összes elem számot eredményező kifejezés
- `foldl(Fun,Acc,Lst)`, `foldr(Fun,Acc,Lst)`  
Az `Acc` akkumulátor és az `Lst` elemeinek `Fun` szerinti redukálása, balról jobbra, illetve jobbról balra haladva

Részletek és továbbiak: Reference Manual.

## Néhány további könyvtári modul és függvény

- `math` modul: `pi()`, `sin(X)`, `acos(X)`, `tanh(X)`, `asinh(X)`, `exp(X)`, `log(X)`, `log10(X)`, `pow(X,Y)`, `sqrt(X)`
  - `io` modul: `write([IoDev,]Term)`, `fwrite(Format,`  
`fwrite([IoDev,]Format,Data),nl([IoDev]),format(Format,`  
`format([IoDev,]Format,Data),get_line([IoDev,]Prompt),`  
`read([IoDev,]Prompt)`
  - Formázójelek (`io` modul)
 

|                                                     |                        |
|-----------------------------------------------------|------------------------|
| <code>~c</code>                                     | az adott kódú karakter |
| <code>~f</code> , <code>~e</code> , <code>~g</code> | lebegőpontos szám      |
| <code>~b</code> , <code>~x</code>                   | egész                  |
| <code>~w</code> , <code>~p</code>                   | Erlang-term            |
| <code>~n</code>                                     | újsor                  |
- ```
1> io:format("~s ~b ~c ~f~n", [[1,2,3],1,2,3,math:exp(1)]).
abc 97 b 2.718282
ok
2> X={"abc", [1,2,3], at}, io:format("~p ~w~n", [X,X]).
{"abc", [1,2,3], at} { [97,98,99], [1,2,3], at}
ok
```

IV. rész

Prolog alapok

- 1 Bevezetés
- 2 Cékla: deklaratív programozás C++-ban
- 3 Erlang alapok
- 4 Prolog alapok**
- 5 Keresési feladat pontos megoldása
- 6 Haladó Prolog
- 7 Haladó Erlang

Deklaratív programozási nyelvek

- A matematika függvényfogalmán alapuló **funkcionális** nyelvek: LISP, SML, Haskell, Erlang, ...
- A matematika relációfogalmán alapuló **logikai** nyelvek: Prolog, SQL, Mercury, Kórlátnyelvek (Constraint Programming), ...
- Közös tulajdonságaik
 - Deklaratív szemantika – a program jelentése matematikai állításként olvasható ki.
 - Deklaratív változó \equiv matematikai változó
– *egyetlen* ismeretlen értéket jelöl, vö. egyszeres értékadás
- Jelmondat
 - **WHAT** rather than **HOW**: a **megoldás módja** helyett **inkább** a megoldandó **feladat specifikációját** kell megadni
 - Általában nem elegendő a **specifikáció (WHAT)**; a feladatok (hatékony) megoldásához szükséges a **HOW** rész végiggondolása **is**
 - Mindazonáltal a **WHAT** rész a fontosabb!

Az előadás LP részének áttekintése

Tartalom

- **1. blokk:** A Prolog nyelv alapjai
 - Szintaxis
 - Deklaratív szemantika
 - Procedurális szemantika (végrehajtási mechanizmus)
- **2. blokk:** Prolog programozási módszerek
 - A legfontosabb beépített eljárások
 - Fejlettebb nyelvi és rendszerelemek
- Kitekintés: Új irányzatok a logikai programozásban

- 4** Prolog alapok
 - Prolog bevezetés – néhány példa
 - A Prolog nyelv alapszintaxisa
 - Listakezelő eljárások Prologban
 - Operátorok
 - További vezérlési szerkezetek
 - Prolog végrehajtás – algoritmusok

A Prolog alapelemei: a családi kapcsolatok példája

• Adatok

- Adottak személyekre vonatkozó állítások, pl.

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

• A feladat:

- Definiálandó az unoka–nagyapa kapcsolat, pl. keressük egy adott személy nagyapját.

A nagyszülő feladat — Prolog megoldás

```
% szuloje(Gy, Sz): Gy szülője Sz.
% Tényállításokból álló predikátum
szuloje('Imre', 'Gizella'). % (sz1)
szuloje('Imre', 'István'). % (sz2)
szuloje('István', 'Sarolt'). % (sz3)
szuloje('István', 'Géza'). % (sz4)
szuloje('Gizella',
        'Burgundi Gizella'). % (sz5)
szuloje('Gizella',
        'Civakodó Henrik'). % (sz6)

% ffi(Szemely): Szemely férfi.
ffi('Imre'). ffi('István'). % (f1)-(f2)
ffi('Géza'). % (f3)
ffi('Civakodó Henrik'). % (f4)

% Gyerek nagyszülője Nagyszulo.
% Egyetlen szabályból álló predikátum
nagyszuloje(Gyerek, Nagyszulo) :-
    szuloje(Gyerek, Szulo),
    szuloje(Szulo, Nagyszulo). % (nsz)
```

```
% Ki Imre nagyapja?
| ?- nagyszuloje('Imre', NA),
    ffi(NA).
NA = 'Civakodó Henrik' ? ;
NA = 'Géza' ? ;
no
% Ki Géza unokája?
| ?- nagyszuloje(U, 'Géza').
U = 'Imre' ? ;
no
% Ki Imre nagyszülője?
| ?- nagyszuloje('Imre', NSz).
NSz = 'Burgundi Gizella' ? ;
NSz = 'Civakodó Henrik' ? ;
NSz = 'Sarolt' ? ;
NSz = 'Géza' ? ;
no
```

Deklaratív szemantika – klózek logikai alakja

- A **szabály** jelentése implikáció: a törzsbeli célok **konjunkciójából** következik a fej.

- Példa: $\text{nagyszuloje}(U, N) :- \text{szuloje}(U, Sz), \text{szuloje}(Sz, N).$

- Logikai alak:

$$\forall UNSz (\text{nagyszuloje}(U, N) \leftarrow \text{szuloje}(U, Sz) \wedge \text{szuloje}(Sz, N))$$

- Ekvivalens alak:

$$\forall UN (\text{nagyszuloje}(U, N) \leftarrow \exists Sz (\text{szuloje}(U, Sz) \wedge \text{szuloje}(Sz, N)))$$

- A **tényállítás** feltétel nélküli állítás, pl.

- Példa: $\text{szuloje}('Imre', 'István').$

- Logikai alakja változatlan

- Ebben is lehetnek változók, ezeket is univerzálisan kell kvantálni

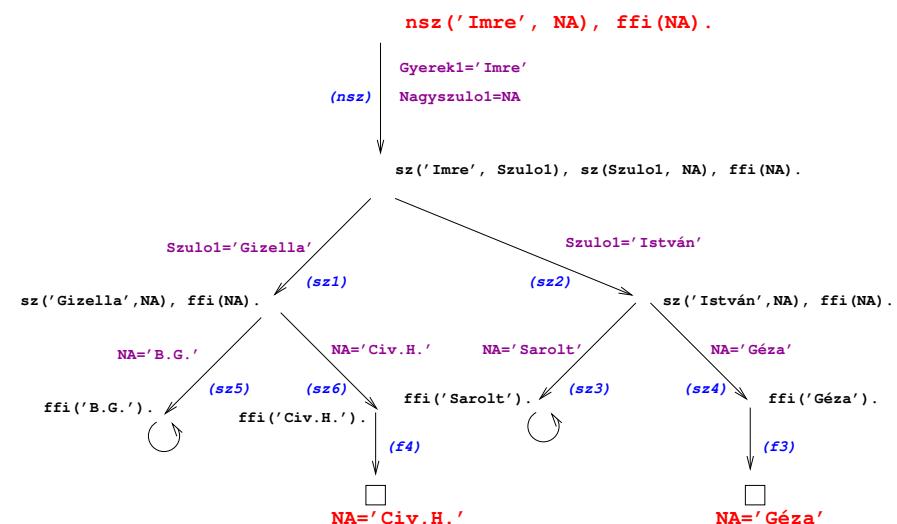
- A **célsorozat** jelentése: keressük azokat a változó-behelyettesítéseket amelyek esetén a célok konjunkciója igaz

- Egy célsorozatra kapott válasz **helyes**, ha az adott behelyettesítésekkel a célsorozat következménye a program logikai alakjának

- A Prolog garantálja a helyességet, de a **teljességet** nem: nem biztos, hogy minden megoldást megkapunk – kaphatunk hibajelzést, végtelen ciklust (végtelen keresési teret) stb.

A nagyszülő példa végrehajtása – keresési tér

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



A Prolog végrehajtás mint logikai következtetés

- A végrehajtáshoz szükséges egy P Prolog program, amely klózik sorozata, valamint egy S_0 kezdeti célsorozat
- A végrehajtás alaplépése: az ún. **redukciós lépés**:
 - bemenetei:
 - egy S_b célsorozat: $c_1, \dots, c_n, n \geq 1$
 - egy $K \in P$ klóz: $f :- d_1, \dots, d_k, k \geq 0$ (tényállítást: $k = 0$) ahol f és c_1 egyesíthetőek, azaz azonos alakra hozhatók, változók behelyettesítésével
 - Jelölés: $\sigma = mgu(a, b)$ az a legáltalánosabb behelyettesítés amelyre $a\sigma = b\sigma$ (**mgu** = **m**ost **g**eneral **u**nifier)
 - kimenete:
 - egy S_k célsorozat: $d_1, \dots, d_k, c_2, \dots, c_n$, hossza: $n - 1 + k \geq 0$, amelyen a σ behelyettesítéseket elvégezzük
- Könnyen végiggondolható, hogy P -ből és S_k -ből **következik** $S_b\sigma$:
 - Ha egy $(\forall x_1, \dots)A$ állítás igaz, akkor az $A\sigma$ állítás is az.
 - Ha $f\sigma = c_1\sigma$, $f\sigma \leftarrow d_1\sigma, \dots, d_k\sigma$, és $S_k\sigma$ igaz akkor $S_b\sigma$ is igaz
- Ha redukciós lépések sorozatával S_0 -ból eljutunk az üres célsorozathoz ($\square \equiv$ igaz) $\implies S_0$ -t igazgá tevő behelyettesítést kapunk.

A Prolog végrehajtási algoritmus – első közelítés

Egy célsorozat végrehajtása

1. Ha az **első** hívás beépített eljárásra vonatkozik, végrehajtjuk a redukciót.
2. Ha az **első** hívás felhasználói eljárásra vonatkozik, akkor megkeressük az eljárás **első** (visszalépés után: következő) olyan klózat, amelynek feje egyesíthető a hívással, és végrehajtjuk a redukciót.
3. Ha nincs egyesíthető fejű klóz, vagy a beépített eljárás megghiúsul, akkor visszalépés következik
 - visszatérünk a legutolsó, felhasználói eljárással történt (sikeres) redukciós lépéshez,
 - annak *bemeneti* célsorozatát megpróbáljuk *újabb* klózzal redukálni – ugrás a 2. lépésre (Ennek megghiúsulása értelemszerűen újabb visszalépést okoz.)
4. Egyébként folytatjuk a végrehajtást 1.-től az új célsorozattal.

A végrehajtás nem „intelligens”

- Pl. `| ?- nagyszuloje(U, 'Géza').` hatékonyabb lenne ha a klóz törzét **jobbról balra** hajtánánk végre
- DE: így a végrehajtás átlátható; a Prolog nem tételbizonyító, hanem programozási nyelv

A Prolog végrehajtás redukciós modellje

Redukciós lépés: egy célsorozat redukálása egy újabb célsorozattá

- egy programklóz segítségével (az első cél felhasználói eljárást hív):
 - A klózt **lemásoljuk**, a változókat szisztematikusan újakra cserélve.
 - A célsorozatot szétbontjuk az első hívásra és a maradékra.
 - Az első hívást **egyesítjük** a klózfejjel
 - Ha az egyesítés nem sikerül, akkor a redukciós lépés is megghiúsul.
 - Sikeres egyesítés esetén az ehhez szükséges behelyettesítéseket elvégezzük a klóz **törzsén** és a **célsorozat** maradékán is
 - Az új célsorozat: a klóztörzs és utána a maradék célsorozat
- egy beépített eljárás segítségével (az első cél beépített eljárást hív):
 - Az első célbeli beépített eljáráshívást végrehajtjuk.
 - Ez lehet sikeres (változó-behelyettesítésekkel), vagy lehet sikertelen.
 - Siker esetén a behelyettesítéseket elvégezzük a célsorozat maradékán, ez lesz az új célsorozat.
 - Ha az eljáráshívás sikertelen, akkor a redukciós lépés megghiúsul.

A Prolog adatfogalma, a Prolog kifejezés

- konstans (**atomic**)
 - számkonstans (**number**) – egész vagy lebegőp, pl. 1, -2.3, 3.0e10
 - névkonstans (**atom**), pl. 'István', szuloje, +, -, <, tree_sum
- összetett- vagy struktúra-kifejezés (**compound**)
 - ún. kanonikus alak: $\langle \text{struktúranév} \rangle (\langle \text{arg}_1 \rangle, \dots, \langle \text{arg}_n \rangle)$
 - a $\langle \text{struktúranév} \rangle$ egy névkonstans, az $\langle \text{arg}_i \rangle$ argumentumok tetszőleges Prolog kifejezések
 - példák: `leaf(1)`, `person(william,smith,2003)`, `<(X,Y), is(X, +(Y,1))`
 - szintaktikus „édesítőszerek”, pl. operátorok: `X is Y+1 \equiv is(X, +(Y,1))`
- változó (**var**)
 - pl. `X`, `Szulo`, `X2`, `_valt`, `_`, `_123`
 - a változó alaphelyzetben behelyettesítetlen, értékkel nem bír, egyesítés során egy tetszőleges Prolog kifejezést (akár egy másik változót) vehet fel értékül
 - ha visszalépünk egy redukciós lépésen keresztül, akkor az ott behelyettesített változók behelyettesítése megszűnik

Aritmetika Prologban – faktoriális

Aritmetikai beépített predikátumok

- X is Kif: A Kif **aritmetikai** kif.-t **kiértékeli** és értékét **egyesíti** x -szel.
- $Kif1 > Kif2$: Kif1 **aritmetikai értéke** nagyobb Kif2 értékénél.
- Hasonlóan: $Kif1 < Kif2$, $Kif1 > Kif2$, $Kif1 = Kif2$, $Kif1 \neq Kif2$ (aritmetikailag egyenlő), $Kif1 \neq Kif2$ (aritmetikailag nem egyenlő)
- Fontos aritmetikai operátorok: +, -, *, /, rem, // (egész-osztás)

A faktoriális függvény definíciója Prologban

- funk. nyelven a faktoriális 1-argumentumú függvény: $Ered = fakt(N)$
- Prologban ennek egy kétargumentumú reláció felel meg: $fakt(N, Ered)$
- Konvenció: az utolsó argumentum(ok) a kimenő paraméter(ek)

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

Adatstruktúrák Prologban – a bináris fák példája

A bináris fa adatstruktúra

- vagy egy csomópont (node), amelynek két részfája van (left, right)
- vagy egy levél (leaf), amely egy egészt tartalmaz

Binárisfa-struktúra C-ben

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

A Prolog dinamikusan típusos nyelv – nincs szükség explicit típusdefinícióra

Mercury típusleírás (komment)

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

A típushoz tartozás ellenőrzése

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

Bináris fák összegzése

- Egy bináris fa levélösszegének kiszámítása:
 - levél esetén a levélben tárolt egész
 - csomópont esetén a két részfa levélösszegének összege

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

% tree_sum(Tree, S):  $\Sigma Tree = S$ .
tree_sum(leaf(Value), Value).
tree_sum(node(Left, Right), S) :-
    tree_sum(Left, S1),
    tree_sum(Right, S2),
    S is S1+S2.
| ?- tree_sum(node(leaf(5),
                  node(leaf(3),
                      leaf(2))), S).

S = 10 ? ;
no
| ?- tree_sum(T, 3).
T = leaf(3) ? ;
! Inst. error in argument 2 of is/2
! goal: 3 is _73+_74
```

Néhány beépített predikátum

Kifejezések egyesítése

- $X = Y$: az X és Y **szimbolikus** kifejezések változók behelyettesítésével azonos alakra hozhatók
- $X \neq Y$: az X és Y kifejezések **nem** hozhatók azonos alakra

További hasznos predikátumok

- true, fail: Mindig sikerül ill. mindig meghiúsul.
- write(X): Az X Prolog kifejezést kiírja.
- write_canonical(X): X kanonikus (alapstruktúra) alakját írja ki.
- nl: Kiír egy újsort.
- trace, notrace: A (teljes) nyomkövetést be- ill. kikapcsolja.

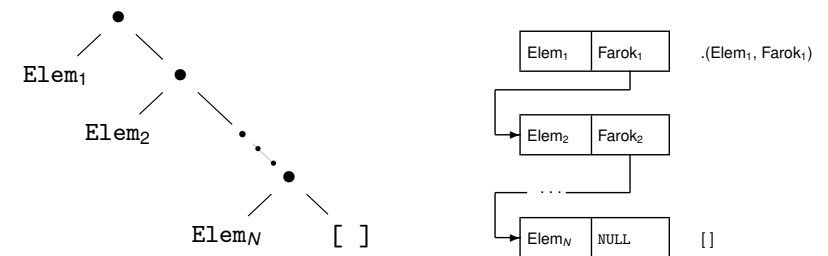
Programfejlesztési beépített predikátumok

- `consult(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.
- `halt`: A Prolog rendszer befejezi működését.

```
> sicstus
SICStus 4.3.2 (x86_64-linux-glibc2.12): Fri May 8 01:05:09 PDT 2015
| ?- consult(tree).
% consulted /home/user/tree.pl in module user, 10 msec 91776 bytes
yes
| ?- tree_sum(node(leaf(3),leaf(2)), S).
S = 5 ? ;
no
| ?- listing(tree).
(...)
yes
| ?- halt.
>
```

A Prolog lista-fogalma

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



Listák jelölése – szintaktikus „édesítőszerek”

- Az alapvető édesítés:
 - `(Fej, Farok)` helyett a `[Fej|Farok]` kifejezést írjuk
- Kiterjesztés N darab „fej”-elemre, a skatulyázás kiküszöbölése:
 - $[Elem_1 | \dots | [Elem_N | Farok] \dots] \Rightarrow [Elem_1, \dots, Elem_N | Farok]$
- Ha a fark `[]`, a „`| []`” jelsorozat elhagyható:
 - $[Elem_1, \dots, Elem_N | []] \Rightarrow [Elem_1, \dots, Elem_N]$

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

Néhány egyszerű listakezelő eljárás

- Egy n -dimenziós vektort egy n -elemű számlistával ábrázolhatunk.
- Írjunk Prolog eljárásokat két vektor összegének, egy vektor és egy skálár (szám) szorzatának, és két vektor skalárszorzatának kiszámítására. Feltételezhető, hogy egy hívásban a vektorok azonos hosszúságúak.

```
% vossz(+A, +B, ?C): C az A és B vektorok összege
v_ossz([], [], []).
v_ossz([A|AL], [B|BL], [C|CL]) :-
    C is A+B,
    vossz(AL, BL, CL).
```

```
% vs_szorz(+A, +S, ?B): B az A vektor S skalárral való szorzata
vs_szorz([], _, []).
vs_szorz([A|AL], S, [B|BL]) :-
    B is A*S, vs_szorz(AL, S, BL).
```

```
% skszorz(+A, +B, ?S): S az A és B vektorok skalárszorzata
skszorz([], [], 0).
skszorz([A|AL], [B|BL], S) :-
    skszorz(AL, BL, S0), S is S0+A*B.
```

Tartalom

4 Prolog alapok

- Prolog bevezetés – néhány példa
- A Prolog nyelv alapszintaxisa**
- Listákezelő eljárások Prologban
- Operátorok
- További vezérlési szerkezetek
- Prolog végrehajtás – algoritmusok

Predikátumok, klózek

• Példa:

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

• Szintaxis:

```
< Prolog program > ::= < predikátum > ...
< predikátum > ::= < klóz > ... {azonos funktorú}
< klóz > ::= < tényállítás > . ⊥ |
           < szabály > . ⊥ {klóz funktora = fej funktora}

< tényállítás > ::= < fej >
< szabály > ::= < fej > :- < törzs >
< törzs > ::= < cél >, ...
< cél > ::= < kifejezés >
< fej > ::= < kifejezés >
```

Prolog kifejezések

• Példa – egy klózfej mint kifejezés:

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

• Szintaxis:

```
< kifejezés > ::= < változó > | {Nincs funktora}
               < konstans > | {Funktora: < konstans >/0}
               < összetett kif. > | {Funktora: < struktúranév >/< arg.sz. >}
               < egyéb kifejezés > | {Operátoros, lista, stb.}

< konstans > ::= < névkonstans > |
               < számkonstans >

< számkonstans > ::= < egész szám > |
                  < lebegőp. szám >

< összetett kif. > ::= < struktúranév > ( < argumentum >, ... )
< struktúranév > ::= < névkonstans >
< argumentum > ::= < kifejezés >
```

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

```
< változó > ::= < nagybetű > < alfanum. jel > ... |
              _ < alfanum. jel > ...
< névkonstans > ::= ' < idézett kar. > ... ' |
                  < kisbetű > < alfanum. jel > ... |
                  < tapadó jel > ... | ! | ; | [ | ] | { | }

< egész szám > ::= {előjeles vagy előjeltelen számjegysorozat}
< lebegőp.szám > ::= {belsejében tizedespontot tartalmazó
                    számjegysorozat esetleges exponenssel}

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

< alfanum. jel > ::= < kisbetű > | < nagybetű > | < számjegy > | _
< tapadó jel > ::= + | - | * | / | \ | $ | ^ | < | > | = | ` | ~ | : | . | ? | @ | # | &
```

Prolog programok formázása

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


```
% predikátumnév(A1, ..., An): A1, ..., An közötti
% összefüggést leíró kijelentő mondat.
```
 - A klózfejet írjuk sor elejére, minden célt lehetőleg külön sorba, néhány szóközzel beljebb kezdve

Összefoglalás: A logikai programozás alapgondolata

- Logikai programozás (LP):
 - Programozás a matematikai logika segítségével
 - egy logikai program nem más mint **logikai állítások halmaza**
 - egy logikai **program futása** nem más mint **következtetési folyamat**
 - De: a logikai következtetés óriási keresési tér bejárását jelenti
 - szorítsuk meg a logika nyelvét
 - válasszunk egyszerű, ember által is követhető következtetési algoritmusokat
 - Az LP máig legelterjedtebb megvalósítása a **Prolog = Programozás logikában (Programming in logic)**
 - az elsőrendű logika egy erősen megszorított résznyelve az ún. **definit-** vagy **Horn-klózk** nyelve,
 - végrehajtási mechanizmusa: **mintaillesztéses** eljáráshíváson alapuló **visszalépéses** keresés.

Erlang és Prolog: néhány eltérés és hasonlóság

Erlang	Prolog
függvény, értéke tetsz. típusú	predikátum, azaz Boole-értékű függvény
arg. bemenő, a fv.érték kimenő	arg.-ok bemenők és kimenők is
egyetlen visszatérési érték	választási pontok, több megoldás lehet
S_1, \dots, S_n szekv. kif., értéke S_n	C_1, \dots, C_n célsor., redukció+visszalépés
külön ennes, lista típusok	a lista is összetett kifejezés
nincsenek felh. operátorok	felh. operátorok definiálhatók
Az = jobb oldalán tömör kif., bal oldalon mintakif.; őrfeltételekkel	az egyesítés szimmetrikus, mindkét oldalon minták

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

Tartalom

- 4 Prolog alapok
 - Prolog bevezetés – néhány példa
 - A Prolog nyelv alapszintaxisa
 - Listákezelő eljárások Prologban
 - Operátorok
 - További vezérlési szerkezetek
 - Prolog végrehajtás – algoritmusok

Listák összefűzése – az `append/3` eljárás

- Ismétlés: Listák összefűzése Céklában:

```
// appf(L1, L2) = L1 ⊕ L2 (L1 és L2 összefűzése)
list appf(const list L1, const list L2) {
    if (L1 == nil) return L2;
    return cons(hd(L1), appf(tl(L1), L2)); }
```
- Írjuk át a kétargumentumú `appf` függvényt `app0/3` Prolog eljárássá!

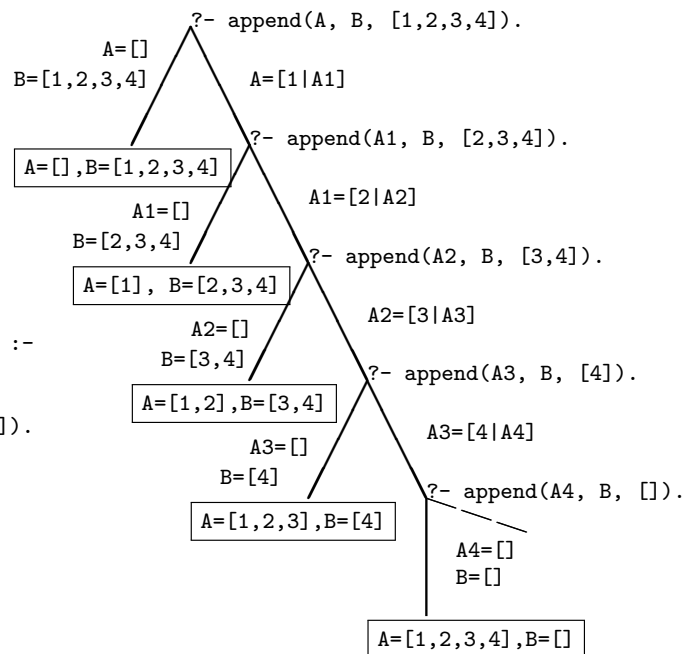
```
app0(L1, L2, Ret) :- L1 = [], Ret = L2.
app0([HD|TL], L2, Ret) :-
    app0(TL, L2, L3), Ret = [HD|L3].
```
- Logikailag tiszta Prolog programokban a `vált` = `Kif` alakú hívások kiküszöbölhetők, ha `vált` minden előfordulását `Kif`-re cseréljük.

```
app([], L2, L2).
app([X|L1], L2, [X|L3]) :- % HD → X, TL → L1 helyettesítéssel
    app(L1, L2, L3).
```
- Az `app... (L1, ...)` komplexitása: a max. futási idő arányos `L1` hosszával
- Miért jobb az `app/3` mint az `app0/3`?
 - `app/3` **jobbrekurzív**, ciklussal ekvivalens (nem fogyaszt vermet)
 - `app([1,...,1000],[0],[2,...])` 1, `app0(...)` 1000 lépésben hiúsul meg.
 - `app/3` használható szétszedésre is (lásd később), míg `app0/3` nem.

Listák szétbontása az `append/3` segítségével

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

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



Lista építése *előlről* – nyílt végű listákkal

- Egy `x` Prolog kif. **nyílt végű lista**, ha `x` változó, vagy `x = [_|Farok]` ahol `Farok` nyílt végű lista.

$$| \text{ ?- } L = [1|_] , L = [_ , 2|_] . \implies L = [1,2|_] ?$$
- A beépített `append/3` azonos az `app/3`-mal:

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).
```
- Az `append` eljárás már az első redukciónál felépíti az eredmény fejét!
 - Célok (pl.): `append([1,2,3], [4], Ered), write(Ered).`
 - Fej: `append([X|L1], L2, [X|L3])`
 - Behelyettesítés: `X = 1, L1 = [2,3], L2 = [4], Ered = [1|L3]`
 - Új célsorozat: `append([2,3], [4], L3), write([1|L3]).`
 (Ered nyílt végű lista, farka még behelyettesítetlen.)
 - A további redukciós lépések behelyettesítése és eredménye:

$$\begin{aligned} L3 &= [2|L3a] && \text{append}([3], [4], L3a), \text{ write}([1|[2|L3a]]). \\ L3a &= [3|L3b] && \text{append}([], [4], L3b), \text{ write}([1,2|[3|L3b]]). \\ L3b &= [4] && \text{write}([1,2,3|[4]]). \end{aligned}$$

Nyílt végű listák az `append` változatokban

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

$$\left\| \begin{aligned} &\text{append}([], L, L). \\ &\text{append}([X|L1], L2, [X|L3]) :- \\ &\quad \text{append}(L1, L2, L3). \end{aligned} \right.$$

- Ha az 1. argumentum zárt végű (n hosszú), mindkét változat legfeljebb $n + 1$ lépésben egyértelmű választ ad, amely lehet nyílt végű:

$$| \text{ ?- } \text{app0}([1,2], L2, L3). \implies L3 = [1,2|L2] ? ; \text{ no}$$
- A 2. arg.-ot nem bontjuk szét \implies mindegy, hogy nyílt vagy zárt végű
- Ha a 3. argumentum zárt végű (n hosszú), akkor az `append` változat legfeljebb $n + 1$ megoldást ad, max. $\sim 2n$ lépésben (ld. előző dia); tehát:
 - `append(L1, L2, L3)` keresési tere véges, ha `L1` vagy `L3` zárt
- Ha az 1. és a 3. arg. is nyílt, akkor a válaszalmaz csak ∞ sok Prolog kifejezéssel fedhető le, pl.

$$_ \oplus [1] = L (\equiv L \text{ utolsó eleme } 1): L = [1]; [_ , 1]; [_ , _ , 1]; \dots$$
- `app0` szétszedésre nem jó, pl. `app0(L, [1,2], []) \implies ∞ ciklus`, mert redukálva a 2. klózzal \implies `app0(L1, [1,2], L3), [X|L3] = []`.
- Az `append` eljárás jobbrekurzív, hála a logikai változó használatának

Variációk append-re – három lista összefűzése (kiegészítő anyag)

- `append(L1,L2,L3,L123): L1 ⊕ L2 ⊕ L3 = L123`
`append(L1, L2, L3, L123) :-`
`append(L1, L2, L12), append(L12, L3, L123).`
- Lassú, pl.: `append([1,...,100],[1,2,3],[1], L)` 103 helyett 203 lépés!
- Szétszedésre nem alkalmas – végtelen választási pontot hoz létre
- Szétszedésre is alkalmas, hatékony változat
`% L1 ⊕ L2 ⊕ L3 = L123,`
`% ahol vagy L1 és L2, vagy L123 adott (zárt végű).`
`append(L1, L2, L3, L123) :-`
`append(L1, L23, L123), append(L2, L3, L23).`
 - `append(+,+,?,?)` esetén az első `append/3` hívás nyílt végű listát ad:
`| ?- append([1,2], L23, L).` \implies `L = [1,2|L23] ?`
 - Az `L3` argumentum behelyettesítettsége (nyílt vagy zárt végű lista-e) nem számít.

Listák megfordítása

- Naív (négyzetes lépésszámú) megoldás
`% nrev(L) = L megfordítása (Cékla)`
`% nrev(L, R): R = L megfordítása.` `list nrev(const list XL) {`
`nrev([], []).` `if (XL == nil) return nil;`
`nrev([X|L], R) :-` `int X = hd(XL); list L = tl(XL);`
`nrev(L, RL),` `list RL = nrev(L);`
`append(RL, [X], R).` `return append(RL,cons(X,nil)); }`
- Lineáris lépésszámú megoldás
`% revapp(L1, R0, R): L1 megfordítását R0 elé fűzve kapjuk R-t.`
`revapp([], R0, R0).`
`revapp([X|L1], R0, R) :-`
`revapp(L1, [X|R0], R).`
`% reverse(R, L): Az R lista az L megfordítása.`
`reverse(R, L) :- revapp(L, [], R).`
- `revapp`-ban `R0,R` egy akkumulátorpár: eddigi ill. végeredmény
- A `lists` könyvtár tartalmazza a `reverse/2` eljárás definícióját, betöltése:
`:- use_module(library(lists)).`

Listák gyűjtése előlről és hátulról (kiegészítő anyag)

Prolog

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

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

C++

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

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

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

Keresés listában – a `member/2` beépített eljárás

- `member(E, L): E az L lista eleme`
`member(Elem, [Elem|_]).`
`member(Elem, [_|Farok]) :-`
`member(Elem, Farok).`
- Eldöntendő (igen-nem) kérdés:
`| ?- member(2, [1,2,3,2]).` \implies `yes` DE
`| ?- member(2, [1,2,3,2]), R=yes.` \implies `R=yes ? ; R=yes ? ; no`
- Lista elemeinek felsorolása:
`| ?- member(X, [1,2,3]).` \implies `X = 1 ? ; X = 2 ? ; X = 3 ? ; no`
`| ?- member(X, [1,2,1]).` \implies `X = 1 ? ; X = 2 ? ; X = 1 ? ; no`
- Listák közös elemeinek felsorolása – az előző két hívásformát kombinálja:
`| ?- member(X, [1,2,3]),`
`member(X, [5,4,3,2,3]).` \implies `X = 2 ? ; X = 3 ? ; X = 3 ? ; no`
- Egy értéket egy (nyílt végű) lista elemévé tesz, végtelen választás!
`| ?- member(1, L).` \implies `L = [1|_A] ? ; L = [_A,1|_B] ? ;`
`L = [_A,_B,1|_C] ? ; ...`
- A `member/2` keresési tere **véges**, ha 2. argumentuma zárt végű lista.

A member/2 predikátum általánosítása: select/3

- `select(E, Lista, M)`: E-t Listából **pont egyszer** elhagyva marad M.
`select(E, [E|Marad], Marad).` % Elhagyjuk a fejet, marad a farok.
`select(E, [X|Farok], [X|M]) :-` % Marad a fej,
`select(E, Farok, M).` % a farokból hagyunk el elemet.
- Felhasználási lehetőségek:
 - | `?- select(1, [2,1,3,1], L).` % Adott elem elhagyása
 \Rightarrow `L = [2,3,1] ? ; L = [2,1,3] ? ; no`
 - | `?- select(X, [1,2,3], L).` % Akármelyik elem elhagyása
 \Rightarrow `L=[2,3], X=1 ? ; L=[1,3], X=2 ? ; L=[1,2], X=3 ? ; no`
 - | `?- select(3, L, [1,2]).` % Adott elem beszúrása!
 \Rightarrow `L = [3,1,2] ? ; L = [1,3,2] ? ; L = [1,2,3] ? ; no`
 - | `?- select(3, [2|L], [1,2,7,3,2,1,8,9,4]).`
 % Beszúrható-e 3 az [1,...]-ba úgy, hogy [2,...]-t kapjunk?
 \Rightarrow `no`
 - | `?- select(1, [X,2,X,3], L).`
 \Rightarrow `L = [2,1,3], X = 1 ? ; L = [1,2,3], X = 1 ? ; no`
- A lists könyvtárban a fenti módon definiált `select/3` eljárás keresési tere **véges**, ha vagy a 2., vagy a 3. argumentuma zárt végű lista.

Listák permutációja (kiegészítő anyag)

- `perm(Lista, Perm)`: Lista permutációja a Perm lista.
`perm([], []).`
`perm(Lista, [Elso|Perm]) :-`
`select(Elso, Lista, Maradek),`
`perm(Maradek, Perm).`
- Felhasználási példák:
 - | `?- perm([1,2], L).`
 \Rightarrow `L = [1,2] ? ; L = [2,1] ? ; no`
 - | `?- perm([a,b,c], L).`
 \Rightarrow `L = [a,b,c] ? ; L = [a,c,b] ? ; L = [b,a,c] ? ;`
 `L = [b,c,a] ? ; L = [c,a,b] ? ; L = [c,b,a] ? ; no`
 - | `?- perm(L, [1,2]).`
 \Rightarrow `L = [1,2] ? ; végtelen keresési tér`
- Ha `perm/2`-ben az első argumentum ismeretlen, akkor a `select` hívás keresési tere végtelen! Illik jelezni az I/O módokat a fejkomentben:
`% perm(+Lista, ?Perm): Lista permutációja a Perm lista.`
- A lists könyvtár tartalmaz egy kétirányban is működő `permutation/2` eljárást.

Tartalom

4 Prolog alapok

- Prolog bevezetés – néhány példa
- A Prolog nyelv alapszintaxisa
- Listákezelő eljárások Prologban
- Operátorok
- További vezérlési szerkezetek
- Prolog végrehajtás – algoritmusok

Operátor-kifejezések

- Példa: `S is -S1+S2` ekvivalens az `is(S, +(-(S1),S2))` kifejezéssel
- Operátoros kifejezések
`<összetett kif.> ::=`
 - `<struktúranév> (<argumentum>, ...)` {eddig csak ez volt}
 - | `<argumentum> <operátornév> <argumentum>` {infix kifejezés}
 - | `<operátornév> <argumentum>` {prefix kifejezés}
 - | `<argumentum> <operátornév>` {posztfix kifejezés}
 - | `(<kifejezés>)` {zárójeles kif.}
- `<operátornév> ::= <struktúranév>` {ha operátorként lett definiálva}
- Egy vagy több azonos jellemzőjű operátor definiálása
`op(Prio, Fajta, OpNév)` vagy `op(Prio, Fajta, [OpNév1, ..., OpNévn])`, ahol
 - `Prio` (prioritás): 1–1200 közötti egész
 - `Fajta`: az `yfx`, `xfy`, `xfx`, `fy`, `fx`, `yf`, `xf` névkonstansok egyike
 - `OpNévi` (az operátor neve): tetszőleges névkonstans
- Az `op/3` beépített predikátum meghívását általában a programot tartalmazó file elején, *direktívában* helyezzük el:
`:- op(800, xfx, [szuloje,nagyszuloje]). 'Imre' szuloje 'István'.`
- A direktívák a programfile *betöltésekor* azonnal végrehajtódnak.

Operátorok jellemzői

- Egy operátort jellemez a fajtája és prioritása
- A fajta az asszociativitás irányát és az írásmódot határozza meg:

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

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

Operátorok implicit zárójelezése – általános szabályok

- Egy $X \ op_1 \ Y \ op_2 \ Z$ zárójelezése, ahol op_1 és op_2 prioritása n_1 és n_2 :
 - ha $n_1 > n_2$ akkor $X \ op_1 \ (Y \ op_2 \ Z)$;
 - ha $n_1 < n_2$ akkor $(X \ op_1 \ Y) \ op_2 \ Z$; (kisebb prio. \Rightarrow erősebb kötés)
 - ha $n_1 = n_2$ és op_1 jobb-asszociatív (xfy), akkor $X \ op_1 \ (Y \ op_2 \ Z)$;
 - egyébként**, ha $n_1 = n_2$ és op_2 bal-assz. (yfx), akkor $(X \ op_1 \ Y) \ op_2 \ Z$;
 - egyébként szintaktikus hiba
- Érdekes példa: $:- \ op(500, \ xfy, \ +^).$ $\% \ :- \ op(500, \ yfx, \ +).$
 $| \ ?- \ :- \ write((1 \ +^ \ 2) \ + \ 3), \ nl. \ \Rightarrow \ (1 \ +^ \ 2) \ + \ 3$
 $| \ ?- \ :- \ write(1 \ +^ \ (2 \ + \ 3)), \ nl. \ \Rightarrow \ 1 \ +^ \ 2 \ + \ 3$
 tehát: konfliktus esetén az **első** operátor asszociativitása „győz”.
- Alapszabály: egy n prioritású operátor zárójelezetlen operandusként
 - legfeljebb $n - 1$ prioritású operátort fogad el az x oldalon
 - legfeljebb n prioritású operátort fogad el az y oldalon
- A zárójelezett kifejezéseket és az alapstruktúra-alakú kifejezéseket feltétel nélkül elfogadjuk operandusként
- Az alapszabály a prefix és posztfix operátorokra is alkalmazandó

Szabványos, beépített operátorok

Szabványos operátorok

Színkód:

már ismert, új aritmetikai, hamarosan jön

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

További beépített operátorok SICStus Prologban

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

Operátorok – kiegészítő megjegyzések

- A „vessző” jel többféle helyzetben is használható:
 - struktúra-argumentumokat, ill. listaelemeket határol el egymástól
 - 1000 prioritású xfy op. pl.: $(p:-a,b,c) \equiv -(p, ', '(a, ', '(b,c))$
 - A vessző **atom**ként csak a ', ', **határoló**ként csak a ,, **operátor**ként mindkét formában – ', ' vagy , – használható.
 - $:- (p, a,b,c)$ többértelmű: $\stackrel{?}{=} :- (p, (a,b,c))$, $\dots \stackrel{?}{=} :- (p, a,b,c) \dots$
 - Egyértelműsítés: argumentumában vagy listaelemében az 1000-nél \geq prioritású operátort tartalmazó kifejezést **zárójelezni kell**:
 $| \ ?- \ write_canonical((a,b,c)). \ \Rightarrow \ ', '(a, ', '(b,c))$
 $| \ ?- \ write_canonical(a,b,c). \ \Rightarrow \ ! \ write_canonical/3 \ \text{does not exist}$
 - Használható-e ugyanaz a név többféle fajtájú operátorként?
 - Nyilván nem lehet egy operátor egyszerre xfy és xfx is, stb.
 - De pl. a + és - operátorok yfx és fy fajtaival is használhatók
 - A könnyebb elemezhetőség miatt a Prolog szabvány kiköti, hogy
 - operátort operandusként zárójelbe kell tenni, pl. $Comp=(>)$
 - egy operátor nem lehet egyszerre infix és posztfix.
- Sok Prolog rendszer (pl. a SICStus) nem követeli meg ezek betartását

Operátorok törlése, lekérdezése

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

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

| ?- op(500, yfx, +).        => yes
| ?- X = +(a,b).            => X = a+b ? ; no
```

- Az adott pillanatban érvényes operátorok lekérdezése:

```
current_op(Prioritás, Fajta, OpNév)

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

Operátoros példa: polinom behelyettesítési értéke

- Formula: az 'x' névkonstansból és számokból az '+' és '*' operátorokkal felépülő kifejezés.
- A feladat: Egy formula értékének kiszámolása egy adott x érték esetén.

```
% erteke(Kif, X, E): A Kif formula x=X helyen vett értéke E.
erteke(x, X, E) :-
    E = X.
erteke(Kif, _, E) :-
    number(Kif), E = Kif.
erteke(K1+K2, X, E) :-
    erteke(K1, X, E1),
    erteke(K2, X, E2),
    E is E1+E2.
erteke(K1*K2, X, E) :-
    erteke(K1, X, E1),
    erteke(K2, X, E2),
    E is E1*E2.

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

Operátorok felhasználása

- Mire jók az operátorok?

- aritmetikai eljárások kényelmes írására, pl. `X is (Y+3) mod 4`
- szimbolikus kifejezések kezelésére (pl. szimbolikus deriválás)
- klózok leírására (:- és ', ' is operátor), és meta-eljárásoknak való átadására, pl. `asserta((p(X):-q(X),r(X)))`
- eljárásfejek, eljárás hívások olvashatóbbá tételére:

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

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

- adatstruktúrák olvashatóbbá tételére, pl.

```
sav(kén, h*2-s-o*4).
```

- Miért rossz a Prolog operátorfogalma?

- A modularitás hiánya miatt:
- Az operátorok egy globális erőforrást képeznek, ez nagyobb projektben gondot okozhat.

Klasszikus szimbolikus kifejezés-feldolgozás: deriválás

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

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

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

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

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

| ?- deriv(I, 0).
=> no
```

Tartalom

4 Prolog alapok

- Prolog bevezetés – néhány példa
- A Prolog nyelv alapszintaxisa
- Listákezelő eljárások Prologban
- Operátorok
- További vezérlési szerkezetek
- Prolog végrehajtás – algoritmusok

Diszjunkció

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

<pre>% fakt(+N, ?F): F = N!. fakt(0, 1). fakt(N, F) :- N > 0, N1 is N-1, fakt(N1, F1), F is F1*N.</pre>	<pre>fakt(N, F) :- (N = 0, F = 1 ; N > 0, N1 is N-1, fakt(N1, F1), F is F1*N).</pre>
--	--
- A diszjunkciót nyitó zárójel elérésekor választási pont jön létre
 - először a diszjunkciót az első ágára redukáljuk
 - visszalépés esetén a diszjunkciót a második ágára redukáljuk
- Tehát az első ág sikeres lefutása után kilépünk a diszjunkcióból, és az utána jövő célokkal folytatjuk a redukálást
 - azaz a ';' elérésekor a ')' -nél folytatjuk a futást
- A ';' skatulyázható (jobbról-balra) és gyengébben köt mint a ','
- Konvenció: a diszjunkciót *mindig* zárójelbe tesszük, a skatulyázott diszjunkciót és az ágakat feleslegesen nem zárójelezzük. Pl. (a felesleges zárójelek aláhúzva, kiemelve): (p; (q;r)), (a; (b,c);d)

A diszjunkció mint szintaktikus édesítőszert

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

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

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

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

Diszjunkció – megjegyzések (kiegészítő anyag)

- Az egyes klózek 'ÉS' vagy 'VAGY' kapcsolatban vannak?
 - A program klózai **ÉS** kapcsolatban vannak, pl.


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

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


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

 „X Imre szülője” akkor **és csak akkor** ha X = István vagy X = Gizella.
- Az (1) predikátum átalakítható egyetlen, diszjunkciós klózzá:


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

Vö. De Morgan azonosságok: $(A \leftarrow B) \wedge (A \leftarrow C) \equiv (A \leftarrow (B \vee C))$

- Általánosan: tetszőleges predikátum egyklózosá alakítható:
 - a klózeket azonos fejűvé alakítjuk, új változók és =-ek bevezetésével:


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

A megghiúsulós negáció (NF – Negation by Failure)

- A $\backslash +$ Hívás vezérlési szerkezet (vö. \nmid – nem bizonyítható) procedurális szemantikája
 - végrehajtja a Hívás hívást,
 - ha Hívás sikeresen lefutott, akkor megghiúsul,
 - egyébként (azaz ha Hívás megghiúsult) sikerül.
- A $\backslash +$ Hívás futása során Hívás legfeljebb egyszer sikerül
- A $\backslash +$ Hívás sohasem helyettesít be változót
- Példa: Keressünk (adatbázisunkban) olyan gyermekeket, aki **nem** szülő!
- Ehhez negációra van szükségünk, egy megoldás:


```
| ?- sz(X, _Sz), \+ sz(Gy, X). % negált cél  $\equiv \neg(\exists Gy.sz(Gy, X))$ 
 $\Rightarrow$  X = 'Imre' ? ; no
```
- Mi történik ha a két hívást megcseréljük?


```
| ?- \+ sz(Gy, X), sz(X, _Sz). % negált cél  $\equiv \neg(\exists Gy.X.sz(Gy, X))$ 
 $\Rightarrow$  no
```
- $\backslash + H$ deklaratív szemantikája: $\neg\exists\vec{X}(H)$, ahol \vec{X} a H -ban a **hívás pillanatában** behelyettesítetlen változók felsorolását jelöli.


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

Gondok a megghiúsulós negációval

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


```
% nem_szulo(+Sz): adott Sz nem szulo
nem_szulo(Sz) :- \+ szuloje(_, Sz).
```
- További gond: „zárt világ feltételezése” (Closed World Assumption – CWA): ami nem bizonyítható, az nem igaz.


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

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

Példa: együttható meghatározása lineáris kifejezésben

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


```
% egyhat(Kif, E): A Kif lineáris formulában az x együtthatója E.
egyhat(x, 1).
egyhat(Kif, E) :-
    number(Kif), E = 0.
egyhat(K1+K2, E) :-
    egyhat(K1, E1),
    egyhat(K2, E2),
    E is E1+E2.
egyhat(K1*K2, E) :-
    egyhat(K2, E0),
    E is K1*E0.
egyhat(K1*K2, E) :-
    number(K2),
    egyhat(K1, E0),
    E is K2*E0.
```
- A fenti megoldás hibás – többszörös megoldást kaphatunk:


```
| ?- egyhat(((x+1)*3)+x+2*(x+x+3), E).  $\Rightarrow$  E = 8 ? ; no
| ?- egyhat(2*3+x, E).  $\Rightarrow$  E = 1 ? ; E = 1 ? ; no
```
- A többszörös megoldás oka:
 az `egyhat(2*3, E)` hívás esetén a (4) és (5) klóz egyaránt sikeres!

Többszörös megoldások kiküszöbölése

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


```
(...)
egyhat(K1*K2, E) :-
    number(K1), egyhat(K2, E0), E is K1*E0.
egyhat(K1*K2, E) :-
    \+ number(K1),
    number(K2), egyhat(K1, E0), E is K2*E0.
```
 - hatékonyabban, feltételes kifejezéssel:


```
(...)
egyhat(K1*K2, E) :-
    ( number(K1) -> egyhat(K2, E0), E is K1*E0
    ; number(K2), egyhat(K1, E0), E is K2*E0
    ).
```
- A feltételes kifejezés hatékonyabban fut, mert:
 - a feltételt csak egyszer értékeli ki
 - nem hagy választási pontot

Feltételes kifejezések

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

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

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

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

Feltételes kifejezés – példák

- Faktoriális

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

- Jelentése azonos a sima diszjunkciós alakkal (lásd **komment**), de annál hatékonyabb, mert nem hagy maga után választási pontot.

- Szám előjele

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

Feltételes kifejezések (folyt.)

- Procedurális szemantika

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

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

- Többszörös elágaztatás skatulyázott feltételes kifejezésekkel:

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

A kiemelt, aláhúzott zárójelek feleslegesek (a ‘;’ egy xfy írásmódú op.).

- Az egyébként rész elhagyható, alapértelmezése: fail.
- \+ felt ekvivalens alakja: (felt -> fail ; true)

További példa: számintervallum felsorolása

- Soroljuk fel az **N** és **M** közötti egészeket (ahol **N** és **M** maguk is egészek)

```
% between0(+M, +N, -I): M =< I =< N, I egész.
between0(M, N, M) :- M =< N.
between0(M, N, I) :- M < N,
    M1 is M+1, between0(M1, N, I).
```

```
| ?- between0(1, 2, _X), between0(3, 4, _Y), Z is 10*_X+_Y.
Z = 13 ? ; Z = 14 ? ; Z = 23 ? ; Z = 24 ? ; no
```

- A `between0(5,5,I)` hívás választási pontot hagy, optimalizált változat:

```
between(M, N, I) :- ( M > N -> fail
                    ; M =:= N -> I = M
                    ; ( I = M
                    ; M1 is M+1, between(M1, N, I)
                    )
                    ).
```

(A () zárójelpár szintaktikusan felesleges, de az olvasónak jelzi, hogy az „else” ágon egy diszjunkció van.)

- A fenti eljárás (még jobban optimalizálva) elérhető a `between` könyvtárban.

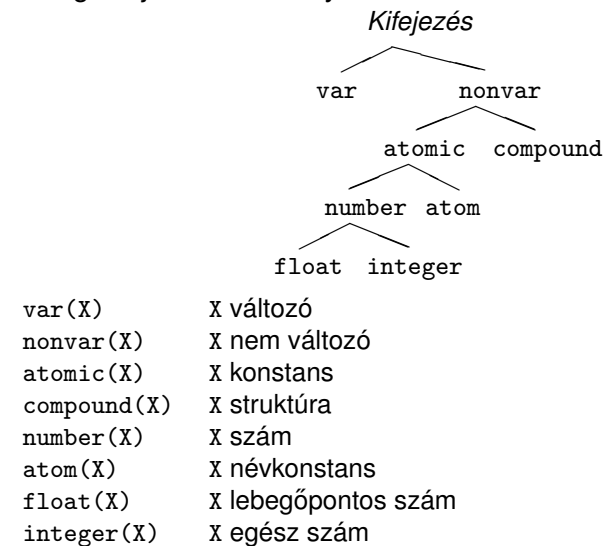
Tartalom

Az egyesítési algoritmus – a Prolog adatfogalma

4 Prolog alapok

- Prolog bevezetés – néhány példa
- A Prolog nyelv alapszintaxisa
- Listákezelő eljárások Prologban
- Operátorok
- További vezérlési szerkezetek
- Prolog végrehajtás – algoritmusok

- Prolog kifejezések osztályozása – kanonikus alak



Az egyesítés célja

- Egyesítés (*unification*): két Prolog kifejezés (pl. egy eljáráshívás és egy klózfej) azonos alakra hozása, változók esetleges behelyettesítésével, a lehető legáltalánosabban (a legkevesebb behelyettesítéssel)
- Az egyesítés **szimmetrikus**: mindkét oldalon lehet – és behelyettesíthető – változó
- Példák
 - Bemenő paraméterátadás – a fej változóit helyettesíti be:


```
hívás:      nagyszuloje('Imre', Nsz),
fej:        nagyszuloje(Gy, N),
behelyettesítés: Gy ← 'Imre', N ← Nsz
```
 - Kimenő paraméterátadás – a hívás változóit helyettesíti be:


```
hívás:      szuloje('Imre', Sz),
fej:        szuloje('Imre', 'István'),
behelyettesítés: Sz ← 'István'
```
 - Kétirányú paraméterátadás – fej- és hívásváltozókat is behelyettesít:


```
hívás:      tree_sum(leaf(5), Sum)
fej:        tree_sum(leaf(V), V)
behelyettesítés: V ← 5, Sum ← 5
```

Az egyesítési algoritmus feladata

- Az egyesítési algoritmus
 - bemenete: két Prolog kifejezés: *A* és *B* (pl. egy klóz fej és egy célsorozat első tagja)
 - feladata: a két kifejezés egyesíthetőségének eldöntése
 - matematikailag az eredménye: megíúsulás, vagy siker és a legáltalánosabb egyesítő – *most general unifier*, *mgu*(*A*, *B*) – előállítása
 - praktikusan nem az *mgu* egyesítő előállítása szükséges, hanem az egyesítő behelyettesítés végrehajtása (a szóbanforgó klóz törzsén és a célsorozat maradékán)
- A legáltalánosabb egyesítő az, amelyik nem helyettesít be „feleslegesen”
 - példa: `tree_sum(leaf(V), V) = tree_sum(T, S)`
 - egy egyesítő behelyettesítés: $V \leftarrow 1, T \leftarrow \text{leaf}(1), S \leftarrow 1$
 - legáltalánosabb egyesítő behelyettesítés: $T \leftarrow \text{leaf}(V), S \leftarrow V$, vagy $T \leftarrow \text{leaf}(S), V \leftarrow S$
 - az *mgu* – változó-átnevezéstől (pl. $V \leftarrow S$) eltekintve – **egyértelmű**
 - minden egyesítő előállítható a legáltalánosabból további behelyettesítéssel, pl. $V \leftarrow 1$ ill. $S \leftarrow 1$

A „praktikus” egyesítési algoritmus

- 1 Ha A és B azonos változók vagy konstansok, akkor kilép sikerrel, behelyettesítés nélkül
 - 2 Egyébként, ha A változó, akkor a $\sigma = \{A \leftarrow B\}$ behelyettesítést elvégzi, és kilép sikerrel
 - 3 Egyébként, ha B változó, akkor a $\sigma = \{B \leftarrow A\}$ behelyettesítést elvégzi, és kilép sikerrel (a 2. és 3. lépések felcserélhetők)
 - 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 , akkor
 - A_1 és B_1 egyesítését elvégzi (beleértve az ehhez szükséges behelyettesítések végrehajtását), ha ez sikertelen, akkor kilép megghiúsulással;
 - A_2 és B_2 egyesítését elvégzi, ha ez sikertelen, akkor kilép megghiúsulással;
 - ...
 - A_N és B_N egyesítését elvégzi, ha ez sikertelen, akkor kilép megghiúsulással
- Kilép sikerrel
- 5 Minden más esetben kilép megghiúsulással (A és B nem egyesíthető)

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 (emiattn. ciklikus kifejezések keletkezhetnek)
 - Szabványos eljárásaként rendelkezésre áll: `unify_with_occurs_check/2`
 - Kiterjesztés (pl. SICStus): az előfordulás-ellenőrzés elhagyása miatt keletkező ciklikus kifejezések tisztességes kezelése.
- Példák:


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

Egyesítési példák a gyakorlatban

- Az egyesítéssel kapcsolatos beépített eljárások:
 - $X = Y$ egyesíti a két argumentumát, megghiúsul, ha ez nem lehetséges.
 - $X \backslash= Y$ sikerül, ha két argumentuma nem egyesíthető, egyébként megghiúsul.
- Példák:


```
| ?- 3+(4+5) = Left+Right.
      Left = 3, Right = 4+5 ?
| ?- node(leaf(X), T) = node(T, leaf(3)).
      T = leaf(3), X = 3 ?
| ?- X*Y = 1+2*3.
      no % mert 1+2*3 ≡ 1+(2*3)
| ?- X*Y = (1+2)*3.
      X = 1+2, Y = 3 ?
| ?- 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 ?
```

Az egyesítési alg. matematikai megfogalmazása (kieg. anyag)

- A *behelyettesítés* egy olyan σ függvény, amely a $Dom(\sigma)$ -beli változókhoz kifejezéseket rendel. Általában posztfix jelölést használunk, pl. $X\sigma$
 - Példa: $\sigma = \{X \leftarrow a, Y \leftarrow s(b, B), Z \leftarrow C\}$, $Dom(\sigma) = \{X, Y, Z\}$, $X\sigma = a$
- A behelyettesítés-függvény természetes módon kiterjeszthető:
 - $K\sigma$: σ alkalmazása egy *tetszőleges* K kifejezésre: σ behelyettesítéseit *egyidejűleg* elvégezzük K -ban.
 - Példa: $f(g(Z, h), A, Y)\sigma = f(g(C, h), A, s(b, B))$
- Kompozíció: $\sigma \otimes \theta = \sigma$ és θ egymás utáni alkalmazása: $X(\sigma \otimes \theta) = X\sigma\theta$
 - A $\sigma \otimes \theta$ behelyettesítés az $x \in Dom(\sigma)$ változókhoz az $(x\sigma)\theta$ kifejezést, a többi $y \in Dom(\theta) \setminus Dom(\sigma)$ változóhoz $y\theta$ -t rendeli ($Dom(\sigma \otimes \theta) = Dom(\sigma) \cup Dom(\theta)$):

$$\sigma \otimes \theta = \{x \leftarrow (x\sigma)\theta \mid x \in Dom(\sigma)\} \cup \{y \leftarrow y\theta \mid y \in Dom(\theta) \setminus Dom(\sigma)\}$$
 - Pl. $\theta = \{X \leftarrow b, B \leftarrow d\}$ esetén $\sigma \otimes \theta = \{X \leftarrow a, Y \leftarrow s(b, d), Z \leftarrow C, B \leftarrow d\}$
- Egy G kifejezés **általánosabb** mint egy S , ha létezik olyan ρ behelyettesítés, hogy $S = G\rho$
 - Példa: $G = f(A, Y)$ általánosabb mint $S = f(1, s(Z))$, mert $\rho = \{A \leftarrow 1, Y \leftarrow s(Z)\}$ esetén $S = G\rho$.

A legáltalánosabb egyesítő előállítás (kiegészítő anyag)

- A és B kifejezések egyesíthetők ha létezik egy olyan σ behelyettesítés, hogy $A\sigma = B\sigma$. Ezt az $A\sigma = B\sigma$ kifejezést A és B egyesített alakjának nevezzük.
- Két kifejezésnek általában több egyesített alakja lehet.
 - Példa: $A = f(X, Y)$ és $B = f(s(U), U)$ egyesített alakja pl.
 - $K_1 = f(s(a), a)$ a $\sigma_1 = \{X \leftarrow s(a), Y \leftarrow a, U \leftarrow a\}$ behelyettesítéssel
 - $K_2 = f(s(U), U)$ a $\sigma_2 = \{X \leftarrow s(U), Y \leftarrow U\}$ behelyettesítéssel
 - $K_3 = f(s(Y), Y)$ a $\sigma_3 = \{X \leftarrow s(Y), U \leftarrow Y\}$ behelyettesítéssel
- A és B legáltalánosabb egyesített alakja egy olyan C kifejezés, amely A és B minden egyesített alakjánál általánosabb
 - A fenti példában K_2 és K_3 legáltalánosabb egyesített alakok
- **Tétel:** A legáltalánosabb egyesített alak, változó-átnevezéstől eltekintve egyértelmű.
- A és B legáltalánosabb egyesítője egy olyan $\sigma = mgu(A, B)$ behelyettesítés, amelyre $A\sigma$ és $B\sigma$ a két kifejezés legáltalánosabb egyesített alakja. Pl. σ_2 és σ_3 a fenti A és B legáltalánosabb egyesítője.
- **Tétel:** A legáltalánosabb egyesítő, változó-átnevezéstől eltekintve egyértelmű.

A „matematikai” egyesítési algoritmus (kiegészítő anyag)

- Az egyesítési algoritmus
 - bemenete: két Prolog kifejezés: A és B
 - feladata: a két kifejezés egyesíthetőségének eldöntése
 - eredménye: siker esetén az $mgu(A, B)$ legáltalánosabb egyesítő
- A rekurzív egyesítési algoritmus $\sigma = mgu(A, B)$ előállítására
 - 1 Ha A és B azonos változók vagy konstansok, akkor $\sigma = \{\}$ (üres).
 - 2 Egyébként, ha A változó, akkor $\sigma = \{A \leftarrow B\}$.
 - 3 Egyébként, ha B változó, akkor $\sigma = \{B \leftarrow A\}$.
(A (2) és (3) lépések sorrendje felcserélődhet.)
 - 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. ...

akkor $\sigma = \sigma_1 \otimes \sigma_2 \otimes \sigma_3 \otimes \dots$

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

Egyesítési példák (kiegészítő anyag)

- $A = \text{tree_sum}(\text{leaf}(V), V)$, $B = \text{tree_sum}(\text{leaf}(5), S)$
 - (4.) A és B neve és argumentumszáma megegyezik
 - (a.) $mgu(\text{leaf}(V), \text{leaf}(5))$ (4., majd 2. szerint) $= \{V \leftarrow 5\} = \sigma_1$
 - (b.) $mgu(V\sigma_1, S) = mgu(5, S)$ (3. szerint) $= \{S \leftarrow 5\} = \sigma_2$
 - tehát $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{V \leftarrow 5, S \leftarrow 5\}$
- $A = \text{node}(\text{leaf}(X), T)$, $B = \text{node}(T, \text{leaf}(3))$
 - (4.) A és B neve és argumentumszáma megegyezik
 - (a.) $mgu(\text{leaf}(X), T)$ (3. szerint) $= \{T \leftarrow \text{leaf}(X)\} = \sigma_1$
 - (b.) $mgu(T\sigma_1, \text{leaf}(3)) = mgu(\text{leaf}(X), \text{leaf}(3))$ (4., majd 2. szerint) $= \{X \leftarrow 3\} = \sigma_2$
 - tehát $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{T \leftarrow \text{leaf}(3), X \leftarrow 3\}$

Az eljárás-redukciós végrehajtási modell

- A Prolog végrehajtás (ismétlés):
 - egy adott célsorozat futtatása egy adott programra vonatkozóan,
 - eredménye lehet:
 - siker – változó-behelyettesítésekkel
 - megghiúsulás (változó-behelyettesítések nélkül)
- A végrehajtás egy állapota: egy célsorozat
- A végrehajtás kétféle lépésből áll:
 - **redukciós lépés:** egy célsorozat + klóz \rightarrow új célsorozat
 - **visszalépés** (zsákutca esetén): visszatérés a legutolsó választási ponthoz és a **további** (eddig nem próbált) klózzal való redukciós lépések

A redukciós modell alapeleme, a redukciós lépés (ismétlés)

- Redukciós lépés: egy célsorozat redukálása egy újabb célsorozattá
 - egy programklóz segítségével (az első cél felhasználói eljárást hív):
 - A klózt **lemásoljuk**, minden változót szisztematikusan új változóra cserélve.
 - A célsorozatot szétbontjuk az első hívásra és a maradékra.
 - Az első hívást **egyesítjük** a klózfejjel
 - A szükséges behelyettesítéseket elvégezzük a klóz **törzsén** és a **célsorozat** maradékán is
 - Az új célsorozat: a klóztörzs és utána a maradék célsorozat
 - Ha a hívás és a klózfej nem egyesíthető \Rightarrow meghíúsulás
 - egy beépített eljárás segítségével (az első cél beépített eljárást hív):
 - A célsorozatot szétbontjuk az első hívásra és a maradékra.
 - A beépített eljáráshívást végrehajtjuk
 - Siker esetén a behelyettesítéseket elvégezzük a célsorozat maradékán ez lesz az új célsorozat
 - Ha a beépített eljárás hívása sikertelen \Rightarrow meghíúsulás

Az eljárás-redukciós végrehajtási algoritmus

- A végrehajtási algoritmus leírásában használt adatstruktúrák:
 - CS – célsorozat
 - egy verem, melynek elemei $\langle CS, I \rangle$ alakú párok – ahol CS egy célsorozat, I a célsorozat első céljának redukálásához használt legutolsó klóz sorszáma.
- A verem a visszalépést szolgálja: minden választási pontnál a veremre mentjük az aktuális $\langle CS, I \rangle$ párt.
- Visszalépéskor a verem tetejéről leemelünk egy $\langle CS, I \rangle$ párt és a végrehajtás következő lépése: CS redukciója az I+1-edik klózzal.

A Prolog végrehajtási algoritmus

- (Kezdeti beállítások:) A verem üres, $CS :=$ kezdeti célsorozat
- (Beépített elj.): Ha CS első hívása beépített akkor hajtsuk végre a red. lépést.
 - Ha ez sikertelen \Rightarrow 6. lépés.
 - Ha ez sikeres, $CS :=$ a redukciós lépés eredménye, és \Rightarrow 5. lépés.
- (Klózszámláló kezdőértékeze:) $I = 1$.
- (Redukciós lépés:) Tekintsük CS első hívására alkalmazható klózok listáját. Ez indexelés nélkül a predikátum összes klóza lesz, indexelés esetén (lásd 2. Prolog blokk) ennek egy megszürt részsorozata. Tegyük fel, hogy ez a lista N elemű.
 - Ha $I > N \Rightarrow$ 6. lépés.
 - Redukciós lépés a lista I-edik klóza és a CS célsorozat között.
 - Ha ez sikertelen, akkor $I := I+1$, és \Rightarrow 4a. lépés.
 - Ha $I < N$ (nem utolsó), akkor veremljük $\langle CS, I \rangle$ -t.
 - $CS :=$ a redukciós lépés eredménye.
- (Siker:) Ha CS üres, akkor sikeres vég, egyébként \Rightarrow 2. lépés.
- (Sikertelenség:) Ha a verem üres, akkor sikertelen vég.
- (Visszalépés:) Leemeljük a (nem üres) verem tetejéről a $\langle CS, I \rangle$ -párt, $I := I+1$, és \Rightarrow 4. lépés.

A faösszegző program többirányú aritmetikával

- Az korábbi faösszegző program a `tree_sum(T, 3)` hívás esetén hibát jelez az `is/2` hívásnál.
- Az `is` beépített eljárás helyett egy saját `plus` eljárást használva egészek korlátos tartományán megoldható a kétirányú működés.

```
% plus(A, B, C): A+B=C, ahol 0 < A,B,C =< 3 egész számok,
plus(1, 1, 2).      plus(1, 2, 3).      plus(2, 1, 3).
```

```
% tree_sum(Tree, S): A Tree fa leveleiben levő számok összege S.
```

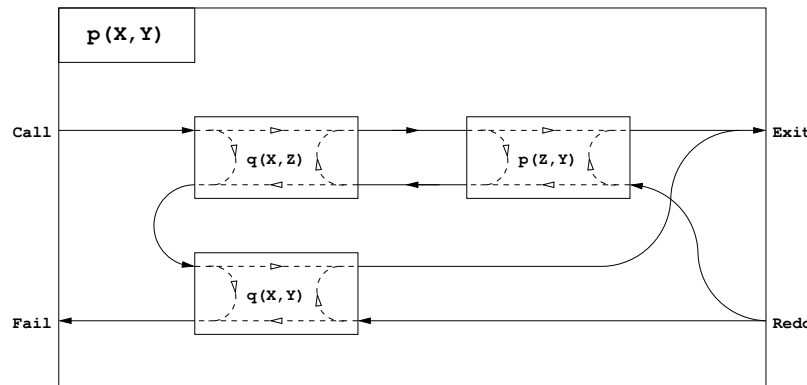
<pre>% tree_sum(+Tree, ?S): tree_sum(leaf(Value), Value). tree_sum(node(Left,Right), S) :- tree_sum(Left, SL), tree_sum(Right, SR), S is SL+SR.</pre>	<pre>% tree_sum(?Tree, ?S): tree_sum(leaf(Value), Value). tree_sum(node(Left,Right), S) :- plus(SL, SR, S), tree_sum(Left, SL), tree_sum(Right, SR).</pre>
---	--

- A jobboldali változat (+,?) módban nagyon kevésbé hatékony :-).

Eljárás-doboz: több klózból álló eljárás

```
p(X,Y) :- q(X,Z), p(Z,Y).
p(X,Y) :- q(X,Y).
```

```
q(1,2). q(2,3). q(2,4).
```



Eljárás-doboz modell – „kapcsolási” alapelvek

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

SICStus nyomkövetés – legfontosabb parancsok

- Beépített eljárások
 - `trace`, `debug`, `zip` – a `c`, `l`, `z` paranccsal indítja a nyomkövetést
 - `notrace`, `nodebug`, `nozip` – kikapcsolja a nyomkövetést
 - `spy(P)`, `nospy(P)`, `nospyall` – töréspont be/ki a `P` eljárásra, \forall ki.
- Alapvető nyomkövetési parancsok, ujjorral (<RET>) kell lezárni
 - `h` (`help`) – parancsok listázása
 - `c` (`creep`) vagy csak <RET> – lassú futás (minden kapunál megáll)
 - `l` (`leap`) – csak töréspontnál áll meg, de a dobozokat építi
 - `z` (`zip`) – csak töréspontnál áll meg, dobozokat nem épít
 - `+` ill. `--` – töréspont be/ki a kurrens eljárásra
 - `s` (`skip`) – eljárástörzs átlépése (Call/Redo \Rightarrow Exit/Fail)
 - `o` (`out`) – kilépés az eljárástörzsből (\Rightarrow szülő Exit/Fail kapu)
- A Prolog végrehajtást megváltoztató parancsok
 - `u` (`unify`) – a kurrens hívást helyettesíti egy egyesítéssel
 - `r` (`retry`) – újrakezdi a kurrens hívás végrehajtását (\Rightarrow Call)
- Információ-megjelenítő és egyéb parancsok
 - `< n` – a kiírási mélységet `n`-re állítja ($n = 0 \Rightarrow \infty$ mélység)
 - `n` (`notrace`) – nyomkövető kikapcsolása
 - `a` (`abort`) – a kurrens futás abbahagyása

Eljárás-doboz modell – OO szemléletben (kiegészítő anyag)

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

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

A p/2 eljárás (225. dia) C++ megfelelőjének „köv. megoldás” metódusa:

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

V. rész

Keresési feladat pontos megoldása

- 1 Bevezetés
- 2 Cékla: deklaratív programozás C++-ban
- 3 Erlang alapok
- 4 Prolog alapok
- 5 Keresési feladat pontos megoldása
- 6 Haladó Prolog
- 7 Haladó Erlang

Tartalom

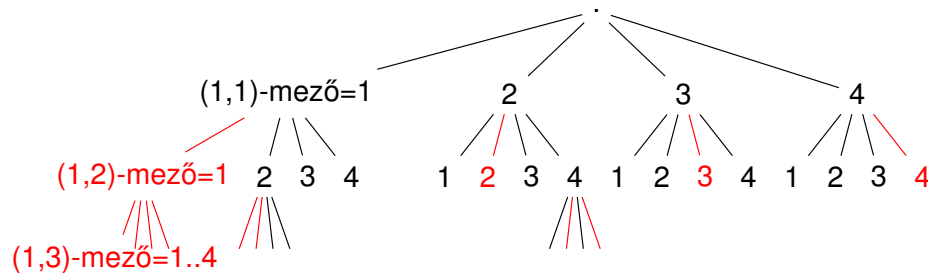
- 5 Keresési feladat pontos megoldása
 - Pontos megoldás funkcionális megközelítésben

Pontos megoldás (Exact solution)

- Kombinatorikában sokszor *optimális megoldás* (optimal solution)
- Egy probléma pontos (egzak) megoldása
- Nem közelítő (approximáció), nem szuboptimális (bizonyos heurisztikák)
- Keresési feladat: valamilyen *értelmezési tartomány* azon elemeit keressük, melyek megfelelnek a kiírt *feltételeknek*
 - lehetséges megoldás = *jelölt*
 - értelmezési tartomány = *keresési tér (search space)*, jelöltek halmaza
 - feltételek = *korlátok vagy kényszerek (constraints)*
- Pl. egy 16 mezős Sudoku-feladvány helyes megoldásai, 8 királynő egy sakktáblán, Hamilton-kör egy gráfban, Imre herceg nagyszülei . . .
- A Prolog végrehajtási algoritmus képes egy predikátumokkal és egy célsorozattal leírt probléma összes megoldását felsorolni (!)
- Funkcionális megközelítésben a megoldások felsorolását a programozónak meg kell írnia (logikaiban is megírható természetesen)

Keresési tér bejárása

- Itt csak véges keresési térrel foglalkozunk
- A megoldás keresését esetekre bonthatjuk, azokat alesetekre stb. \leadsto Ilyenkor egy *keresési fát* járunk be
- Pl. 16 mezős Sudoku (1. sor, 1. oszlop) mezeje lehet 1,2,3,4 Ezen belül (1. sor, 2. oszlop) mezeje lehet 1,2,3,4 stb.



- Bizonyos eseteknél (piros) tudjuk, hogy nem lesz megoldás (ha egy sorban egy érték több mezőben is szerepel)
- Hatékony megoldás: a keresési fa részeit levágjuk (nem járjuk be)

Példa: Send + More = Money

- Feladat: Keressük meg azon (S,E,N,D,M,O,R,Y) számnyolcasokat, melyekre $0 \leq S,E,N,D,M,O,R,Y \leq 9$ és $S,M > 0$, ahol az eltérő betűk eltérő értéket jelölnek, és

```
S E N D
+ M O R E
-----
```

M O N E Y a papíron történő összeadás szabályai szerint, vagyis

$$(1000S + 100E + 10N + D) + (1000M + 100O + 10R + E) = 10000M + 1000O + 100N + 10E + Y.$$

- Naív megoldásunk: járjuk be a teljes keresési teret, és szűrjük meg azon nyolcasokra, melyekre teljesülnek a feltételek
- Keresési tér $\subseteq \{0, 1, \dots, 9\}^8$, azaz egy 8-elemű Descartes-szorzat, mérete 10^8 (10 számjegy 8-adosztályú ismétléses variációi)
- Megoldás:

$$\{(S, E, N, D, M, O, R, Y) \mid S, E, N, D, M, O, R, Y \in \{0..9\}, \text{all_different}, S, M > 0, \text{SEND} + \text{MORE} = \text{MONEY}\}$$

Kimerítő keresés

Exhaustive search, Generate and test, Brute force

- Kimerítő keresés: teljes keresési tér bejárása, jelöltek szűrése

sendmoney.erl – Send More Money megoldások, alapfogalmak

```
% @type d() = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.
```

```
% @type octet() = {d(),d(),d(),d(),d(),d(),d(),d()}.
```

```
% @spec num(Ns::[d()]) -> N::integer().
```

```
% Az Ns számjegylista decimális számként N.
```

```
num(Ns)-> lists:foldl(fun(X,E) -> E*10+X end, 0, Ns).
```

```
% @spec check_sum(octet()) -> bool().
```

```
% A jelölt teljesíti-e az összeadási feltételt.
```

```
check_sum({S,E,N,D,M,O,R,Y}) ->
```

```
    Send = num([S,E,N,D]),
```

```
    More = num([M,O,R,E]),
```

```
    Money = num([M,O,N,E,Y]),
```

```
    Send+More == Money.
```

Kimerítő keresés – folytatás

sendmoney.erl – folytatás

```
% @spec all_different(Xs::[any()]) -> B::bool()
```

```
all_different(L) -> length(L) == length(lists:usort(L)).
```

```
% @spec smm0() -> [octet()].
```

```
smm0() -> Ds = lists:seq(0, 9),
```

```
    [{S,E,N,D,M,O,R,Y} ||
```

```
        S <- Ds,
```

```
        E <- Ds,
```

```
        N <- Ds,
```

```
        D <- Ds,
```

```
        M <- Ds,
```

```
        O <- Ds,
```

```
        R <- Ds,
```

```
        Y <- Ds,
```

```
        all_different([S,E,N,D,M,O,R,Y]),
```

```
        S > 0, M > 0,
```

```
        check_sum({S,E,N,D,M,O,R,Y})]].
```

G

E

N

E

R

A

T

E

and

T E S T

Keresési fa csökkentése (1)

- 10^8 eset ellenőrzése túl sokáig tart
- Ötlet: korábban, már generálás közben is szűrhetjük az egyezéseket

sendmory.erl – folytatás

% @spec smm1() -> [octet()].

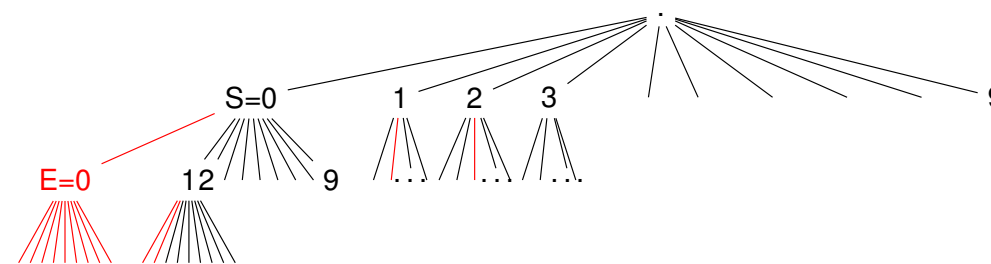
smm1() ->

```

Ds = lists:seq(0, 9),
[ {S,E,N,D,M,O,R,Y} ||
  S <- Ds,
  E <- Ds, E /= S,
  N <- Ds, not lists:member(N, [S,E]),
  D <- Ds, not lists:member(D, [S,E,N]),
  M <- Ds, not lists:member(M, [S,E,N,D]),
  O <- Ds, not lists:member(O, [S,E,N,D,M]),
  R <- Ds, not lists:member(R, [S,E,N,D,M,O]),
  Y <- Ds, not lists:member(Y, [S,E,N,D,M,O,R]),
  S > 0, M > 0,
  check_sum({S,E,N,D,M,O,R,Y}) ].

```

Keresési fa csökkentése (2)



- A keresési fában **azon részfákat, amelyekben egyezés van (pirosak)**, már generálás közben elhagyhatjuk
- Ez már nem kimerítő keresés (nem járjuk be az összes jelöltet)
- A javulást annak köszönhetjük, hogy jelöltek tesztelését előrébb hoztuk
- Vegyük észre, hogy a keresési tér csökkentésével is ide juthatunk: új keresési tér $\subseteq \{10 \text{ elem } 8\text{-adosztályú ismétlés nélküli variációi}\}$
- Mérete $10! / (10 - 8)! = 1\,814\,400 \ll 100\,000\,000$

Variációk felsorolása listanézettel

```

1> Domain = [a,b,c,d].           % A halmaz.
[a,b,c,d]
2> IVar = [ {X,Y,Z} ||           % Ismétléses variációk.
  X <- Domain,
  Y <- Domain,
  Z <- Domain ].
[{a,a,a}, {a,a,b}, {a,a,c}, {a,a,d}, {a,b,a}, {a,b,b}, {...}|...]
3> length(IVar).
64                               % 4*4*4 = 64.
4> INVar = [ {X,Y,Z} ||           % Ismétlés nélküli variációk.
  X <- Domain,
  Y <- Domain -- [X],
  Z <- Domain -- [X,Y] ].
[{a,b,c}, {a,b,d}, {a,c,b}, {a,c,d}, {a,d,b}, {a,d,c},
 {b,a,c},
 {...}|...]
5> length(INVar).
24                               % 4!/1! = 24.

```

Keresési tér csökkentése

- Újból kimerítő keresés, de kisebb a keresési tér

sendmory.erl – folytatás

% @spec smm2() -> [octet()].

% Minden ellenőrzés a generálás után történik.

smm2() ->

```

Ds = lists:seq(0, 9),
[ {S,E,N,D,M,O,R,Y} ||
  S <- Ds -- [],
  E <- Ds -- [S],
  N <- Ds -- [S,E],
  D <- Ds -- [S,E,N],
  M <- Ds -- [S,E,N,D],
  O <- Ds -- [S,E,N,D,M],
  R <- Ds -- [S,E,N,D,M,O],
  Y <- Ds -- [S,E,N,D,M,O,R],
  S > 0, M > 0,
  check_sum({S,E,N,D,M,O,R,Y}) ].

```

Kimerítő keresés újból: keresési tér explicit felsorolása

- Vajon érdemes-e a jelöltek generálását elválasztani a teszteléstől? **Nem!**

sendmory.erl – folytatás

```
% @spec invars() -> [octet()].
% Számjegyek ismétlés nélküli 8-adosztályú variációi
invars() -> Ds = lists:seq(0,9),
             [{S,E,N,D,M,O,R,Y} ||
              S <- Ds -- [],
              E <- Ds -- [S],
              N <- Ds -- [S,E],
              D <- Ds -- [S,E,N],
              M <- Ds -- [S,E,N,D],
              O <- Ds -- [S,E,N,D,M],
              R <- Ds -- [S,E,N,D,M,O],
              Y <- Ds -- [S,E,N,D,M,O,R] ].

% @spec smm3() -> [octet()].
smm3() -> [Sol || {S,_E,_N,_D,M,_O,_R,_Y} = Sol <- invars(),
                 S > 0, M > 0, check_sum(Sol)].
```

Vágások a keresési fában generálás közben

- Ötlet: építsük hátulról a számokat, és ellenőrizzük a részösszegeket még generálás közben

sendmory.erl – folytatás

```
smm5() ->
  Ds = lists:seq(0, 9),
  [{S,E,N,D,M,O,R,Y} ||
   D <- Ds -- [],
   E <- Ds -- [D],
   Y <- Ds -- [D,E]
   (D+E) rem 10 == Y,
   N <- Ds -- [D,E,Y],
   R <- Ds -- [D,E,Y,N],
   (num([N,D])+num([R,E])) rem 100 == num([E,Y]),
   O <- Ds -- [D,E,Y,N,R],
   (num([E,N,D])+num([O,R,E])) rem 1000 == num([N,E,Y]),
   S <- Ds -- [D,E,Y,N,R,O,O],
   M <- Ds -- [D,E,Y,N,R,O,S,O],
   check_sum({S,E,N,D,M,O,R,Y})].
```

Kimerítő keresés újból: keresési tér explicit felsorolása (2)

- Tovább csökkenthető a keresési tér, ha előrébb mozgatunk feltételeket

sendmory.erl – folytatás

```
% @spec smm4() -> [octet()].
% További ellenőrzések generálás közben.
smm4() ->
  Ds = lists:seq(0,9),
  [{S,E,N,D,M,O,R,Y} ||
   S <- Ds -- [0],
   E <- Ds -- [S],
   N <- Ds -- [S,E],
   D <- Ds -- [S,E,N],
   M <- Ds -- [0,S,E,N,D],
   O <- Ds -- [S,E,N,D,M],
   R <- Ds -- [S,E,N,D,M,O],
   Y <- Ds -- [S,E,N,D,M,O,R],
   check_sum({S,E,N,D,M,O,R,Y})].
```

Vágások a keresési fában generálás közben (2)

- A vágások eredményeképpen nagyságrendileg gyorsabb megoldást kapunk
- A generálás minél korábbi fázisában vágunk, annál jobb: a keresési fában nem a legalsó szintről kell *visszalépni*, hogy új megoldást keressünk
- Előzőből ötlet: építsünk részmegoldásokat, és minden építő lépésnél ellenőrizzük, hogy lehet-e értelme a részmegoldást bővíteni megoldássá

sendmory.erl – folytatás

```
% @type partial_solution() = {[d()], [d()], [d()]}.

% @spec smm6() -> [octet()].
smm6() ->
  smm6({[], [], []}, 5, lists:seq(0,9)).
```

- {[], [], []} a kiindulási részmegoldásunk
- 5 méretű megoldásokat kell építeni
- lists:seq(0,9) a változók tartománya

Vágások a keresési fában generálás közben (3)

- Egy PartialSolution = {SendLista, MoreLista, MoneyLista} részmegoldás csak akkor bővíthető megoldássá, ha
 - A listák számjegyei jó pozícióban helyezkednek el: azonos betűk egyeznek, többi számjegy különbözik
 - A részletösszeg is helyes, csak az átvitelben térhet el

sendmoney.erl – folytatás

```
% @spec check_equals(partial_solution()) -> bool().
check_equals(PartialSolution) ->
case PartialSolution of
  {[D], [E], [Y]}          -> all_different([D,E,Y]);
  {[N,D], [R,E], [E,Y]}    -> all_different([N,D,R,E,Y]);
  {[E,N,D], [O,R,E], [N,E,Y]} -> all_different([O,N,D,R,E,Y]);
  {[S,E,N,D], [M,O,R,E], [O,N,E,Y]} -> all_different([S,M,O,N,D,R,E,Y]);
  {[O,S,E,N,D], [O,M,O,R,E], [M,O,N,E,Y]} ->
    all_different([S,M,O,N,D,R,E,Y]) andalso all_different([O,S,M]);
  _ -> false
end.
```

Vágások a keresési fában generálás közben (5)

sendmoney.erl – folytatás

```
% @spec smm6(PS::partial_solution(), Num::integer(),
%           Domain::[integer()]) -> Sols::[octet()].
% Sols az összes megoldás, mely a PS részmegoldásból építhető,
% mérete (Send hossza) <= Num, a számjegyek tartománya Domain.
smm6({Send,_,_} = PS, Num, _Domain) when length(Send) <= Num ->
  {[O,S,E,N,D], [O,M,O,R,E], [M,O,N,E,Y]} = PS,
  {[S,E,N,D,M,O,R,Y]};
smm6({Send,More,Money}, Num, Domain) when length(Send) < Num ->
[Solution ||
  Dsend <- Domain,
  Dmore <- Domain,
  Dmoney <- Domain,
  PSol1 <- [ {[Dsend|Send], [Dmore|More], [Dmoney|Money]} ],
  % pl. így tudunk lekötöni változót: PSol1 <- [ Érték ],
  check_equals(PSol1),
  check_partialsom(PSol1),
  Solution <- smm6(PSol1, Num, Domain) ].
```

Vágások a keresési fában generálás közben (4)

- Egy PartialSolution = {Sendlista, Morelista, Moneylista} részmegoldás csak akkor bővíthető megoldássá, ha
 - A listák számjegyei jó pozícióban helyezkednek el: azonos betűk egyeznek, többi számjegy különbözik
 - A részletösszeg is helyes, csak az átvitelben térhet el

sendmoney.erl – folytatás

```
% @spec check_sum(partial_solution()) -> bool().
% Ellenőrzi, hogy aritmetikai szempontból helyes-e a részmegoldás.
% Az átvittel (carry) nem foglalkozik, mert mindkettő helyes:
% {[1,2],[3,4],[4,6]} és {[9],[2],[1]},
% mert építhető belőlük teljes megoldás.
check_partialsom({Send, More, Money}) ->
  S = num(Send), M = num(More), My = num(Money),
  (S+M) rem round(math:pow(10,length(Send))) == My.
```

Korlát-Kielégítési Probléma (Constraint Satisfaction Problem)

- Eddig előre „könnyen” átlátható keresési fát terveztünk meg, vágunk meg és jártunk be; de a végső cél nem az átlátható keresési fa
- CSP-megközelítés:
 - amíg lehet, szűkítsük a választási lehetőségeket a *korlátok* alapján
 - ha már nem lehet, bontsuk esetekre a választási lehetőségeket

SMM mint CSP = (Változók, Tartományok, Korlátok)

- Változók: S, E, N, D, M, O, R, Y, segédváltozók: 0, C₁, C₂, C₃, C₄

Tartományok:	0	c ₁	c ₂	c ₃	c ₄	s	e	n	d	m	o	r	y
Alsó határ:	0	0	0	0	0	1	0	0	0	1	0	0	0
Felső határ:	0	1	1	1	1	9	9	9	9	9	9	9	9

- Korlátok:

$$\begin{array}{rcl}
 \begin{array}{cccc}
 S & E & N & D \\
 + & M & O & R & E \\
 \hline
 M & O & N & E & Y
 \end{array}
 &
 \begin{array}{l}
 d + e + 0 = y + 10 \cdot c_1 \\
 n + r + c_1 = e + 10 \cdot c_2 \\
 e + o + c_2 = n + 10 \cdot c_3 \\
 s + m + c_3 = o + 10 \cdot c_4 \\
 0 + 0 + c_4 = m + 10 \cdot 0
 \end{array}
 \end{array}$$

CSP tevékenységek – szűkítés

- **Szűkítés egy korlát szerint:** egy korlát egy változójának d_i értéke *felesleges*, ha nincs a korlát többi változójának olyan értékrendszere, amely d_i -vel együtt kielégíti a korlátot
Pl. az utolsó korlát: $0 + 0 + c_4 = m + 10 \cdot 0$, a változók tartománya:
 $0 \in [0]$, $c_4 \in [0,1]$, $m \in [1,2,3,4,5,6,7,8,9]$
 $m \in [2,3,4,5,6,7,8,9]$ értékek feleslegesek!
- Felesleges érték elhagyásával (szűkítéssel) ekvivalens CSP-t kapunk
- SMM kezdeti tartománya; és megszüktíve, tovább már nem szűkíthető:

c1: 01	c1: 01
c2: 01	c2: 01
c3: 01	c3: 01
c4: 01	c4: 1
s: 123456789	s: 89
e: 0123456789	e: 0123456789
n: 0123456789	n: 0123456789
d: 0123456789	d: 0123456789
m: 123456789	m: 1
o: 0123456789	o: 01
r: 0123456789	r: 0123456789
y: 0123456789	y: 0123456789

szűkítés az összes
lehetséges
korláttal, ameddig
sikerül:

CSP tevékenységek – visszalépés

- Ha nincs többértelműség, és a tartományok nem szűkíthetők tovább, két eset lehet:
 - Ha valamely változó tartománya üres, nincs megoldás ezen az ágon
 - Ha minden változó tartománya egy elemű, előállt egy megoldás

Az SMM CSP megoldás folyamata összefoglalva:

1. Felvesszük a változók és segédváltozók tartományait, ez az első *állapotunk* (az állapot egy CSP), ezt betesszük az S listába
2. Ha az S lista üres, megállunk, nincs több megoldás
3. Az S listából kivesszünk egy állapotot, és szűkítjük, ameddig csak lehet
4. Ha van üres tartományú változó, akkor az állapotból nem jutunk megoldáshoz, folytatjuk a 2. lépéssel
5. Ha nincs többértelmű változó az állapotban, az állapot egy megoldás, eltesszük, folytatjuk a 2. lépéssel
6. Valamelyik többértelmű változó tartományát részekre osztjuk, az így keletkező állapotokat visszatesszük a listába, folytatjuk a 2. lépéssel

CSP tevékenységek – címkézés (labeling)

- Tovább már nem szűkíthető CSP esetén vizsgáljuk a többértelműséget:
- Többértelműség: van legalább két elemet tartalmazó tartomány, és egyik tartomány sem üres
- **Címkézés (elágazás):**
 1. kiválasztunk egy többértelmű változót (pl. a legkisebb tartományút),
 2. a tartományt két vagy több részre osztjuk (választási pont),

c1: 01	Két új	c1: 0	c1: 1
c2: 01	CSP-t	c2: 01	c2: 01
c3: 01	készítünk:	c3: 01	c3: 01
c4: 1	c1=0 és	c4: 1	c4: 1
s: 89	c1>0	s: 89	s: 89
e: 0123456789	esetek:	e: 0123456789	e: 0123456789
...	

3. az egyes választásokat mind megoldjuk, mint új CSP-ket.

SMM CSP megoldással – részlet

smm99.erl – SMM CSP megoldásának alapjai

```
% @type state() = {varname(), domain()}.
% @type varname() = any().
% @type domain() = [d()].

% @spec initial_state() -> St::state().
% St describes the variables of the SEND MORE MONEY problem.
initial_state() ->
    VarNames = [0,c1,c2,c3,c4,s,e,n,d,m,o,r,y],
    From      = [0, 0, 0, 0, 0, 1,0,0,0,1,0,0,0],
    To        = [0, 1, 1, 1, 1, 9,9,9,9,9,9,9,9],
    [ {V,lists:seq(F,T)} ||
      {V,{F,T}} <- lists:zip(VarNames, lists:zip(From, To))].

% @spec smm() -> [octet()].
smm() ->
    St = initial_state(),
    process(St, [], []).
```

SMM CSP megoldással – részlet (2)

smm99.erl – SMM CSP megoldásának fő függvénye

```
% process(St0::state(),Sts::[state()],Sols0::[octet()])->Sols::[octet()].
% Sols = Sols1++Sols0 s.t. Sols1 are the sols obtained from [St0|Sts].
process(...) -> ...;
process(St0, Sts, Sols0) ->
    St = narrow_domains(St0),
    DomSizes = [ length(Dom) || {_,Dom} <- St ],
    Max = lists:max(DomSizes),
    Min = lists:min(DomSizes),
    if Min == 0 ->                % there are empty domains
        process(final, Sts, Sols0);
    (St /= St0) ->                % state changed
        process(St, Sts, Sols0);
    Max == 1 ->                  % all domains singletons, solution found
        Sol = [Val || {_,[Val]} <- problem_vars(St)],
        process(final, Sts, [Sol|Sols0]);
    true ->
        {CSt1, CSt2} = make_choice(St), % labeling
        process(CSt1, [CSt2|Sts], Sols0)
end.
```

VI. rész

Haladó Prolog

- 1 Bevezetés
- 2 Cékla: deklaratív programozás C++-ban
- 3 Erlang alapok
- 4 Prolog alapok
- 5 Keresési feladat pontos megoldása
- 6 **Haladó Prolog**
- 7 Haladó Erlang

Haladó Prolog – tartalomjegyzék

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

- Meta-logikai eljárások
- Megoldásgyűjtő eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió, akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Tartalom

A meta-logikai, azaz a logikán túlmutató eljárások fajtái:

6

Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

- A Prolog kifejezések pillanatnyi behelyettesítettségi állapotát vizsgáló eljárások (értelemszerűen logikailag nem tiszták):

- kifejezések osztályozása (1)

```
| ?- var(X) /* X változó? */, X = 1. => X = 1
| ?- X = 1, var(X). => no
```

- kifejezések rendezése (4)

```
| ?- X @< 3 /* X megelőzi 3-t? */, X = 4. => X = 4
% a változók megelőzik a nem változó kifejezéseket
| ?- X = 4, X @< 3. => no
```

- Prolog kifejezéseket szétszedő vagy összerakó eljárások:

- (struktúra) kifejezés \iff név és argumentumok (2)

```
| ?- X = f(alma,körte), X =.. L => L = [f,alma,körte]
```

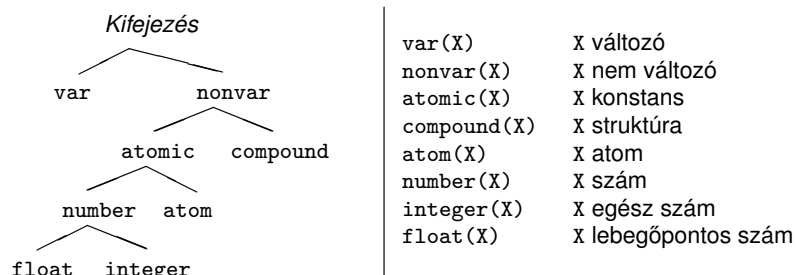
- névkonstansok és számok \iff karaktereik (3)

```
| ?- atom_codes(A, [0'a,0'b,0'a]) => A = aba
```

Kifejezések osztályozása

Osztályozó eljárások: elágaztatás behelyettesítettség alapján

- Kifejezésfajták – osztályozó beépített eljárások (ismétlés)



- SICStus-specifikus osztályozó eljárások:

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

- Az osztályozó eljárások használata – példák

- `var`, `nonvar` – többirányú eljárásokban elágaztatásra
- `number`, `atom`, ... – nem-megkülönböztetett uniók feldolgozása (pl. szimbolikus deriválás)

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

```
% length(?L, ?N): Az L lista N hosszú.
```

```
length(L, N) :- var(N), length(L, 0, N).
```

```
length(L, N) :- nonvar(N), dlength(L, 0, N).
```

```
% length(?L, +IO, -I):
```

```
% Az L lista I-IO hosszú.
```

```
length([], I, I).
```

```
length([_|L], IO, I) :-
```

```
I1 is IO+1,
```

```
length(L, I1, I).
```

```
% dlength(?L, +IO, +I):
```

```
% Az L lista I-IO hosszú.
```

```
dlength([], I, I).
```

```
dlength([_|L], IO, I) :-
```

```
IO<I, I1 is IO+1,
```

```
dlength(L, I1, I).
```

```
| ?- length([1,2], Len). (length/3) => Len = 2 ? ; no
```

```
| ?- length([1,2], 3). (dlength/3) => no
```

```
| ?- length(L, 3). (dlength/3) => L = [_A,_B,_C] ? ;no
```

```
| ?- length(L, Len). (length/3) => L = [], Len = 0 ? ;
L = [_A], Len = 1 ? ; L = [_A,_B], Len = 2 ?
```

Struktúrák szétszedése és összerakása: az *univ* eljárás

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

Példák

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

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**.
 - Az *arg*/3 eljárás így nem csak egy argumentum elővételére, hanem a struktúra változó-argumentumának behelyettesítésére is használható (ld. a 2. példát alább).

Példák:

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

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

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

Struktúrák szétszedése és összerakása: a *functor* eljárás

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

Példák:

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

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, X) :- atomic(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.
    ; EKif = 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(L, EL): EL = { EX | X ∈ L, egysz1(X, EX) }
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)
```

functor/3 és arg/3 alkalmazása: részkifejezések keresése

- A feladat: egy tetszőleges kifejezéshez soroljuk fel a benne levő számokat, és minden szám esetén adjuk meg annak a *kiválasztóját*!
- Egy részkifejezés kiválasztója egy olyan lista, amely megadja, mely argumentumpozíciók mentén juthatunk el hozzá.
- Az $[i_1, i_2, \dots, i_k]$ $k \geq 0$ lista egy *Kif*-ből az i_1 -edik argumentum i_2 -edik argumentumának, ... i_k -edik argumentumát választja ki. (Az [] kiválasztó *Kif*-ből *Kif*-et 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, X, []) :-
    number(X).
kif_szám(X, N, [I|Kiv]) :-
    compound(X), % a var(X) eset kizárása miatt fontos!
    functor(X, _F, ArgNo), between(1, ArgNo, 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
```

Általános kifejezés-bejárás *univ*-val: kiírás

- A feladat: egy tetszőleges kifejezés kiírása úgy, hogy
 - a kétargumentumú operátorok zárójelezett infix formában,
 - minden más alap-struktúra alakban jelenjék meg.

```
ki(Kif) :- compound(Kif), Kif =.. [Func, A1|ArgL],
    ( % kétargumentumú kifejezés, funktora infix operátor
      ArgL = [A2], current_op(_, Kind, Func), infix_fajta(Kind)
    -> write('('), ki(A1),
        write(' '), write(Func), write(' '), ki(A2), write(')')
      ; write(Func), write('('), ki(A1), listaki(ArgL), write(')')
    ).
ki(Kif) :- simple(Kif), write(Kif).

% infix_fajta(F): F egy infix operátorfajta.
infix_fajta(xfx). infix_fajta(xfy). infix_fajta(yfx).

% Az [A1,...,An] listát ",A1,...,An" alakban kiírja.
listaki([]).
listaki([A|AL]) :- write(','), ki(A), listaki(AL).

| ?- ki(f(+a, X*c*X, e)). => f(+a),(( _117 * c) * _117),e
```

Atomok szétszedése és összerakása

- atom_codes/2*: névkonstans és karakterkód-lista közötti átalakítás
 - Hívási minták: *atom_codes(+Atom, ?KódLista)*
atom_codes(-Atom, +KódLista)
 - Jelentése: Atom karakterkódjainak a listája KódLista.
 - Végrehajtása:
 - Ha Atom adott (bemenő), és a $c_1 c_2 \dots c_n$ karakterekből áll, akkor KódLista-t egyesíti a $[k_1, k_2, \dots, k_n]$ listával, ahol k_i a c_i karakter kódja.
 - Ha KódLista egy adott karakterkód-lista, akkor ezekből a karakterekből összerak egy névkonstanst, és azt egyesíti Atom-mal.

- Példák:

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


Atomok szétszedése és összerakása – példák

• Keresés névkonstansokban

% Atom-ban a Rész nem üres részatom kétszer ismétlődik.

```
dadogó_rész(Atom, Rész) :-
```

```
    atom_codes(Atom, Cs),
```

```
    Ds = [_|_],
```

```
    append([_,Ds,Ds,_], Cs),
```

```
    atom_codes(Rész, Ds).
```

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

• Atomok összefűzése

% atom_concat(+A, +B, ?C): A és B névkonstansok összefűzése C.

% (Szabványos beépített eljárás atom_concat(?A, ?B, +C) módban is.)

```
atom_concat(A, B, C) :-
```

```
    atom_codes(A, Ak), atom_codes(B, Bk),
```

```
    append(Ak, Bk, Ck),
```

```
    atom_codes(C, Ck).
```

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

Kifejezések rendezése: szabványos sorrend

- A Prolog szabvány definiálja két tetszőleges Prolog kifejezés sorrendjét.
- Jelölés: $X \prec Y$ – az X kifejezés megelőzi az Y kifejezést.
- A szabványos sorrend definíciója:
 - 1 X és Y azonos $\Leftrightarrow X \prec Y$ és $Y \prec X$ egyike sem igaz.
 - 2 Ha X és Y különböző osztályba tartozik, akkor az osztály dönt:
változó \prec *lebegőpontos szám* \prec *egész szám* \prec *név* \prec *struktúra*.
 - 3 Ha X és Y változó, akkor sorrendjük rendszerfüggő.
 - 4 Ha X és Y lebegőpontos vagy egész szám, akkor $X \prec Y \Leftrightarrow X < Y$.
 - 5 Ha X és Y név, akkor a lexikografikus (abc) sorrend dönt.
 - 6 Ha X és Y struktúrák:
 - 1 Ha X és Y aritása (\equiv argumentumszáma) különböző, akkor
 $X \prec Y \Leftrightarrow X$ aritása kisebb mint Y aritása.
 - 2 Egyébként, ha a struktúrák neve különböző, akkor
 $X \prec Y \Leftrightarrow X$ neve $\prec Y$ neve.
 - 3 Egyébként (azonos név, azonos aritás) balról az első nem azonos argumentum dönt.
- (A SICStus Prologban kiterjesztésként megengedett végtelen (ciklikus) kifejezésekre a fenti rendezés nem érvényes.)

Számok szétszedése és összerakása

• number_codes/2: szám és karakterkód-lista közötti átalakítás

- Hívási minták: `number_codes(+Szám, ?KódLista)`
`number_codes(-Szám, +KódLista)`

- Jelentése: Igaz, ha Szám tizes számrendszerbeli alakja a KódLista karakterkód-listának felel meg.

• Végrehajtása:

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

• Példák:

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

Kifejezések összehasonlítása – beépített eljárások

- Két tetszőleges kifejezés összehasonlítását végző eljárások:

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

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

```
| ?- [1, 2, 3, 4] @< struktúra(1, 2, 3). ==>    sikerül (6.1 szabály)
```

- Lista rendezése: `sort(+L, ?S)`

- Jelentése: az L lista $@<$ szerinti rendezése S ,
 $==/2$ szerint azonos elemek ismétlődését kiszűrve.

```
| ?- sort([a,c,a,b,b,c,c,e,b,d], S).
```

```
S = [a,b,c,d,e] ? ;
```

```
no
```


Összefoglalás: a Prolog egyenlőség-szerű beépített eljárásai

- $U = V$: U egyesítendő V -vel.
Soha sem jelez hibát.
 - $U == V$: U azonos V -vel.
Soha sem jelez hibát és soha sem helyettesít be.
 - $U := V$: Az U és V aritmetikai kifejezések értéke megegyezik.
Hibát jelez, ha U vagy V nem (tömör) aritmetikai kifejezés.
 - $U \text{ is } V$: U egyesítendő a V aritmetikai kifejezés értékével.
Hiba, ha V nem (tömör) aritmetikai kifejezés.
 - $(U =.. V$: U „szétszedettje” a V lista)
- | | |
|--|--|
| ?- $X = 1+2.$ \implies $X = 1+2$ | |
| ?- $3 = 1+2.$ \implies no | |
| ?- $X == 1+2.$ \implies no | |
| ?- $3 == 1+2.$ \implies no | |
| ?- $+(1,2) == 1+2 \implies \text{yes}$ | |
| ?- $X := 1+2.$ \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 | |
| ?- $X \text{ is } 1+2.$ \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 | |
| ?- $1+2 =.. X.$ \implies $X = [+ , 1, 2]$ | |
| ?- $X =.. [f, 1]. \implies X = f(1)$ | |

Összefoglalás: a Prolog nem-egyenlő jellegű beépített eljárásai

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

- $U \backslash= V$: U nem egyesíthető V -vel.
Soha sem jelez hibát.
 - $U \backslash== V$: U nem azonos V -vel.
Soha sem jelez hibát.
 - $U \backslash= V$: Az U és V aritmetikai kifejezések értéke különbözik.
Hibát jelez, ha U vagy V nem (tömör) aritmetikai kifejezés.
- | | |
|---|--|
| ?- $X \backslash= 1+2.$ \implies no | |
| ?- $+(1,2) \backslash= 1+2.$ \implies no | |
| ?- $X \backslash== 1+2.$ \implies yes | |
| ?- $3 \backslash== 1+2.$ \implies yes | |
| ?- $+(1,2) \backslash== 1+2 \implies \text{no}$ | |
| ?- $X \backslash= 1+2.$ \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

		Egyesítés		Azonosság		Aritmetika		
U	V	$U = V$	$U \backslash= V$	$U == V$	$U \backslash== V$	$U := V$	$U \backslash= V$	$U \text{ is } V$
1	2	no	yes	no	yes	no	yes	no
a	b	no	yes	no	yes	error	error	error
1+2	+(1,2)	yes	no	yes	no	yes	no	no
1+2	2+1	no	yes	no	yes	yes	no	no
1+2	3	no	yes	no	yes	yes	no	no
3	1+2	no	yes	no	yes	yes	no	yes
X	1+2	$X=1+2$	no	no	yes	error	error	$X=3$
X	Y	$X=Y$	no	no	yes	error	error	error
X	X	yes	no	yes	no	error	error	error

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

Tartalom

6 Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Keresési feladat Prologban – felsorolás vagy gyűjtés?

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

% Gyűjtés:

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

% Felsorolás:

```
% páros_eleme(L, P): P egy páros
% eleme az L listának.
páros_eleme([X|L], P) :-
    X mod 2 =:= 0, P = X.
páros_eleme([_X|L], P) :-
    % _X akár páros, akár páratlan
    % folytatjuk a felsorolást:
    páros_eleme(L, P).

% egyszerűbb megoldás:
páros_eleme2(L, P) :-
    member(P, L), P mod 2 =:= 0.
```

Gyűjtés és felsorolás kapcsolata

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


```
páros_eleme(L, P) :-
    páros_elemei(L, Pk), member(P, Pk).
```
 - Természetesen ez így nem hatékony!
- Ha adott páros_eleme, hogyan definiálható páros_elemei?
 - Megoldásgyűjtő beépített eljárás segítségével, pl.


```
páros_elemei(L, Pk) :-
    findall(P, páros_eleme(L, P), Pk).
    % páros_eleme(L, P) összes P megoldásának listája Pk.
```
 - a findall/3 beépített eljárás – és társai – az Erlang listanézetéhez hasonlóak, pl.:


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

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

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


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

Lista = { Gyűjtő *másolat* | (∃ X...Z)Cél igaz }

ahol X, ..., Z a findall hívásban levő *szabad változók*.

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

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

- Példa az eljárás használatára:


```
gráf([a-c,a-b,b-c,c-e,b-d]).

| ?- gráf(_G), findall(B, member(A-B, _G), VegP).
    => VegP = [c,b,c,e,d] ? ; no
| ?- gráf(_G), bagof(B, member(A-B, _G), VegPk).
    => A = a, VegPk = [c,b] ? ;
    => A = b, VegPk = [c,d] ? ;
    => A = c, VegPk = [e] ? ; no
```
- Az eljárás végrehajtása (procedurális szemantikája):
 - a Cél kifejezést eljáráshívásként értelmezi, meghívja;
 - összegyűjti a megoldásait (a Gyűjtő-t és a szabad változók behelyettesítéseit);
 - a szabad változók összes behelyettesítését *felsorolja* és mindegyik esetén a Lista-ban megadja az összes hozzá tartozó Gyűjtő értéket.
- A bagof eljárás jelentése (deklaratív szemantikája):

Lista = { Gyűjtő | Cél igaz }, Lista ≠ [].

A bagof megoldásgyűjtő eljárás (folyt.)

- Explicit egzisztenciális kvantorok
 - `bagof(Gyűjtő, V1 ^ ... ^ Vn ^ Cél, Lista)` alakú hívása a V_1, \dots, V_n változókat egzisztenciálisan kvantálnak tekinti, így ezeket nem sorolja fel.
 - jelentése: $Lista = \{ Gyűjtő \mid (\exists V_1, \dots, V_n) Cél \text{ igaz} \} \neq []$.
 $\mid ?- \text{gráf}(_G), \text{bagof}(B, A \text{ member}(A-B, _G), VegP).$
 $\implies VegP = [c,b,c,e,d] ? ; no$
- Egymásba ágyazott gyűjtések
 - szabad változók esetén a `bagof` nemdeterminisztikus lehet, így érdemes lehet skatulyázni:
% A G irányított gráf fokszámlistája FL:
 $FL = \{ A-N \mid N = \{ \{ V \mid A-V \in G \} \}, N > 0 \}$
`fokszámai(G, FL) :-`
`bagof(A-N, Vk(bagof(V, member(A-V, G), Vk),`
`length(Vk, N)), FL).`
 $\mid ?- \text{gráf}(_G), \text{fokszámai}(_G, FL).$
 $\implies FL = [a-2, b-2, c-1] ? ; no$

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, amely az `L0` listát @< szerint rendez, az ismétlődések kiszűrésével, és az eredményt `Lista`-ban adja vissza.
- Példa a `setof/3` eljárás használatára:
`gráf([a-c, a-b, b-c, c-e, b-d]).`
% Gráf egy pontja P.
`pontja(P, Gráf) :- member(A-B, Gráf), (P = A ; P = B).`
% A G gráf pontjainak listája Pk.
`gráf_pontjai(G, Pk) :- setof(P, pontja(P, G), Pk).`
 $\mid ?- \text{gráf}(_G), \text{gráf_pontjai}(_G, Pk).$
 $\implies Pk = [a,b,c,d,e] ? ; no$
 $\mid ?- \text{gráf}(_G), \text{bagof}(P, \text{pontja}(P, _G), Pk).$
 $\implies Pk = [a,c,a,b,b,c,c,e,b,d] ? ; no$

A bagof megoldásgyűjtő eljárás (folyt.)

- Fokszámlista kicsit hatékonyabb előállítás
 - Az előző példában a meta-argumentumban célsorozat szerepelt, ez mindenképpen interpretáltan fut – nevezzük el segédeljárásként
 - A segédeljárás bevezetésével a kvantor is szükségtelenné válik:
% pont_foka(?A, +G, ?N): Az A pont foka a G irányított gráfban N, N>0.
`pont_foka(A, G, N) :-`
`bagof(V, member(A-V, G), Vk), length(Vk, N).`
% A G irányított gráf fokszámlistája FL:
`fokszámai(G, FL) :- bagof(A-N, pont_foka(A, G, N), FL).`
- Példák a `bagof/3` és `findall/3` közötti kisebb különbségekre:
 - $\mid ?- \text{findall}(X, (\text{between}(1, 5, X), X < 0), L). \implies L = [] ? ; no$
 - $\mid ?- \text{bagof}(X, (\text{between}(1, 5, X), X < 0), L). \implies no$
 - $\mid ?- \text{findall}(S, \text{member}(S, [f(X,X), g(X,Y)]), L).$
 $\implies L = [f(_A, _A), g(_B, _C)] ? ; no$
 - $\mid ?- \text{bagof}(S, \text{member}(S, [f(X,X), g(X,Y)]), L).$
 $\implies L = [f(X,X), g(X,Y)] ? ; no$
- A `bagof/3` logikailag tisztább mint a `findall/3`, de időigényesebb!

Tartalom

6 Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Prolog nyelvi eszközök a keresési tér szűkítésére

- Eszközök
 - Az első Prolog rendszerektől kezdve: vágó, szabványos jelölése !
 - Későbbi kiterjesztés: az (if -> then ; else) feltételes szerk.
- Feltételes szerkezet – procedurális szemantika (ismétlés)

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

 - Végrehajtjuk a felt hívást (egy önálló végrehajtási környezetben).
 - Ha felt sikeres \Rightarrow „akkor,folyt” célsorozattal folytatjuk, a felt első megoldása által eredményezett behelyettesítésekkel. A felt cél többi megoldását nem keressük meg!
 - Ha felt meghiúsul \Rightarrow „egyébként,folyt” célsorozattal folytatjuk.
- Feltételes szerkezet – alternatív procedurális szemantika:
 - A feltételes szerkezetet egy speciális diszjunkciónak tekintjük:


```
( felt, {vágás}, akkor
; egyébként
)
```
 - A {vágás} jelentése: megszünteti a felt-beli választási pontokat, és egyébként választását is letiltja.

A vágó eljárás

- A vágó beépített eljárás (!) végrehajtása:
 - 1 letiltja az adott predikátum további klózainak választását,


```
első_poz_elem([X|_], X) :- X > 0, !.
első_poz_elem([X|L], EP) :- X <= 0, első_poz_elem(L, EP).
```
 - 2 megszünteti a választási pontokat az előtte levő eljáráshívásokban.


```
első_poz_elem(L, EP) :- member(EP, L), EP > 0, !.
```
- Miért vágunk le ágakat a keresési térben?
 - Mi tudjuk, hogy nincs megoldás, de a Prolog nem – **zöld** vágó
 - (Például, a legtöbb Prolog megvalósítás „nem tudja”, hogy a $X > 0$ és $X \leq 0$ feltételek kizárják egymást.)
 - Eldobunk megoldásokat – **vörös** vágó, ez a program jelentését megváltoztatja
 - (Vörös vágó lesz a zöldből ha a „felesleges” feltételeket elhagyjuk (pl. az $X \leq 0$ feltételt a fenti 2. klózban)

Feltételes szerkezet: választási pontok a feltételben

- Eddig főleg determinisztikus (választásmentes) feltételeket mutattunk.
- Példafeladat: első_poz_elem(L, P): P az L lista első pozitív eleme.
 - Első megoldás, rekurzióval (**mérnöki**)


```
első_poz_elem([X|_], X) :- X > 0.
első_poz_elem([X|L], EP) :- X <= 0, első_poz_elem(L, EP).
```
 - Második megoldás, visszalépéses kereséssel (**matematikus**)


```
első_poz_elem(L, EP) :-
    append(NemPozL, [EP|_], L), EP > 0,
    \+ van_poz_eleme(NemPozL).

van_poz_eleme(L) :- member(P, L), P > 0.
```
 - Harmadik megoldás, feltételes szerkezettel (**Prolog hekker**)


```
első_poz_elem(L, EP) :-
    ( member(X, L), X > 0 -> EP = X % (1)
    ; fail % ez a sor elhagyható
    ).
```
- Figyelem: a harmadik megoldás épít a member/2 felsorolási sorrendjére!
- Az (1) sorban az EP = X egyenlőség kiküszöbölése esetén első_poz_elem(+,+) módban hibásan működhet!

Példák a vágó eljárás használatára

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

zöld

```
% last(+L, ?E): L utolsó eleme E.
last([E], E) :- !.
last(_|L, Last) :- last(L, Last).
```

zöld

```
% pozitívak(+L, -P): P az L pozitív elemeiből áll.
pozitívak([], []).
pozitívak([E|Ek], [E|Pk]) :-
    E > 0, !,
    pozitívak(Ek, Pk).
pozitívak(_|Ek, Pk) :-
    /* \+ _E > 0, */ pozitívak(Ek, Pk).
```

vörös

Ha nincs **kikommentezve**
akkor **zöld**

Figyelem: a fenti példák nem tökéletesek, hatékonyabb ill. általánosabban használható változatukat később ismertetjük!

A vágó definíciója

- Segédfogalom: egy cél **szülőjének** az őt tartalmazó klóz fejével illesztett hívást nevezzük
 - A 4-kapus modellben a szülő a körülvevő dobozhoz rendelt cél.
 - `Pl. last([E], E) :- !. -` a vágó szülője lehet a `last([7], X)` hívás.
 - A `g` nyomkövető parancs a cél őseit (a szülőt, a szülő szülőjét stb) listázza ki.
- A vágó végrehajtása:
 - mindig sikerül; de mellékhatásként a végrehajtás adott állapotától visszafelé egészen a szülő célig – azt is beleértve – megszünteti a választási pontokat.
- A vágás kétféle választási pontot szüntet meg:


```
r(X) :- s(X), !. % az s(X)-beli választási pontokat – a vágót megelőző
                % cél(ok)nak az első megoldására való megszorítás
r(X) :- t(X). % az r(X) további klózainak választását – a vágót tartalmazó
                % klóz mellett való elköteleződés (commit)
```
- A vágó szemléltetése a 4-kapus doboz modellben: a vágó Redo kapujából a körülvevő (szülő) doboz Fail kapujára megyünk.

A vágó által megszüntetett választási pontok

% vágó nélküli példa

`q(X) :- s(X).`

`q(X) :- t(X).`

% ugyanaz a példa vágóval

`r(X) :- s(X), !.`

`r(X) :- t(X).`

`s(a).`

`s(b).`

`t(c).`

% a vágó nélküli példa futása

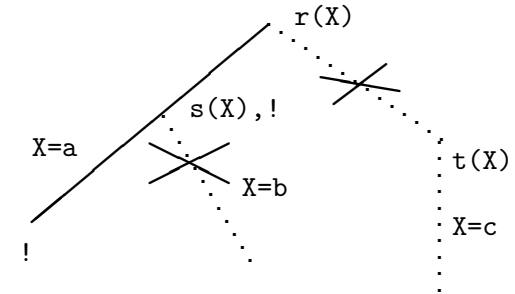
`:- q(X), write(X), fail.`

`--> abc`

% a vágót tartalmazó példa futása

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

`--> a`



A diszjunktív feltételes szerkezet visszavezetése vágóra

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

```
p :-
    aaa,
    ( felt1 -> akkor1
    ; felt2 -> akkor2
    ; ...
    ; egyébként
    ),
    zzzz.

p :-
    aaa, segéd(...), zzzz.

segéd(...) :- felt1, !, akkor1.
segéd(...) :- felt2, !, akkor2.
segéd(...) :- egyébként.
```

- Az egyébként ág elmaradhat, ilyenkor a megfelelő klóz is elmarad.
- A `felt` részekben értelmetlen vágót használni
- Az `akkor` részekben lehet vágó. Ennek hatásköre, a `->` nyílból generált vágóval ellentétben, a teljes `p` predikátum (ilyenkor a Prolog megvalósítás egy speciális, ún. távolbaható vágót használ).
- Vágót rendkívül ritkán szükséges feltételes szerkezetben szerepeltetni.

A vágás első alapesete – klóz mellett való elköteleződés

- A klóz melletti elköteleződés egy 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.
```

 De: a fenti két alak csak akkor ekvivalens, ha `feltétel` egyszerű, nincs benne választás.
- Analógia: ha `a`, `b` és `c` Boole-értékű változók, 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 és fejlesztés

- Vigyázat: a tényleges feltétel részét képezik a fejbeli egyesítések!

```
% abs(X, A): A = |X| (A az X abszolút értéke).
abs(X, X) :- X >= 0, !.          abs(X, A) :- A = X, X >= 0, !.
abs(X, A) :- A is -X.           abs(X, A) :- A is -X.
% a vágó előtt van fej-egyesítés % az egyesítés explicitté téve
```

- 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: csak akkor helyettesíti be a kimenő paramétert, ha már tudja, mi az értéke (nincs „előre-behelyettesítés”, mint a fenti példákban).
- („kimenő” paraméterek** – vágó alkalmazásakor általában nincs többirányú használat :-)

A bevezető példának a vágás alapszabályát betartó változata

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

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

```
% pozitívak(+L, ?Pk): Pk az L pozitív elemeiből áll.
pozitívak([], []).
pozitívak([E|Ek], Pk) :-
    E > 0, !, Pk = [E|Pk0], pozitívak(Ek, Pk0).
pozitívak(_|Ek, Pk) :-
    /* \+ _E > 0, */ pozitívak(Ek, Pk).
```

Megjegyzés: a diszjunktív alakban a feltételek eleve explicitek, nincs fejillesztési probléma, ezért **a diszjunktív feltételes szerkezet használatát javasoljuk a vágó helyett.**

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

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

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

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

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

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

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

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

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

A vágás második alapesete – első megoldásra való megszorítás

- Mikor használjuk az első megoldásra megszorító vágót?
 - behelyettesítést nem okozó, eldöntendő kérdés esetén;
 - feladatspecifikus optimalizálásra (hekkelésre :-);
 - végtelen választási pontot létrehozó eljárások megszelidítésére.
- Eldöntendő kérdés: eljáráshívás csupa bemenő paraméterrel

```
% egy_komponensbeli(+A, +B, +Gráf):
% Az A és B pontok a G gráfnak ugyanabban a komponensében vannak.
egy_komponensbeli(A, B, Graf) :-
    utvonal(A, B, Graf), !.
```

- Eldöntendő kérdés esetén általában nincs értelme többszörös választ adni/várni.

Feladatspecifikus optimalizálás (Kiegészítő anyag)

- Az alábbi példa ugyanazt a módszert használja, mint az `első_poz_elem/2`, csak sokkal bonyolultabb feladatra.
- A feladat: megállapítandó egy lista első fennsíkjának a hossza. (*Fennsíknak* nevezzük egy számlista olyan folytonos, nem üres részlistáját, amelyik pozitív számokból áll és semelyik irányban sem terjeszthető ki.)

```
% Az L lista első fennsíkjának a hossza H.
efhossz(L, H) :-
    append(_NemFennsik, FennsikMaradek, L),
    FennsikMaradek = [X|_], X > 0, !,
    append(Fennsik, Maradek, FennsikMaradek),
    ( Maradek = []
    ; Maradek = [Y|_], Y <= 0
    ), !,
    length(Fennsik, H).
```

- a fenti **diszjunkció** kiváltható egy negációval:

```
\+␣( Maradek = [Y|_], Y > 0 )
```

Végtelen választás megszelidítése: `memberchk`

- A `memberchk/2` beépített eljárás Prolog definíciója:

```
% memberchk(X, L): "X eleme az L listának" kérdés első megoldása.
```

```
% 1. változat                                % 2. ekvivalens változat
memberchk(X, L):-                             memberchk(X, [X|_]) :- !.
    member(X, L), !.                          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

```
szótáraz(Sz):-
    read(M-A), !,
    % A read(X) beépített eljárás egy kifejezést
    % olvas be és egyesíti X-szel
    memberchk(M-A,Sz),
    write(M-A), nl,
    szótáraz(Sz).
szótáraz(_).
```

Egy futása:

```
| ?- szótáraz(Sz).
|: alma-apple.           |: alma-X.
alma-apple               alma-apple
|: korte-pear.           |: X-pear.
korte-pear               korte-pear
                          |: vege.    % nem egyesíthető M-A-val

Sz = [alma-apple,korte-pear|_A] ?
```

Tartalom

6 Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Vezérlési eljárások**
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Vezérlési eljárások, a call/1 beépített eljárás

- Vezérlési eljárás: A Prolog végrehajtáshoz kapcsolódó beépített eljárás.
- A vezérlési eljárások többsége **magasabbrendű** eljárás, azaz olyan eljárás, amely egy vagy több argumentumát eljáráshívásként értelmezi. (A magasabbrendű Prolog eljárásokat szokás **meta-eljárásnak** is hívni.)
- A meta-eljárások fő képviselője a `call(+Cél)`:
 - Cél egy struktúra vagy névkonstans (vö. `callable/1`).
 - Jelentése (deklaratív szemantika): Cél igaz.
 - Hatása (procedurális szemantika): a Cél kifejezést **hívássá alakítja** és végrehajtja.
- A klóztörzsben célként megengedett egy X változó használata, ezt a rendszer egy `call(Modulnév:X)` hívássá alakítja át.

```
| kétszer(X) :- call(X), X.
```

```
| ?- kétszer(write(ba)), nl.    =>    baba
```

```
| ?- listing(kétszer).          =>    kétszer(X) :-
                                     call(user:X), call(user:X).
```

Vezérlési szerkezetek mint eljárások

- A `call/1` argumentumában szerepelhetnek vezérlési szerkezetek is, mert ezek beépített eljárásként is jelen vannak a Prolog rendszerben:
 - `(', ')/2`: konjunkció.
 - `(;)/2`: diszjunkció.
 - `(->)/2`: if-then; `(;)/2`: if-then-else.
 - `(\+)/2`: megghiúsulós negáció.
- A `call`-ban szereplő vezérlési szerkezetek ugyanúgy futnak, mint az interpretált (azaz `consult`-tal betöltött) kód.
- A Cél-beli vágó csak a `call` belsejében vág (szülője a `call(Cél)` hívás).
- `\+` Cél: Cél „nem bizonyítható”. A beépített eljárás definíciója vágóval:


```
\+ X :- call(X), !, fail.
\+ _X.
```
- Példák:


```
| ?- _Cél = (kétszer(write(ba)), write(' ')), kétszer(_Cél), nl.
baba baba
| ?- kétszer((member(X, [a,b,c,d]), write(X), fail ; nl)).
abcd
abcd
```

call/1 példa: futási időt mérő meta-eljárás

```
% Kiírja Goal első megoldásának előállításához vagy a megghiúsuláshoz
% szükséges időt, a Txt szöveg kíséretében.
time(Txt, Goal) :-
    statistics(runtime, [T0,_]), % T0 az indítás óta eltelt CPU idő,
                                % msec-ban (személggyűjtés nélkül).
    (   call(Goal) -> Res = true
    ;   Res = false
    ),
    statistics(runtime, [T1,_]), T is T1-T0,
    format('~w futási idő = ~3d sec\n', [Txt,T]),
    % ~w formázó: kiírás a write/1 segítségével
    % ~3d formázó: I egész kiírása I/1000-ként, 3 tizedesre
    Res = true. % megghiúsul, ha Goal megghiúsult
```

A `call/1` költséges: egy 4472 hosszú lista megfordítása `nrev`-vel (kb. 10 millió `append` hívás), minden `append` körül egy felesleges `call`-al ill. anélkül:

	call nélkül	call-lal	Lassulás
lefordítva	0.47 sec	2.46 sec	5.23
interpretálva	6.97 sec	8.66 sec	1.24

További beépített vezérlési eljárások

- `once(Cél)`: Cél igaz, és csak az első megoldását kérjük. Definíciója:


```
once(X) :- call(X), !.
```

 vagy, feltételes szerkezettel


```
once(X) :- (   call(X) -> true   ).
```
- `true`: azonosan igaz, `fail`: azonosan hamis (mindig megghiúsul).
- `repeat`: végtelen sokszor igaz (végtelen választási pont). Definíciója:


```
repeat.
repeat :- repeat.
```
- A `repeat` eljárást egy mellékhatásos eljárás ismétlésére használhatjuk.
- Példa (egyszerű kalkulátor):


```
bc :- repeat, read(Expr),
    (   Expr = end_of_file -> true
    ;   Res is Expr, write(Expr = Res), nl, fail
    ),
    !.
```
- A végtelen választási pontot kötelező egy vágóval semlegesíteni!

Példa: magasabbrendű reláció definiálása – Kiegészítő anyag

- Az implikáció ($P \Rightarrow Q$) megvalósítása negáció segítségével:

```
% P minden megoldása esetén Q igaz.
forall(P, Q) :-
    \+ (P, \+Q). % Szintaktikus emlékeztető:
                % az első \+ után kötelező a szóköz!

| ?- _L = [1,2,3],
    % _L minden eleme pozitív:
    forall(member(X, _L), X > 0).
true ?
| ?- _L = [1,-2,3], forall(member(X, _L), X > 0).
no
| ?- _L = [1,2,3],
    % _L szigorúan monoton növe:
    forall(append(_, [A|_], _L), A < B).
true ?
```

- forall/2 csak eldöntendő kérdés esetén használható.

Tartalom

6 Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés**
- Jobbrekurzió és akkumulátorok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Determinizmus

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

p(1, a).	?- p(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 és a választásmentesség

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

Egyszerű feltétel

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

Indexelés

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

Indexelés és vágó

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

Választásmentesség feltételes szerkezetek esetén

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

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

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

Indexelés

- Mi az indexelés?
 - egy adott hívásra illeszthető klózek 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 fejjargumentum külső funktora:
 - C szám vagy névkonstans esetén C/0;
 - R nevű és N argumentumú struktúra esetén R/N;
 - változó esetén nem értelmezett.
- Az indexelés megvalósítása:
 - Fordítási időben: funktor \Rightarrow illeszthető fejű klózek részhalmaza.
 - Futási időben: a részhalmaz lényegében konstans idejű kiválasztása (hash tábla használatával).
 - Fontos:** ha egyelemű a részhalmaz, nincs választási pont!

Példa indexelésre

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

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

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

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

Struktúrák, változók a fejjargumentumban

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

```
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).
```

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

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

- A 2. klóz kevésbé hatékony változatai

```
rhossz([X|L], RH) :- length([X|L], H), RH is 1/H.
                    % ^ újra felépíti [X|L]-t.
rhossz(L, RH) :- L \= [], length(L, H), RH is 1/H.
                    % L=[] esetén választási pontot hagy.
```

Indexelés – további tudnivalók

- Indexelés és aritmetika
 - Az indexelés nem foglalkozik aritmetikai vizsgálatokkal.
 - Pl. az $N = 0$ és $N > 0$ feltételek esetén a SICStus Prolog nem veszi figyelembe, hogy ezek kizárják egymást.
 - Az alábbi fakt/2 eljárás lefutása nem-determinisztikus:


```
fakt(0, 1).
fakt(N, F) :- N > 0, N1 is N-1, fakt(N1, F1), F is N*F1.
```
- Indexelés és listák
 - Gyakran kell az üres és nem-üres lista esetét szétválasztani.
 - A bemenő lista-argumentumot célszerű az első argumentum-pozícióba tenni.
 - Az [] és [...|...] eseteket az indexelés megkülönbözteti (funktork: '[]' / 0 ill. '[' / 2).
 - A két klóz sorrendje nem érdekes (feltéve, hogy zárt listával hívjuk az első pozíción) – de azért tegyük a leálló klózt mindig előre.

Listakezelő eljárások indexelése: példák

- Az append/3 választásmentesen fut le, ha első argumentuma zárt végű.


```
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```
- A last/2 közvetlen megfogalmazása nemdeterminisztikusan fut le:


```
% last(L, E): Az L lista utolsó eleme E.
last([E], E).
last(_|L, E) :- last(L, E).
```
- Érdemes segédeljárást bevezetni, last2/2 választásmentesen fut


```
last2([X|L], E) :- last2(L, X, E).

% last2(L, X, E): Az [X|L] lista utolsó eleme E.
last2([], E, E).
last2([X|L], _, E) :- last2(L, X, E).
```
- Az utolsó listaelemet választásmentesen felsoroló member/2:


```
member(E, [H|T]) :- member_(T, H, E).

% member_(L, X, E): Az [X|L] lista eleme E.
member_(_, E, E).
member_([H|T], _, E) :- member_(T, H, E).
```

Az indexelés és a vágó kölcsönhatása

- Hogyan vehető figyelembe a vágó az indexelés fordításakor?
- Példa: a $p(1, A)$ hívás választásmentes, de a $q(1, A)$ nem!


```
p(1, Y) :- !, Y = 2. % (1)
p(X, X). % (2)
Arg1=1 → (1), Arg1≠1 → (2) | q(1, 2) :- !. % (1)
q(X, X). % (2)
Arg1=1 → {(1),(2)}, Arg1≠1 → (2)
```
- A fordító figyelembe veszi a vágót az indexelésben, ha garantált, hogy egy adott fő funktor esetén a vágót elérjük. Ennek feltételei:
 - 1. arg. változó, konstans, vagy csak változókat tartalmazó struktúra,
 - a további argumentumok változók,
 - a fejben az összes változóelőfordulás különböző,
 - a törzs első hívása a vágó (előtte megengedve egy fejillesztést kiváltó egyenlőséget).
- Ekkor az adott funktorhoz tartozó listából kihagyja a vágó utáni klózoikat.
- Példa: $p(X, D, E) :- X = s(A, B, C), !, \dots$ $p(X, Y, Z) :- \dots$
- Ez egy újabb érv a vágás alapszabálya mellett:

A kimenő paraméterek értékadását mindig a vágó után végezzük!

A vágó és az indexelés hatékonysága – kieg. anyag

- Fibonacci-szerű sorozat: $f_1 = 1; f_2 = 2; f_n = f_{\lfloor 3n/4 \rfloor} + f_{\lfloor 2n/3 \rfloor}, n > 2$

<pre>% determ. xx=' ' fib(1, 1). fib(2, 2). fib(N, F) :-</pre>	<pre>% determ. lefut. xx='c' fibc(1, 1) :- !. fibc(2, 2) :- !. fibc(N, F) :-</pre>	<pre>% választásmentes, xx='ci' fibci(1, F) :- !, F = 1. fibci(2, F) :- !, F = 2. fibci(N, F) :-</pre>
--	--	--

$N > 2, N2 \text{ is } N*3//4, N3 \text{ is } N*2//3,$
 $\text{fibxx}(N2, F2), \text{fibxx}(N3, F3),$
 $F \text{ is } F2+F3.$

- Futási idők $N = 6000$ esetén

	fib	fibc	fibci
futási idő	1.25 sec	1.22 sec	1.13 sec
meghiúsulási idő	0.29 sec	0.03 sec	0.00 sec
összesen	1.54 sec	1.25 sec	1.13 sec
nyom-verem mérete	37.4Mbyte	18.7 Mbyte	240 byte

- fibc esetén a meghiúsulási idő azért nem 0, mert a rendszer a nyom-vermet (trail-stack) dolgozza fel. (A nyom-verem tárolja a változó-értékadások visszacsinálási információit.)

Tartalom

6 Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Jobbrekurzió (farok-rekurzió, tail-recursion) optimalizálás

- Az általános rekurzió költséges, helyben és időben is.
- Jobbrekurzióról beszélünk, ha
 - a rekurzív hívás a klóztörzs utolsó helyén van, vagy az utolsó helyen szereplő diszjunkció egyik ágának utolsó helyén stb., és
 - a rekurzív hívás pillanatában **nincs választási pont a predikátumban** (a rekurzív hívást megelőző célok determinisztikusan futottak le, nem maradt nyitott diszjunkciós ág).
- Jobbrekurzió optimalizálás: az utolsó hívás végrehajtása **előtt** az eljárás által lefoglalt hely felszabadul ill. személygyűjtésre alkalmassá válik.
- Ez az optimalizálás nemcsak rekurzív hívás esetén, hanem minden **utolsó** hívás esetén megvalósul – a pontos név: utolsó hívás optimalizálás (last call optimisation).
- A jobbrekurzió így tehát nem növeli a memória-igényt, korlátlan mélységig futhat – mint a ciklusok az imperatív nyelvekben. Példa:


```
ciklus(Állapot) :- lépés(Állapot, Állapot1), !, ciklus(Állapot1).
ciklus(_Állapot).
```

Predikátumok jobbrekurzív alakra hozása – listaösszeg

Az akkumulátorok használata

- A listaösszegzés „természetes”, nem jobbrekurzív definíciója:


```
% sum0(+L, ?S): L elemeinek összege S (S = 0+Ln+Ln-1+...+L1).
sum0([], 0).
sum0([X|L], S):-      sum0(L,S0), S is S0+X.
```
- Jobbrekurzív lista-összegző:


```
% sum(+L, ?S): L elemeinek összege S (S = 0+L1+L2+...+Ln).
sum(L, S):-          sum(L, 0, S).

% sum(+L, +S0, ?S): L elemeit S0-hoz adva kapjuk S-t. (≡ Σ L = S-S0)
sum([], S, S).
sum([X|L], S0, S):-  S1 is S0+X, sum(L, S1, S).
```
- A jobbrekurzív sum eljárás több mint **3-szor gyorsabb** mint a sum0!
- Az **akkumulátor** az imperatív (azaz megváltoztatható értékű) változó fogalmának deklaratív megfelelője:
 - A sum/3-ban az S0 és S argumentumok akkumulátorpárt alkotnak.
 - Az akkumulátorpár két része az adott változó mennyiség (a példában az összeg) különböző időpontokban vett értékeit mutatja:
 - S0 az összeg a sum/3 **meghívásakor**: a változó kezdőértéke;
 - S az összeg a sum/3 **lefutása után**: a változó végértéke.

- Az akkumulátorokkal általánosan több egymás utáni változtatást is leírhatunk:

```
p(..., A0, A):-
    q0(..., A0, A1), ...,
    q1(..., A1, A2), ...,
    qn(..., An, A).
```

- A sum/3 második klóza ilyen alakra hozva:


```
sum([X|L], S0, S):- plus(X, S0, S1), sum(L, S1, S).

plus(X, S0, S) :- S is S0+X.
```
- Akkumulátorváltozók elnevezési konvenciója: kezdőérték: *vált0*; közbülső értékek: *vált1*, ..., *vált_n*; végérték: *vált*.
- A Prolog akkumulátorpár nem más mint a funkcionális programozásból ismert gyűjtőargumentum és a függvény eredményének együttese.
- A DCG formalizmus – akkumulátorpárok automatikus „átszövése”:


```
sum([X|L]) --> plus(X), sum(L).
```


Akkumulátorok használata – folytatás

- Többszörös akkumulálás – lista összege és négyzetösszege

```
% sum2(+L, +S0, ?S, +Q0, ?Q): S-S0 = Σ Li, Q-Q0 = Σ Li2
sum2([], S, S, Q, Q).
sum2([X|L], S0, S, Q0, Q):-
    S1 is S0+X, Q1 is Q0+X*X, sum2(L, S1, S, Q1, Q).
```

- Többszörös akkumulátorok összevonása egyetlen **állapotváltozóvá**

```
% sum3(+L, +S0/Q0, ?S/Q): S-S0 = Σ Li, Q-Q0 = Σ Li2
sum3([], SQ, SQ).
sum3([X|L], SQ0, SQ) :-
    plus3(X, SQ0, SQ1), sum3(L, SQ1, SQ).
% teljesen analóg a "sima" összegzővel
```

```
plus3(X, S0/Q0, S/Q) :- S is S0+X, Q is Q0+X*X.
```

Különbséglisták

- 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 (ez feltételezi, hogy L0 szuffixuma L-nek).
- Például az [1,2,3] listának megfelelő különbséglisták:
 - [1,2,3,4]-[4], [1,2,3,a,b]-[a,b], [1,2,3]-[], ...
 - A legáltalánosabb (nyílt) különbséglistában a „kivonandó” változó:
`[1,2,3|L]-L`
- Egy nyílt különbséglista konstans időben összefűzhető egy másikkal:
`% app_d1(DL1, DL2, DL3): DL1 és DL2 különbséglisták összefűzése DL3.`
`app_d1(L-L0, L0-L1, L-L1).`
`| ?- app_d1([1,2,3|L0]-L0, [4,5|L1]-L1, DL).`
 `=> DL = [1,2,3,4,5|L1]-L1, L0 = [4,5|L1]`
- A nyílt különbséglista „egyszer használatos”, egy hozzáfűzés után már nem lesz nyílt!

Különbséglisták (folyt. – kiegészítő anyag)

- Példa: lineáris idejű listafordítás, nrev stílusában, különbséglistával:
`% nrev(L, DR): Az L lista megfordítása a DR különbséglista.`
`nrev_d1([], L-L). % L-L ≡ üres különbséglista`
`nrev_d1([X|L], DR) :-`
 `nrev_d1(L, DR0),`
 `app_d1(DR0, [X|T]-T, DR). % [X|T]-T ≡ egyelemű különbséglista`
`% app_d1(DL1, DL2, DL3): DL1 és DL2 különbséglisták összefűzése DL3.`
`app_d1(L-L0, L0-L1, L-L1).`
`% Az L lista megfordítása R`
`rev(L, R) :- nrev_d1(L, R-[]).`
- Az nrev_d1/2 eljárás törzsében érdemes a két hívást megcserélni (jobbrekurzió!).
- `nrev_d1(L, R-R0) => rev2(L, R0, R)` átalakítással és `app_d1` kiküszöbölésével a fenti `nrev_d1/2` eljárásból kapunk egy `rev2/3`-t, amely azonos `revapp/3`-mal!
- Ettől az átalakítástól kb **3-szor gyorsabb** lesz a program => érdemes a különbséglisták helyett akkumulátorpárokat használni!
- A továbbiakban a különbséglista jelölést csak a fejkommentek megfogalmazásában használjuk.

Az append mint akkumuláló eljárás (kiegészítő anyag)

- Írjunk egy `eleje_marad(Eleje, L, Marad)` eljárást!
`% eleje_marad(Eleje, L, Marad): Az L lista prefixuma az Eleje lista,`
`% ennek L-ből való elhagyása után marad a Marad lista.`
`eleje_marad([], L, L).`
`eleje_marad([X|Xs], L0, L) :-`
 `L0 = [X|L1], eleje_marad(Xs, L1, L).`
- Az akkumulálási lépés: `L0 = [X|L1]`, egy elem **elhagyása** a lista elejéről.
- A 2. és 3. argumentum felcserélésével az `eleje_marad` eljárás átalakul az `append` eljárássá!
- Tehát az `append` is tekinthető akkumuláló eljárásnak (a 2. és 3. argumentum a szokásos akkumulátorpárokhoz képest fel van cserélve):
`% append(Xs, L, L0): L0 elejéről Xs elemeit lehvagyva marad L.`
`% Másképpen: Xs = L0-L.`
`append([], L, L).`
`append([X|Xs], L, L0) :-`
 `L0 = [X|L1], append(Xs, L, L1).`
- Az akkumulálási lépés: az `L0` változó értékül kap egy listát, melynek farka `L1`, az akkumulált mennyiség: az a változó, amelyben az összefűzés eredményét várjuk.

Tartalom

Egy mintafeladat: $a^n b^n$ alakú sorozat előállítás (kieg. anyag)

6 Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

• 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).

% 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).
```

• Második megoldás, $2n$ lépés

```
anbn(N, L) :-
    an(N, b, [], BN),
    an(N, a, BN, 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 (kieg. anyag, folyt.)• Harmadik megoldás, n lépés

```
anbn(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.
anbn(0, L0, L) :- !, L = L0.
anbn(N, L0, [a|L]) :-
    N > 0,
    N1 is N-1,
    anbn(N1, [b|L0], L).
```

• A második klóz nem jobbrekurzív változata

```
anbn(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
```

 $a^n b^n$ alakú sorozatok – C++ megoldás (kiegészítő anyag)

• C++ megoldás

```
link *anbn(unsigned n) {
    link *l = 0, *b = 0; // ez elé építjük a b-ket
    link **a = &l;       // ebbe tesszük az a-kat
    for (; n > 0; --n) {
        *a = new link('a'); // előlről
        a = &(*a)->next;    // hátra épít
        b = new link('b', b); // hátulról előre épít
    }
    *a = b; return l;
}
```

Összetettebb adatstruktúrák akkumulálása (kiegészítő anyag)

- Az adatstruktúra:

```
% :- type bfa -> ures ; bfa(int, bfa, 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 akkumulátorpár, 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 – kieg. anyag, folyt.

- Lista konverziója bináris fává

```
% lista_bfa(L, BF0, BF): L elemeit 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).
BF = bfa(3,bfa(1,ures,ures),bfa(5,ures,ures)) ? ;
no

| ?- lista_bfa([3,1,5,1,2,4], ures, BF).
BF = bfa(3,bfa(1,ures,bfa(2,ures,ures)),
        bfa(5,bfa(4,ures,ures),ures)) ? ;
no
```

Akkumulálás bináris fákkal (kieg. anyag)

- Elem beszúrása bináris fába

```
% beszur(BF0, E, BF): E beszúrása BF0 rendezett fába
% 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 – kieg. anyag, 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
```

Tartalom

6 Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Hogyan írjunk át imperatív nyelvű algoritmust Prolog programmá?

- Példafeladat: Hatékony hatványozási algoritmus
 - Alaplépés: a kitevő felezése, az alap négyzetre emelése.
 - A kitevő kettes számrendszerbeli alakja szerint hatványoz.
- Az algoritmust megvalósító C nyelvű függvény:


```
/* hatv(a, h) = a**h */
int hatv(int a, unsigned h)
{
    int e = 1;
    while (h > 0)
    {
        if (h & 1) e *= a;
        h >>= 1; a *= a;
    }
    return e;
}
```
- Az algoritmusban három változó van: a, h, e:
 - a és h végértékére nincs szükség,
 - e végső értéke szükséges (ez a függvény eredménye).

A hatv C függvénynek megfelelő Prolog eljárás

- Kétargumentumú C függvény \Rightarrow 2+1-argumentumú Prolog eljárás.
- A függvény eredménye \Rightarrow utolsó arg.: $\text{hatv}(+A, +H, ?E): A^H = E$.
- Ciklus \Rightarrow segédeljárás: $\text{hatv}(+A0, +H0, +E0, ?E): A0^{H0} * E0 = E$.
- »a« és »h« C változók \Rightarrow »+A0« és »+H0« bemenő paraméterek (nem kell végérték),
- »e« C változó \Rightarrow »+E0, ?E« akkumulátorpár (kezdőérték, végérték).

```
hatv(A, H, E) :-
    hatv(A, H, 1, E).

hatv(A0, H0, E0, E) :- H0 > 0, !,
    (
        H0 /\ 1 == 1
        % /\ == bitenkénti "és"
    -> E1 is E0*A0
    ;   E1 = E0
    ),
    H1 is H0 >> 1,
    A1 is A0*A0,
    hatv(A1, H1, E1, E).
hatv(_, _, E, E).
```

```
int hatv(int a, unsigned h)
{
    int e = 1;

    ism: if (h > 0)
        { if (h & 1)
            e *= a;

            h >>= 1;
            a *= a;
            goto ism;
        } else return e;
}
```

A C ciklus és a Prolog eljárás kapcsolata

- A ciklust megvalósító Prolog eljárás minden pontján minden C változónak megfeleltethető egy Prolog változó (pl. h-nak H0, H1, ...):
 - A ciklusmag elején a C változók a megfelelő Prolog argumentumban levő változónak felelnek meg.
 - Egy C értékadásnak egy új Prolog változó bevezetése felel meg, az ez után következő kódban az új változó felel meg a C változónak.
 - Ha a diszjunkció egyik ága megváltoztat egy változót, akkor a többi ágon is be kell vezetni az új Prolog változót, a régivel azonos értékkel (ld. if (h & 1) ...).
- A C ciklusmag végén a Prolog eljárást vissza kell hívni, argumentumaiban az egyes C változóknak pillanatnyilag megfeleltetett Prolog változóval.
- A C ciklus **ciklus-invariánsa** nem más mint a Prolog eljárás fejkommentje, a példában:

$$\% \text{hatv}(+A0, +H0, +E0, ?E): A0^{H0} * E0 = E.$$

Programhelyesség-bizonyítás (kiegészítő anyag)

- Egy algoritmus (függvény) specifikációja:
 - előfeltételek:** a bemenő paramétereknek teljesíteniük kell ezeket,
 - utófeltételek:** a paraméterek és az eredmény kapcsolatát írják le.
- Egy algoritmus **helyes**, ha minden, az előfeltételeket kielégítő adatra a függvény hibátlanul lefut, és eredményére fennállnak az utófeltételek.
- Példa: `x = mfoku_gyok(a,b,c)`
 - előfeltételek: $b*b-4*a*c \geq 0$, $a \neq 0$
 - 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 (kieg. anyag)

- A ciklusokat „fel kell vágni” egy **ciklus-invariánssal**, amely:
 - az előfeltételekből és a ciklust megelőző értékadásokból következik,
 - ha a ciklus elején fennáll, akkor a ciklus végén is (indukció),
 - belőle és a leállási feltételből következik a ciklus utófeltétele.

```
int hatv(int a0, unsigned h0) /*utófeltétel: hatv(a0, h0) = a0h0 */
{ int e = 1, a = a0, h = h0;
  while /*ciklus-invariáns: a0h0 == e*ah */ (h > 0)
  {
    /* induláskor 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 */
```

Tartalom

6 Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás**
 - Magasabbrendű eljárások
 - Dinamikus adatbáziskezelés
 - Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
 - Egy összetettebb példaprogram
 - „Hagyományos” beépített eljárások
 - Fejlettebb nyelvi és rendszerelemek

Modulok definiálása SICStus Prolog nyelven

- 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/arg.szám)
- Példa:


```
:- 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 névkonstans, vagy pl. `library(KönyvtárNév)`:


```
:- use_module(plato).                % a fenti modul betöltése
:- use_module(library(lists), [last/2]). % csak last/2 importált
```
- Modulkvalifikált hívási forma: *Modul:Hívás* a *Modul*-ban futtatja *Hívás*-t.
- A modulfogalom nem szigorú, egy nem exportált eljárás is meghívható modulkvalifikált formában, pl. `platók:első_fennsík(...)`.

Meta-eljárások modularizált programban

- Meta-paraméterek átadása modulközi hívásokban:

modul1.pl állomány:

```
:- module(modul1, [kétszer/1]).
```

```
%:- meta_predicate kétszer(:).  (*)
kétszer(X) :-
    X, X.
```

```
p :- write(bu).
```

- Futtatás:

```
| ?- [modul1,modul2].
| ?- q. => bubu
| ?- r. => baba
```

- Automatikus modul-kvalifikáció meta-predikátum deklarációval:
Ha modul1.pl-ben elhagyjuk a (*)-gal jelzett sor előtti % kommentjelet, akkor

```
| ?- q. => baba!
```

modul2.pl állomány:

```
:- module(modul2, [q/0,r/0]).
```

```
:- use_module(modul1).
```

```
q :- kétszer(p).
```

```
r :- kétszer(modul2:p).
```

```
p :- write(ba).
```

Meta-predikátum deklaráció, modulnév-kiterjesztés

- Meta-predikátum deklaráció

- Formája:

```
:- meta_predicate <eljárásnév>(<módspec1>, ..., <módspecn>),
....
```

- <módspec_i> lehet ':', '+', '-', vagy '?'.
A ':' mód azt jelzi, hogy az adott argumentumot **betöltéskor** ún.

- A ':' mód azt jelzi, hogy az adott argumentumot **betöltéskor** ún. modulnév-kiterjesztésnek kell alávetni. (A többi mód hatása azonos, be/kimenő irányt jelezhetünk segítségükkel.)

- Egy *Kif* kifejezés modulnév-kiterjesztése a következő átalakítást jelenti:

- ha *Kif* *M:X* alakú, vagy olyan változó, amely az adott eljárás fejében meta-argumentum pozíción van, akkor változatlanul hagyjuk;

- egyébként helyettesítjük *CurMod:Kif*-fel, ahol *CurMod* a kurrens modul.

- Példa folyt. (tfh. a modul1-beli kétszer meta-predikátumnak deklarált!)

```
:- module(modul2, [négyyszer/1,q/0]).
```

```
:- use_module(modul1).
```

```
q :- kétszer(p).
```

% a tárolt alak:

⇒ q :- kétszer(modul2:p).

```
:- meta_predicate négyyszer(:).
```

```
négyyszer(X) :- kétszer(X), kétszer(X). ⇒ változatlan
```

Tartalom

6 Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások**
- Dinamikus adatbáziskezelés
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

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

- Páros elemek kiválasztása (vö. Erlang filter)

% Az L egész-lista páros elemeinek listája Pk.

```
páros_elemei(L, Pk) :-
```

```
    findall(X, (member(X, L), X mod 2 =:= 0), Pk).
```

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

- A listaelemek négyzetre emelése (vö. Erlang map)

% Az L számlista elemei négyzeteinek listája Nk.

```
négyzetei(L, Nk) :-
```

```
    findall(Y, (member(X, L), négyzete(X, Y)), Nk).
```

```
négyzete(X, Y) :- Y is X*X.
```

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


Részlegesen paraméterezett eljáráshívások – segédeszközök

- A `négyzete/0` kifejezés a `négyzete/2` **részlegesen paraméterezett** hívásának tekinthető.
- Ilyen hívások kiegészítésére és meghívására szolgálnak a `call/N` eljárások.
- `call(RPred, A1, A2, ...)` végrehajtása: az `RPred` **részleges** hívást kiegészíti az `A1, A2, ...` argumentumokkal, és meghívja.
- A `call/N` eljárások SICStus 4-ben már beépítettek, SICStus 3-ban még definiálni kellett ezeket, pl. így:

```
:- 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).
```

```
...
```

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

- *% foldl(+Xs, :Pred, +Y0, -Y): Y0-ból indulva, az Xs elemeire balról jobbra sorra alkalmazva a Pred által leírt kétargumentumú függvényt kapjuk Y-t.*
- ```
foldl([X|Xs], Pred, Y0, Y) :-
 call(Pred, X, Y0, Y1), foldl(Xs, Pred, Y1, Y).
foldl([], _, Y, Y).
```
- ```
jegyhozzá(Alap, Jegy, Szam0, Szam) :- Szam is Szam0*Alap+Jegy.
```
- ```
| ?- foldl([1,2,3], jegyhozzá(10), 0, E). => E = 123
```
- *% foldr(+Xs, :Pred, +Y0, -Y): Y0-ból indulva, az Xs elemeire jobbról balra sorra alkalmazva a Pred kétargumentumú függvényt kapjuk Y-t.*
- ```
foldr([X|Xs], Pred, Y0, Y) :-
    foldr(Xs, Pred, Y0, Y1), call(Pred, X, Y1, Y).
foldr([], _, Y, Y).
```
- ```
| ?- foldr([1,2,3], jegyhozzá(10), 0, E). => E = 321
```

Részlegesen paraméterezett eljárások – rekurzív `map/3`

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

```
% 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([], _, []).
```

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

- Példák:

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

- A `call/N`-re épülő megoldás előnyei:

- általánosabb és hatékonyabb lehet, mint a `findall`-ra épülő;
- alkalmazható akkor is, ha az elemekre elvégzendő műveletek nem függetlenek, pl. `foldl`.

## Tartalom

## 6 Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

## 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`
- A klózfelvétel ill. törlés **tartós** mellékhatás, visszalépéskor **nem** áll vissza a korábbi állapot.

## Klóz felvétele: `asserta/1`, `assertz/1`

- `asserta(:@Klóz)`
  - A Klóz kifejezést klózként értelmezve felveszi a programba az adott predikátum *első* klózaként. A Klózban levő változók szisztematikusan újakra cserélődnek.
  - 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)`
  - Ugyanaz mint `asserta`, csak a Klóz kifejezést az adott predikátum *utolsó* klózaként veszi fel.
- Példa:
 

```
| ?- assertz((p(1,X):-q(X))), asserta(p(2,0)),
 assertz((p(2,Z):-r(Z))), listing(p). => p(2, 0).
 p(1, A) :- q(A).
 p(2, A) :- r(A).

| ?- assert(s(X,X)), s(U,V), U == V, X \== U.
=> V = U ? ; no
```

## Klóz törlése: `retract/1`

- `retract(:@Klóz)`
  - A Klóz 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), Cl = (p(2,_):-_),
 retract(Cl), format('Del: ~w.\n', [Cl]), listing(p), fail.
```

- A futás kimenete:

|                                                               |                                                                         |                                                          |
|---------------------------------------------------------------|-------------------------------------------------------------------------|----------------------------------------------------------|
| <pre>p(2, 0). p(1, A) :-     q(A). p(2, A) :-     r(A).</pre> | <pre>Del: p(2,0):-true. p(1, A) :-     q(A). p(2, A) :-     r(A).</pre> | <pre>Del: p(2,_537):-r(_537). p(1, A) :-     q(A).</pre> |
| <pre>=&gt; no</pre>                                           |                                                                         |                                                          |

## Alkalmazási példa – egyszerűsített `findall`

- A `findall1/3` eljárás hatása megegyezik a beépített `findall`-al, de
    - nem jó, ha a Cél-ban újabb, skatulyázott `findall1` hívás van.
- ```
:- dynamic(megoldás/1).

% findall1(Minta, Cél, L): Cél összes megoldására Minták listája L.
findall1(Minta, Cél, _MegoldL) :-
    call(Cél),
    asserta(megoldás(Minta)), % fordított sorrendben vesszük fel!
    fail.
findall1(_Minta, _Cél, MegoldL) :-
    megoldás_lista([], MegoldL).

% 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). => 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 (tényállítási esetén `Törzs = true`).
  - Ha az illesztés sikerült, akkor sikeresen lefut.
  - Visszalépés esetén folytatja a keresést (illeszt, sikerül stb.)

•

## • Példa:

```
:- listing(p), clause(p(2, 0), T).
```

```
p(2, 0).
```

```
p(1, A) :-
 q(A).
```

```
p(2, A) :-
 r(A).
```

```
T = true ? ;
T = r(0) ? ;
no
```

## A clause eljárás alkalmazása: egyszerű nyomkövető interpreter

- Az alábbi interpreter csak „tisztá”, beépített eljárást nem alkalmazó Prolog programok futtatására alkalmas.

*% 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), 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:*/ format('~|~t~*+', [D]),
 write(Port), write(': '), write(G), nl.
```

## Nyomkövető interpreter - példafutás

```
:- dynamic app/3, app/4. % (*)

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

app(L1, L2, L3, L123) :-
 app(L1, L23, L123),
 app(L2, L3, L23).

A (*) sor elhagyható, ha a fenti (mondjuk app34) állományt az alábbi (SICStus-specifikus) beépített eljárással töltjük be:

| ?- load_files(app34,
 compilation_mode(
 assert_all)).
| ?- interp(app(_, [b,c], L, [c,b,c,b]), 0).
call: app(_203, [b,c], _253, [c,b,c,b])
call: app(_203, _666, [c,b,c,b])
exit: app([], [c,b,c,b], [c,b,c,b])
call: app([b,c], _253, [c,b,c,b])
fail: app([b,c], _253, [c,b,c,b])
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], [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] ?
```

## Tartalom

## 6 Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

## A DCG (Definite Clause Grammars) formalizmus

- DCG: előfeldolgozó eszköz nyelvtani elemzők írásához.
- DCG szabály:  $Fej \rightarrow Törzs \implies Fej(A_0, A_m) :- Törzs(A_0, A_m)$ . A törzsben:
  - $\{Cél\} \implies Cél$  (akkumulálást nem végző cél)
  - $[E_1, E_2, \dots, E_k], k \geq 0 \implies A_n = [E_1, E_2, \dots, E_k | A_{n+1}]$  (elemek akk.-a)
  - $p(X_1, X_2, \dots, X_j), j \geq 0 \implies p(X_1, X_2, \dots, X_j, A_n, A_{n+1})$  (akk.-t végző cél)
  - Vezérlés: konj. (.), diszj. (;), ha-akkor ( $\rightarrow$ ), vágó (!), negáció ( $\backslash$ )
- Példa: egy lista pozitív elemeinek kigyűjtése
 

```
% pe(L, Pk0, Pk): Az L számlista pozitív elemeinek listája Pk0-Pk.
% Másszóval: L pozitív elemeinek listáját Pk elé fűzve kapjuk Pk0-t
pe([], Pk0, Pk) :- Pk0 = Pk.
pe([X|L], Pk0, Pk) :- (X > 0 -> Pk0 = [X|Pk1], pe(L, Pk1, Pk)
 ; pe(L, Pk0, Pk)
).
```
- A DCG jelölést használó, a fentivel azonos kódot eredményező eljárás:
 

```
pe2([]) --> [].
pe2([X|L]) --> ({X > 0} -> [X], pe2(L)
 ; pe2(L)
).
```

## DCG nyelvtani elemzés – további részletek

- Az elemzés – a Prolog végrehajtás miatt – nem-determinisztikus, pl.
 

```
| ?- szám("123 abc", L).
L = " abc" ? ; % leelemeztük a 123 számot
L = "3 abc" ? ; % leelemeztük a 12 számot
L = "23 abc" ? ; % leelemeztük az 1 számot
no
```
- A számmaradék eljárás determinisztikus változata
 

```
% számmaradék2(L0, L): L0-L számjegykódok maximális listája
számmaradék2 --> (számjegy -> számmaradék2
 ; ""
).
```

vagy

```
számmaradék3 --> számjegy, !, számmaradék3. % A vágó köré nem kell {}
számmaradék3 --> "".
```
- Futás:
 

```
| ?- szám2("123 abc", L).
L = " abc" ? ; % leelemeztük a (lehető leghosszabb) 123 számot
no
```

## A DCG formalizmus használata nyelvtani elemzésre

- Példa – decimális számok elemzését végző szám(L0, L) Prolog eljárás
- Az L0, L paraméterek: karakterkódok listái
 

```
% szám(L0, L): Az L0-L különbséglista számjegykódok nem-üres listája
% Másszóval: L0 elejéről leelemezhető egy szám, és marad L
szám --> számjegy, számmaradék.

% számmaradék(L0, L): Az L0-L különbséglista számjegykódok listája
számmaradék --> számjegy, számmaradék ; "".
```

$\% "" \equiv []$

```
% számjegy(L0, L): L0 - [K|L], ahol K egy számjegy kódja
számjegy --> "0"; "1"; "2"; "3"; "4"; "5"; "6"; "7"; "8"; "9". % "9" \equiv [0'9]
```
- A számjegy/2 eljárás egy másik megvalósítása
 

```
számjegy --> [K], {decimális_jegy_kódja(K)}.
```

% K egy számjegy kódja.

```
decimális_jegy_kódja(K) :- K >= 0'0, K <= 0'9.
```
- A fenti DCG szabály Prolog megfelelője:
 

```
számjegy(L0, L) :-
 L0 = [K|L], % K a következő listaelem
 decimális_jegy_kódja(K). % megfelelő-e a K?
```

## Az elemző kiegészítése jelentéshordozó argumentumokkal

- Egy DCG szabály az elemzéssel párhuzamosan további (kimenő) argumentum(ok)ban felépítheti a kielemezett dolog „jelentését”
- Példa: szám elemzése és értékének kiszámítása:
 

```
% Leelemezhető egy Sz értékű nem-üres számjegysorozat
szám(Sz) --> számjegy(J), számmaradék(J, Sz).

% Leelemezhető számjegyek egy esetleg üres listája, amelynek
% az eddig leelemzett Sz0-val együtt vett értéke Sz.
számmaradék(Sz0, Sz) -->
 számjegy(J), !, {Sz1 is Sz0*10+J}, számmaradék(Sz1, Sz).
számmaradék(Sz0, Sz0) --> [].
```

% leelemezhető egy J értékű számjegy.

```
számjegy(J) --> [K], {decimális_jegy_kódja(K), J is K-0'0}.

| ?- szám(Sz, "102 56", L). \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.
```
- Itt két akkumulátorpár van: egy „kézi” (Sz) és egy DCG-ből generált (L).

## Aritmetikai kifejezések elemzése

- Egyszerű aritmetikai kifejezések elemzése és kiértékelése.

```
% kif0(Z, L0, L): L0-L egy Z aritmetikai kifejezéssé elemezhető ki
kif0(X+Y) --> tag0(X), "+", !, kif0(Y).
kif0(X-Y) --> tag0(X), "-", !, kif0(Y).
kif0(X) --> tag0(X).

tag0(X) --> szám(X). % egyelőre
| ?- kif0(Z, "4-2+1", []). => Z = 4-(2+1) Jobbról balra elemez!
```

- Egy lehetséges javítás

```
kif(Z) --> tag(X), kifmaradék(X, Z).
kifmaradék(X, Z) --> "+", tag(Y), !, kifmaradék(X+Y, Z).
kifmaradék(X, Z) --> "-", tag(Y), !, kifmaradék(X-Y, Z).
kifmaradék(X, X) --> [].

tag(Z) --> szám(X), tagmaradék(X, Z).
tagmaradék(X, Z) --> "*", szám(Y), !, tagmaradék(X*Y, Z).
tagmaradék(X, Z) --> "/", szám(Y), !, tagmaradék(X/Y, Z).
tagmaradék(X, X) --> [].
```

```
| ?- kif(Z, "5*4-2+1", []), Val is Z. => Z = 5*4-2+1, Val = 19 ? ; no
```

## Do-ciklusok (do-loops)

- Szintaxis:

```
(Iterátor1, ..., Iterátorm
do Célsorozat
)
```

- Az L lista minden elemét megnövelve 1-gyel kapjuk az NL listát:

```
novel(L, NL) :-
 (foreach(X, L), foreach(Y, NL)
do Y is X+1
).
```

- Az L lista minden elemét megszorozva N-nel kapjuk az NL listát:

```
szoroz(L, N, NL) :-
 (foreach(X, L), foreach(Y, ML), param(N)
do Y is N*X
).
```

## Do-ciklusok: példák további iterátorokra

```
| ?- (for(I, 1, 5), foreach(I, List)
do true % I = 1, 2, ..., 5
).

List = [1,2,3,4,5] ? ; no

| ?- (foreach(X, [1,2,3]), fromto(0, In, Out, Sum)
do Out is In+X
%In1=0, Out1=In1+X1, In2=Out1, ..., Out3=In3+X3, Sum=Out3
).

Sum = 6 ? ; no

| ?- (foreach(X, [a,b,c,d,e]), count(I, 1, N), foreach(I-X, Pairs)
do true % I = 1, ..., N
).

N = 5, Pairs = [1-a,2-b,3-c,4-d,5-e] ? ; no

| ?- (foreacharg(A, f(a,b,c,d,e), I), foreach(I-A, List)
do true
).

List = [1-a,2-b,3-c,4-d,5-e] ? ; no
```

## Tartalom

### 6 Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek



## Egy nagyobb DCG példa: „természetes” nyelvű beszélgetés

```
% mondat(Alany, Áll, LO, L): LO-L kielemezhető egy Alany alanyból és Áll
% állítmányból álló mondat. Alany lehet első vagy második személyű
% névmás, vagy egyetlen szóból álló (harmadik személyű) alany.
mondat(Alany, Áll) -->
 {én_te(Alany, Ige)}, én_te_perm(Alany, Ige, Áll).
mondat(Alany, Áll) -->
 szó(Alany), szavak(Áll).

% én_te(Alany, Ige):
% Az Alany első/második személyű névmásnak megfelelő létige az Ige.
én_te("én", "vagyok").
én_te("te", "vagy").

% én_te_perm(Ki, Ige, Áll, LO, L): LO-L kielemezhető egy Ki
% névből, Ige igealakból és Áll állítmányból álló mondat.
én_te_perm(Alany, Ige, Áll) -->
 (szó(Alany), szó(Ige), szavak(Áll)
 ; szó(Alany), szavak(Áll), szó(Ige)
 ; szavak(Áll), szó(Ige), szó(Alany)
 ; szavak(Áll), szó(Ige)
).
```

## Példa: „természetes” nyelvű beszélgetés – párbeszéd-szervezés

```
% :- type mondás --> kérdez(szó) ; kijelent(szó,list(szó)) ; un.

% Megvalósít egy párbeszédet.
párbeszéd :-
 repeat,
 read_line(L), % beolvas egy sort, L a karakterkódok listája
 (menet(Mondás, L, []) -> feldolgoz(Mondás)
 ; write('Nem értem\n'), fail
),
 Mondás = un, !.

% menet(Mondás, LO, L): Az LO-L kielemezett 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 – szavak elemzése

```
% szó(Sz, LO, L): LO-L egy Sz betűsorozatból álló (nem üres) szó.
szó(Sz) --> betű(B), szómaradék(SzM), {illik([B|SzM], Sz)}, köz.

% szómaradék(Sz, LO, L): LO-L egy Sz kódlistából álló (esetleg üres) szó.
szómaradék([B|Sz]) --> betű(B), !, szómaradék(Sz).
szómaradék([]) --> [].

% illik(Szó0, Szó): Szó0 = Szó, vagy a kezdő kis-nagy betűben különböznek.
illik([B0|L], [B|L]) :-
 (B = B0 -> true
 ; abs(B-B0) == 32
).

% köz(LO, L): LO-L nulla, egy vagy több szóköz.
köz --> (" " -> köz ; "").

% betű(K, LO, L): LO-L egy K kódú "betű" (különbözik a " ." jelektől)
betű(K) --> [K], {\+ member(K, " .?")}.

% szavak(SzL, LO, L): LO-L egy SzL szó-lista.
szavak([Sz|Szk]) --> szó(Sz), (szavak(Szk)
 ; {Szk = []}
).
```

## Példa: „természetes” nyelvű beszélgetés – válaszok előállítás

```
:- dynamic tudom/2.

% feldolgoz(Mondás): feldolgozza a felhasználótól érkező Mondás üzenetet.
feldolgoz(un) :-
 write('ÉN is.\n').
feldolgoz(kijelent(Alany, Áll)) :-
 assertz(tudom(Alany,Áll)),
 write('Felfogtam.\n').
feldolgoz(kérdez(Alany)) :-
 tudom(Alany, _), !,
 válasz(Alany).
feldolgoz(kérdez(_)) :-
 write('Nem tudom.\n').

% Felsorolja az Alany ismert tulajdonságait.
válasz(Alany) :-
 tudom(Alany, Áll),
 (member(Szó, Áll), format('~s ', [Szó]), fail
 ; nl
), fail.
válasz(_).
```



## Beszélgetős DCG példa – egy párbeszéd

```
| ?- párbeszéd.
|: Magyar legény vagyok én.
Felfogtam.
|: Ki vagyok én?
Magyar legény
|: Péter kicsoda?
Nem tudom.
|: Péter tanuló.
Felfogtam.
|: Péter jó tanuló.
Felfogtam.
|: Péter kicsoda?
tanuló
jó tanuló
|: Boldog vagyok.
Felfogtam.

|: Én vagyok Jeromos.
Felfogtam.
|: Te egy Prolog program vagy.
Felfogtam.
|: Ki vagyok én?
Magyar legény
Boldog
Jeromos
|: Okos vagy.
Felfogtam.
|: Ki vagy te?
egy Prolog program
Okos
|: Valóban?
Nem értem
|: Unlak.
Én is.
```

## Tartalom

### 6 Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

## 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       |
| -                  | kivonás   | **  | hatványozás        |
| *                  | szorzás   | mod | modulus képzés     |
| /                  | osztás    | rem | maradék képzés     |
| 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  $0, 1, \dots$  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.
- `writeq(@X)`: Mint `write(X)`, csak gondoskodik, hogy szükség esetén az névkonstansok idézőjelek közé legyenek téve.
- `write_canonical(@X)`: Mint `writeq(X)`, csak operátorok nélkül, minden struktúra szabványos alakban jelenik meg.
- `write_term(@X, +Opciók)`: Az `Opciók` opciólista szerint kiírja `X`-et.
- `format(@Formátum, @AdatLista)`: A `Formátum`-nak megfelelő módon kiírja `AdatLista`-t. A formázójelek alakja: `~<szám esetleg><formázójel>`.

```
| ?- write('Helló világ'). => Helló világ
| ?- writeq('Helló világ'). => 'Helló világ'
| ?- write_canonical('*' - '%'). => -(*, '%')
| ?- write_canonical([1,2]). => '.(1, '.(2, []))
| ?- write_term([1,2,3], [max_depth(2)]). => [1,2|...]
| ?- format('X=~s -- ~3d s', [[0'j,0'6],3245]). => X=jó -- 3.245 s
```

## Karakterek kiírása és beolvasása

- `put_code(+Kód)`: Kiírja az adott kódú karaktert.
- `nl`: 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:

```
% rd_line(L): L a következő sor karakterkódjainak listája.
% read_line néven beépített eljárás SICStus 3.9.0-tól.
rd_line(L) :-
 peek_code(0'\n), !, get_code(_), L = [].
rd_line([C|L]) :-
 get_code(C), rd_line(L).

| ?- rd_line(L), member(X, L), put_code(X), write(' '), fail ; nl.
|: Hello world!
H e l l o w o r l d !
```

## Kifejezések kiírása – felhasználó vezérelte formázás

- `print(@X)`: Alapértelmezésben azonos `write`-tal. Ha a felhasználó definiál egy `portray/1` eljárást, akkor a rendszer minden a `print`-tel kinyomtatandó részkifejezésre meghívja `portray`-t. Ennek sikere esetén feltételezi, hogy a kiírás megtörtént, meghiúsulás esetén maga írja ki a részkifejezést. 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 = [[1,2], [3,4], [5,6]].
X =
[1,2]
[3,4]
[5,6] ?
```

## 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:

```
consult_body :-
 repeat,
 read(Term),
 (Term = end_of_file -> true
 ; assertz(Term), fail
),
 !.

| ?- consult_body.
|: p(X) :- q(X), r(X).
|: ~D
yes
```

```
| ?- listing([p/1]).
p(A) :-
 q(A),
 r(A).
yes
```

## Be- és kiviteli csatornák

- Csatornák megnyitása és kezelése:

- `open(@Filenév, @Mód, -Csatorna)`: Megnyitja a Filenév 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 ismertetett összes be- és kiviteli eljárásnak van egy eggyel több argumentumú változata, amelynek első argumentuma a csatorna. Ezek: `write/2`, `writeq/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 állománynevét egyesíti Filenév-val.
- `seen, told`: Lezárja a jelenlegi beviteli/kiviteli csatornát.
- Példák – nagyon egyszerű `consult` variánsok:

```
consult_dec10_style(File) :-
 seeing(Old), see(File),
 repeat,
 read(Term),
 (Term = end_of_file
 -> seen
 ; assertz(Term), fail
),
 !,
 see(Old).
```

```
consult_with_streams(File) :-
 open(File, read, S),
 repeat,
 read(S, Term),
 (Term = end_of_file
 -> close(S)
 ; assertz(Term), fail
),
 !.
```

## 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”, azaz a HibaKif hibahelyzet kiváltása:  
`throw(@HibaKif),`  
`raise_exception(@HibaKif)`
- 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-lal.
    - 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 körülvéző `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:
  - `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])`.

## Tartalom

### 6 Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

## Programfejlesztési eljárások (folytatás)

- `statistics`: Különböző statisztikákat ír ki az aktuális kimenetre.
- `statistics(?Fajta, ?Érték)`: Érték a Fajta fajtájú mennyiség értéke.
  - Példa: `statistics(runtime, E) => 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.
- `nospay(:@EljárásSpec)`: Megszünteti a megadott spion-pontokat.
- `nospayall`: Az összes spion-pontot megszünteti.

## Külső nyelvi interfész (kiegészítő anyag)

- Hagományos (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 pointereket 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 (kiegészítő anyag)

- A példa a `library(bdb)` megvalósításából származik.
- A C nyelven megírandó eljárás Prolog hívási alakja:  
`index_keys(+Spec, +Kif, -Kulcs, -Szám)`
- A megírandó eljárás 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 + névkonstansok száma, *Kulcs* pedig *Kif* megfelelő argumentumainak 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 interfész – a C kód (`ixkeys.c` állomány, kieg. anyag)

```
#include <sicstus/sicstus.h>

#define NA -1 /* not applicable */
#define NI -2 /* instantiatedness */

long 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;
}
```

## Külső nyelvi interfész – példa (SICStus 3) – kiegészítő anyag

- A példaeljárás használata
 

```
| ?- [ixtest].
| ?- index_keys(f(+, -, +, +),
 f(12.3, _, s(1, _, z(2)), t),
 Kulcs, Szam).
Kulcs = [12.3,s,3,t], Szam = 3 ?
```
- Az `ixtest.pl` Prolog file tartalmazza az interfész specifikációját:
 

```
foreign(ixkeys, index_keys(+term, +term, -term, [-integer])).
 % 1. arg: bemenő, általános kifejezés
 % 2. arg: bemenő, általános kifejezés
 % 3. arg: kimenő, általános kifejezés
 % 4. arg: a C függvény értéke, egész (long)

foreign_resource(ixkeys, [ixkeys]).

:- load_foreign_resource(ixkeys).
```
- A C programot elő kell készíteni a Prolog számára az `splfr` (link [foreign resource](#)) eszköz segítségével:
 

```
splfr ixkeys ixtest.pl +c ixkeys.c
```

## Hasznos lehetőségek SICStus Prolog-ban (kieg. anyag)

- 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 névkonstans 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-ban kieg. anyag, folyt.)

- Visszaléptethető módon változtatható kifejezések

```
create_mutable(Adat, ValtKif)
```

Adat kezdőértékkel létrehoz egy új változtatható kifejezést, ez lesz ValtKif. Adat nem lehet üres változó.

```
get_mutable(Adat, ValtKif)
```

Adat-ba előveszi ValtKif pillanatnyi értékét.

```
update_mutable(Adat, ValtKif)
```

A ValtKif 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, meghiúsult vagy kivételt dobott.

## Fejlett vezérlési lehetőségek SICStusban (kieg. anyag)

### Blokk-deklarációk

- Példa:

```
:- block p(-, ?, -, ?, ?).
```

Jelentése: ha az első és a harmadik argumentum is behelyettesítetlen 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 (kieg. anyag, folyt.)

- Generál-és-ellenőriz típusú programok gyorsítása

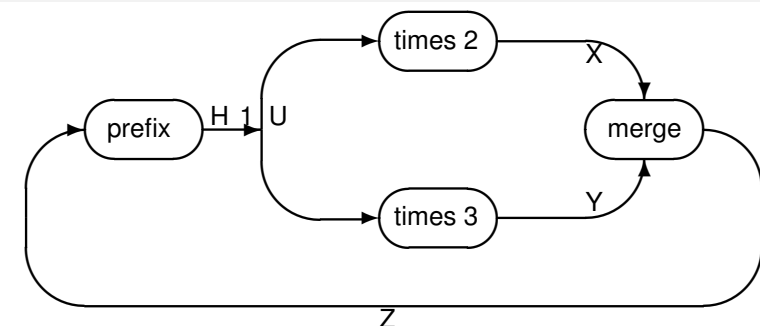
- általában nem hatékonyak (pl megrajzolja\_1), mert túl sok visszalépést használnak
- korutinszervezéssel a generáló és ellenőrző rész „automatikusan” összefésülhető
- ehhez az ellenőrző részt kell előre tenni és megfelelően blokkolni

- 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 (kieg. anyag)



% A H lista az első N, csak a 2 és 3 tényezőkből álló szám.

```
hamming(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 (kieg. anyag, folyt.)

```
% merge(X, Y, Z): Z az X és Y összefésülése.
:- block merge(-, ?, ?), merge(?, -, ?).
% Csak akkor fusson, ha az első két argumentum ismert
merge([A|X], [B|Y], V) :-
 A < B, !, V = [A|Z], merge(X, [B|Y], Z).
merge([A|X], [B|Y], V) :-
 B < A, !, V = [B|Z], merge([A|X], Y, Z).
merge([A|X], [A|Y], [A|Z]) :-
 merge(X, Y, Z).
merge([], X, X) :- !.
merge(_, [], []).

% prefix(N, X, Y): Az X lista első N eleme Y.
prefix(0, _, []) :- !.
prefix(N, [A|X], [A|Y]) :-
 N > 0, N1 is N-1, prefix(N1, X, Y).
```

## Korutinszervező eljárások (kieg. anyag)

- `freeze(X, Hivas)`  
Hivast felfüggeszti mindaddig, amíg X behelyettesíthetően 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_vars(Hivas, Valtozok)`  
Hivas-t végrehajtja, és a Valtozok listában visszaadja mindazokat az új (a Hivas alatt létrejött) változókat, amelyekre vonatkoznak felfüggesztett hívások. Pl.  

```
| ?- call_residue_vars((dif(X,f(Y)), X=f(Z)), Vars).
 X = f(Z),
 Vars = [Z,Y],
 prolog:dif(f(Z),f(Y)) ?
```

## SICStus könyvtárak (kieg. anyag)

- Könyvtár betöltése `:- use_module(library(könyvtárnév)).`
- A legfontosabb könyvtárak
  - `avl` – AVL fák segítségével megvalósított „asszociációs listák” (kiterjeszthető leképezések Prolog kifejezések között)
  - `atts` – tetszőleges attributumok hozzárendelése Prolog változókhoz, tárolórekeszként és az egyesítés módosítására is használható
  - `bdb` – Prolog kifejezések állományokban való tárolására szolgáló adatbázis, felhasználó által definiált többszörös indexelési lehetőséggel
  - `between` – számintervallumok felsorolása
  - `codesio` – karakterlistából olvasó ill. abba író be- és kiviteli eljárások
  - `file_systems` – állományok és könyvtárak kezelése
  - `heaps` – bináris kazal (heap), pl. prioritásos sorokhoz (priority queue)
  - `lists` – listakezelő alapműveletek
  - `logarr` – logaritmikus elérési idejű kiterjeszthető tömbök
  - `odbc` – ODBC adatbázis interfész
  - `ordsets` – halmazműveletek (halmaz  $\equiv$  @< szerint rendezett lista)
  - `queues` – sorokra (queue, FIFO store) vonatkozó műveletek

## Legfontosabb SICStus könyvtárak, kieg. anyag, folyt.

- `random` – véletlenszám-generátor
- `samsort` – általános rendezés (nem szűri az ismétlődő elemeket)
- `sockets` – socket interfész
- `system` – operációsrendszer-szolgáltatások elérése
- `terms` – általános Prolog kifejezések kezelése (pl. változók kigyűjtése)
- `timeout` – célok futási idejének korlátozása
- `trees` – az arrays könyvtárhoz hasonló, de nem-kiterjeszthető logaritmikus elérési idejű tömbfogalom, bináris fakkal
- `ugraphs` – élcimké nélküli irányított és irányítatlan gráfok
- `wgraphs` – irányított és irányítatlan gráfok, egészértékű élcimkével
- `linda/client` és `linda/server` – Linda-szerű processzkommunikáció
- `clpb` – Boole-értékekre vonatkozó korlátmegoldó (constraint solver)
- `clpq` és `clpr` – korlátmegoldó a racionálisak és a valósok tartományán
- `clpfd` – véges tartományokra vonatkozó korlátmegoldó
- `tcltk` – A *Tcl/Tk* nyelv és eszközkészlet elérése
- `gauge` – Prolog programok a teljesítményvizsgálata `tcltk` grafikus felülettel

- Bevezetés a Logikai Programozásba c. jegyzet 6. fejezete:
  - Párhuzamos megvalósítások
  - Az Andorra-I rendszer rövid bemutatása
  - A Mercury nagyhatékonyságú LP megvalósítás
  - Korlát-logikai programozás (Constraint Logic Programming – CLP)
- Az utolsó két témával foglalkozik a „**Nagyhatékonyságú deklaratív programozás**” c. MSc szakirányos tárgy

- 1 Bevezetés
- 2 Cékla: deklaratív programozás C++-ban
- 3 Erlang alapok
- 4 Prolog alapok
- 5 Keresési feladat pontos megoldása
- 6 Haladó Prolog
- 7 Haladó Erlang

## Prolog és Erlang: néhány eltérés és hasonlóság

| Prolog                                          | Erlang                                                          |
|-------------------------------------------------|-----------------------------------------------------------------|
| predikátum, kétféle érték                       | függvény, értéke tetsz. típusú                                  |
| siker esetén változóbehelyettesítés             | csak bemenő argumentum és visszatérési érték van                |
| választási pontok, többféle megoldás            | determinizmus, egyetlen mo.                                     |
| $C_1, \dots, C_n$ célsor., redukció+visszalépés | $S_1, \dots, S_n$ szekv. kif., értéke $S_n$                     |
| összetett kifejezés (struktúra), a lista is     | ennes, lista típusok (tuple, list)                              |
| operátor definiálása                            | -                                                               |
| egyesítés szimmetrikus                          | jobb oldalon tömör kif., bal oldalon mintakif.; őrfeltételekkel |

- Néhány hasonlóság:
  - a függvény is klózokból áll, kiválasztás mintaillesztéssel, sorrendben
  - a függvényt is a funktora (pl. `bevezeto:fac/1`) azonosítja
  - változóhoz csak egyszer köthető érték
  - lista szintaxisa (de: Erlangban önálló típus), sztring (füzér), atom

## Tartalom

- 7 Haladó Erlang
  - Kivételkezelés
  - Rekord
  - Rekurzív adatstruktúrák
  - Rekurzió fajtái
  - Halmazműveletek (rendezetlen listával)
  - Generikus keresőfák
  - Lusta farkú lista Erlangban

## Kivételkezelés

- Kivétel jelzése háromféle *kivételtípussal* lehetséges
  - `throw(Why)`  
Olyan hiba jelzésére, amelynek kezelése várható az alkalmazástól, vagy egyszerűen mély hívásból visszatérni
  - `exit(Why)`  
A futó processz befejezésére
  - `error(Why)`  
Rendszerhiba jelzésére, amelynek kezelése komplex
- Kivétel elkapása kétféleképpen lehetséges
  - `try ... catch` kifejezéssel
  - `catch` kifejezéssel:
    - visszaadja a keletkező kivétel termjét, vagy ha nincs hiba, a kifejezés kiértékelését
    - debughoz hasznos, könnyen felderíthető a kivétel pontos értéke

## Példák `try ... catch` és `catch` használatára

### `kiv.erl` – Példák kivételkezelésre

```
% Ha Fun(Arg) hibát ad, 'error', különben {ok, Fun(Arg)}.
safe_apply(Fun, Arg) -> try Fun(Arg) of
 V -> {ok,V}
catch throw:_Why -> error;
 error:_Why -> error
end. % például error:function_clause
```

```
2> lists:last([a,b,c]).
c
3> lists:last([]).
** exception error: no function clause matching lists:last([])
4> catch lists:last([]).
{'EXIT',{function_clause,[...% stack trace]}}
5> kiv:safe_apply(fun lists:last/1, [a,b,c]).
{ok,c}
6> kiv:safe_apply(fun lists:last/1, []).
error
```

## Kivételkezelés: `try ... catch`

```
try Kifejezés [of
 Minta1 [when ŐrSz1] -> Kif1;
 ...
 Mintan [when ŐrSzn] -> Kifn]
catch
 ExTípus1: ExMinta1 [when ExŐrSz1] -> ExKif1;
 ...
 ExTípusn: ExMintan [when ExŐrSzn] -> ExKifn
[after
 AfterKif]
end
```

- Ha a Kifejezés kiértékelése sikeres, az értékét az Erlang megpróbálja az `of` és `catch` közötti mintákra illeszteni
- Ha a kiértékelés sikertelen, az Erlang a jelzett kivételt próbálja meg illeszteni a `catch` és `after` közötti mintákra
- Minden esetben kiértékeli az `after` és `end` közötti kifejezést
- A `try` szerkezet speciális esete a `case`, amelyben nincs kivételkezelés

## Példa `try ... catch` és `catch` használatára

### `kiv.erl` – folytatás

```
genExc(A,1) -> A;
genExc(A,2) -> throw(A);
genExc(A,3) -> exit(A);
genExc(A,4) -> error(A).
```

```
tryGenExc(X,I) -> try genExc(X,I) of
 Val -> {I, 'Lefutott', Val}
catch
 throw:X -> {I, 'Kivetelt dobott', X};
 exit:X -> {I, 'Befejezodott', X};
 error:X -> {I, 'Sulyos hibat jelzett', X}
end.
```

```
7> [kiv:tryGenExc(X,I) || {X,I} <- [{ 'No',1},{ 'Th',2},{ 'Ex',3},{ 'Er',4}]].
[{I, 'Lefutott', 'No'}, {2, 'Kivetelt dobott', 'Th'}, {3, 'Befejezodott', 'Ex'},
{4, 'Sulyos hibat jelzett', 'Er'}]
8> [catch kiv:genExc(X,I) || {X,I} <- [{ 'No',1},{ 'Th',2},{ 'Ex',3},{ 'Er',4}]].
['No', 'Th', {'EXIT', 'Ex'}, {'EXIT', 'Er', [% stack trace]}]
```

## Tartalom



### Haladó Erlang

- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Rekurzió fajtái
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Lusta farkú lista Erlangban

## Modul

- Modul: attribútumok és függvénydeklarációk sorozata

- Attribútumok

```
-module(Modname).
-export([F1/Arity1,...]).
-import(Modname,[F1/Arity1,...])

-compile(Opciók).
-include("filename.hrl").
-define(Makrónév,Helyettesítés).
-undef,-ifdef,-ifndef,-else,-endif
-vsn(Verzióleírás).
-saját_attribútum(Info).
```

Modulnév: atom

Kívülről is látható függvények listája

Más modulok modulnév nélkül használható függvényeinek listája

Fordítási opciók

Rekord definíciós fájl beemelése

Makró manipuláció

Feltételes fordítás

Verzióinfó

Bővíthető saját attribútummal

- Modulinformációk lekérdezése: `m(Modname)`, `Modname:module_info()`

```
2> m(khf).
...
3> khf:module_info().
...
```

## Rekord

- Ha egy ennesnek sok a tagja, nehéz felidézni, melyik tag mit jelent
- Ezért vezették be a rekordot - bár önálló rekordtípus nincs
- Rekord = címkézett ennes; szintaktikai édesítőszers
- $n$  mezejű rekord =  $n + 1$  elemű ennes:  $\{\text{rekordnév}, m_1, \dots, m_n\}$
- Rekord deklaráció (csak modulban!):  
`-record(rn, {p1=d1, ..., pn=dn}),`  
 ahol
  - $r_n$ : rekordnév,
  - $p_j$ : mezőnév,
  - $d_j$ : alapértelmezett érték (opcionális).
- Rekord létrehozása és változóhoz kötése:  
 $X = \#rn\{m_1=v_1, \dots, m_n=v_n\}$
- Egy mezőérték lekérdezése:  $X\#rn.m_i$
- Egy/több mezőérték változóhoz kötése:  $\#rn\{m_2=V, m_4=W\} = X$

## Rekord: példák

A `todo.hrl` rekorddefiníciós fájl tartalma:

```
-record(todo, {sts=remind, who='HP', txt}).
```

- Csak így használhatja több Erlang modul ugyanazt a rekorddefiníciót  
`(-include("todo.hrl"). attribútum)`
- Deklaráció beolvasása (csak Erlang shellben!)  

```
1> rr("todo.hrl").
[todo]

Új, alapértelmezett rekord (X) létrehozása
2> X0 = #todo{}.
#todo{sts = remind, who = 'HP', txt = undefined}

X1 is új
3> X1 = #todo{sts=urgent, who='KR', txt="Fóliák!"}.
#todo{sts = urgent, who = 'KR', txt = "Fóliák!"}

Rekord (X1) másolása frissítéssel; X2 is új
4> X2 = X1#todo{sts=done}.
#todo{sts = done, who = 'KR', txt = "Fóliák!"}
```

## Rekord: további példák

## ● Mezőértékek lekérdezése

```
5> #todo{who=W,txt=T} = X2.
#todo{sts = done,who = 'KR',txt = "Fóliák!"}

6> W.
'KR'

7> T.
"Fóliák!"

8> X1#todo.sts.
urgent
```

## ● Rekorddeklaráció elfelejtése

```
9> rf(todo).
ok

10> X2.
{todo,done,'KR',"Fóliák!"}
```

A rekord az Erlangon belül: ennes.

## Tartalom

## 7 Haladó Erlang

- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Rekurzió fajtái
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Lusta farkú lista Erlangban

## Lineáris rekurzív adatstruktúrák – Verem (Stack)

- Lista: rekurzív adatstruktúra: `@type list() = [] | [any()|list()]`
- Verem: ennessel valósítjuk meg, listával triviális lenne
- Műveletek: üres verem létrehozása, verem üres voltának vizsgálata, egy elem berakása, utoljára berakott elem leválasztása

stack.erl

```
%% @type stack() = empty | {any(),stack()}
```

```
empty() -> empty.
```

```
is_empty(empty) -> true;
is_empty({_,_}) -> false.
```

```
push(X, empty) -> {X,empty};
push(X, {_X,_S}=S) -> {X,S}. % {_X,_S}=S: réteges minta
```

```
pop(empty) -> error;
pop({X,S}) -> {X,S}.
```

## Verem példák

```
2> S1 = stack:push(1, stack:empty()).
{1,empty}
3> S2 = stack:push(2, S1).
{2,{1,empty}}
4> S3 = stack:push(3, S2).
{3,{2,{1,empty}}}
```

- Pl. megfordíthatunk egy listát; 1. lépés: verembe tesszük az elemeket

```
5> Stack = lists:foldl(fun stack:push/2, stack:empty(), "szoveg").
{103,{101,{118,{111,{122,{115,empty}}}}}}
```

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

stack.erl – folytatás

```
% to_list(S) az S verem elemeit tartalmazó lista LIFO sorrendben.
to_list(empty) -> [];
to_list({X,S}) -> [X|to_list(S)].
```

```
6> stack:to_list(Stack).
"gevozs"
```

## Elágazó rekurzív adatstruktúrák (pl. bináris fa)

- Műveletek bináris fák: létrehozása, mélysége, leveleinek száma

tree.erl – Műveletek bináris fák: létrehozása, mélysége, leveleinek száma

```
% @type btree() = leaf | {any(),btree(),btree()}.
```

```
empty() -> leaf. % Üres fa.
```

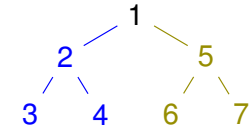
```
node(V, Lt, Rt) -> {V,Lt,Rt}. % Lt és Rt fák összekapcsolása
 % egy új V értékű csomóponttal.
```

```
depth(leaf) -> 0; % Fa legnagyobb mélysége.
depth({_,Lt,Rt}) -> 1 + erlang:max(depth(Lt), depth(Rt)).
```

```
leaves(leaf) -> 1; % Fa leveleinek száma.
leaves({_,Lt,Rt}) -> leaves(Lt) + leaves(Rt).
```

## Bináris fa (folyt.): listából fa, fából lista

```
L=empty(), T=node(1, node(2, node(3,L,L),
 node(4,L,L)),
 node(5, node(6,L,L),
 node(7,L,L)))
```



$T \mapsto \{1, \{2, \{3, \text{leaf}, \text{leaf}\}, \{4, \text{leaf}, \text{leaf}\}\}, \{5, \{6, \text{leaf}, \text{leaf}\}, \{7, \text{leaf}, \text{leaf}\}\}$

tree.erl – folytatás

```
to_list_prefix(leaf) -> [];
to_list_prefix({V,Lt,Rt}) ->
 [V] ++ to_list_prefix(Lt) ++ to_list_prefix(Rt).
```

```
to_list_infix(leaf) -> [];
to_list_infix({V,Lt,Rt}) ->
 to_list_infix(Lt) ++ ([V] ++ to_list_infix(Rt)).
```

```
from_list([]) -> empty();
from_list(L) -> {L1,[X|L2]} = lists:split(length(L) div 2, L),
 node(X, from_list(L1), from_list(L2)).
```

## Elágazó rekurzív adatstruktúrák – könyvtárszerkezet

```
2> Home = {d,"home", % home
 [{d,"kitti", % home/kitti
 [{d,".firefox",[]}, % home/kitti/.firefox
 {f,"dir.erl"}, % home/kitti/dir.erl
 {f,"khf1.erl"}, % home/kitti/khf1.pl
 {f,"khf1.pl"}]}, % home/kitti/khf1.erl
 {d,"ludvig",[]]}]. % home/ludvig
```

dir.erl – Könyvtárszerkezet kezelése

```
% @type tree() = file() | directory().
% @type file() = {f, name()}.
% @type directory() = {d, name(), [tree()]}.
% @type name() = string().
```

```
% Fa mérete (könyvtárak és fájlok számának összege).
```

```
count({f,_}) -> 1;
count({d,_,L}) -> 1 + lists:sum([count(I) || I <- L]).
```

## Könyvtárszerkezet – folytatás

dir.erl – Könyvtárszerkezet kezelése (folytatás)

```
% @spec subtree(Path::[name()], Tree::tree()) -> tree() | notfound.
% Tree fa Path útvonalon található részfája.
```

```
subtree([Name], {f,Name} = Tree) -> Tree;
subtree([Name], {d,Name,_} = Tree) -> Tree;
subtree([Name| [Sub|_] = SubPath], {d,Name,L}) ->
 case lists:keyfind(Sub, 2, L) of
 false -> notfound;
 SubTree -> subtree(SubPath, SubTree)
 end;
subtree(_, _) -> notfound.
```

```
3> dir:subtree(string:tokens("home/kitti/.firefox", "/"), Home).
{d,".firefox",[]}
```

```
4> dir:subtree(string:tokens("home/kitti/firefox", "/"), Home).
notfound
```



## Tartalom

### 7 Haladó Erlang

- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Rekurzió fajtái
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Lusta farkú lista Erlangban

## Rekurzió alapesetek

### • Lineáris rekurzió

Példa: lista összegének meghatározása

`rek.erl` – Rekurzió példák

```
sum([]) -> 0;
sum([H|T]) -> H + sum(T).
```

### • Elágazó rekurzió (Tree recursion)

Példa: bináris fa leveleinek száma

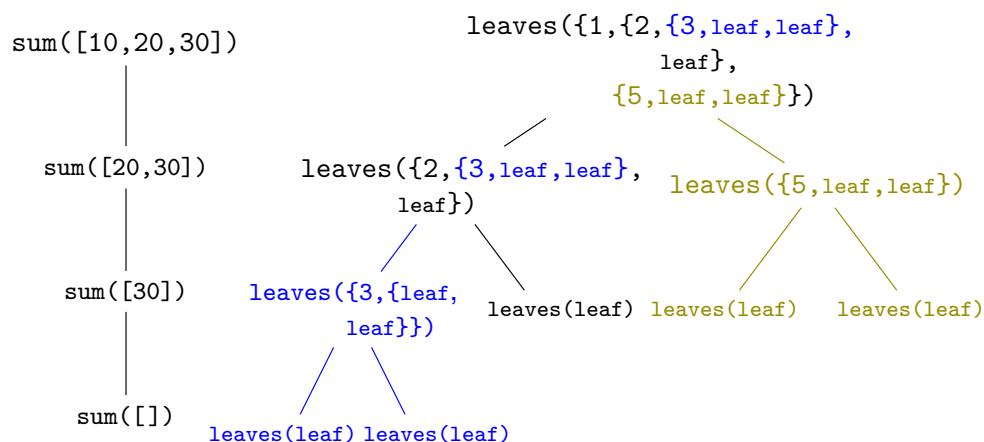
```
% @type btree() = leaf | {any(), btree(), btree()}.
leaves(leaf) -> 1;
leaves({_, Lt, Rt}) -> leaves(Lt) + leaves(Rt).
```

- Mindkettőből *rekurzív folyamat* jön létre, ha alkalmazzuk: minden egyes rekurzív hívás mélyíti a vermet
- Például `sum/1` az egész listát kiteríti a vermen: `sum([1,2,3]) →`

`1 + sum([2,3]) → 1 + (2 + sum([3])) → 1 + (2 + (3 + sum([])))`

## Rekurzív folyamat erőforrásigénye

- Hívási fa (call graph, CG): futás során meghívott függvények



- A lépések száma főként a *CG méretének* függvénye
- A tárigény (veremigény) főként a *CG mélységének* függvénye<sup>7</sup>

<sup>7</sup>itt lineáris függvénye

## Jobbrekurzió, iteráció

Gyakran érdemes akkumulátorok bevezetésével jobbrekurzióvá alakítani

- Példa: lista összegének meghatározása

```
sumi(L) -> sumi(L,0).
sumi([], N) -> N;
sumi([H|T], N) -> sumi(T, N+H).
```

- A segédfüggvényt jobb nem exportálni, hogy elrejtjük az akkumulátort
- A jobbrekurzióból *iteratív folyamat* hozható létre, amely nem mélyíti a vermet: `sumi/2` tárigénye konstans: `sumi([1,2,3],0) →`  
`sumi([2,3],1) → sum([3],3) → sum([],6)`
- Ne tévesszük össze egymással a rekurzív számítási folyamatot és a rekurzív függvényt, eljárást!
  - Rekurzív függvény esetén csupán a szintaxisról van szó, arról, hogy hivatkozik-e a függvény, eljárás *önmagára*
  - Folyamat esetében viszont a folyamat menetéről, lefolyásáról beszélünk
- Ha egy függvény *jobbrekurzív (tail-recursive)*, a megfelelő folyamat – az értelmező/fordító jószágától függően – lehet iteratív

## Rekurzív folyamat erőforrásigénye – Példák

| Példa <sup>8</sup>                                          | Lépések<br>(CG méret) | CG mélység           | Tárigény $\approx$<br>mélység* állapot |
|-------------------------------------------------------------|-----------------------|----------------------|----------------------------------------|
| <code>sum(lists:seq(1, n))</code>                           | $\Theta(n)$           | $\Theta(n)$          | $\Theta(n) \cdot \Theta(\log n)$       |
| <code>sumi(lists:seq(1, n))</code>                          | $\Theta(n)$           | $\Theta(1)$ (konst.) | $\Theta(1) \cdot \Theta(\log n)$       |
| SEND+MORE=MONEY kimerítő keresés, itt $n = 8$               | $\Theta(10^n)$        | $\Theta(n)$          | $\Theta(n) \cdot \Theta(\log n)$       |
| Mastermind $(m, h)$ kimerítő keresés, tipikusan $h \leq 20$ | $\Theta(m^h)$         | $\Theta(h)$          | $\Theta(h \cdot mh \log m)$            |

- A rekurzióból fakadó tárigény lehet jelentős is (vö `sum/1`, `sumi/1`), és lehet elhanyagolható is a lépésekhez képest (SMM, Sudoku, MMind)
- Az eljárások, függvények olyan *minták*, amelyek megszabják a számítási folyamatok, processzek menetét, *lokális* viselkedését
- Egy számítási folyamat *globális* viselkedését (pl. idő- és tárigény) általában nehéz megbecsülni, de törekednünk kell rá

<sup>8</sup> $f(n) = \Theta(g(n))$  jelentése:  $g(n) \cdot k_1 \leq f(n) \leq g(n) \cdot k_2$  valamilyen  $k_1, k_2 > 0$ -ra

## Elágazó rekurzió példa: Fibonacci-sorozat

Kiegészítő anyag

- Amikor hierarchikusan strukturált adatokon kell műveleteket végezni, pl. egy fát kell bejárni, akkor az elágazó rekurzió nagyon is természetes és hasznos eszköz
- Az elágazó rekurzió numerikus számításoknál az algoritmus első megfogalmazásakor is hasznos lehet; példa: írjuk át a Fibonacci-számok matematikai definícióját programmá – 0,0,1,1,2,3,5,8,13,...

$$F(n) = \begin{cases} 0 & \text{ha } n = 0, \\ 1 & \text{ha } n = 1, \\ F(n-1) + F(n-2) & \text{különben.} \end{cases}$$

Naív Fibonacci, előfelt.:  $N \in \mathbb{N}$ 

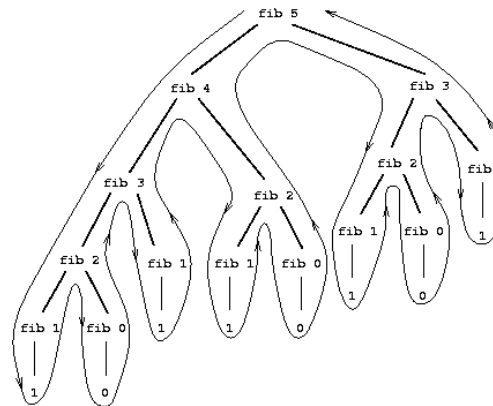
```
fib(0) -> 0;
fib(1) -> 1;
fib(N) -> fib(N-2) + fib(N-1).
```

- Ha már értjük a feladatot, az első, rossz hatékonyságú változatot könnyebb átírni jó, hatékony programmá. Az elágazó rekurzió segíthet a feladat megértésében.
- Hivatkozás:* Structure and Interpretation of Computer Programs, 2nd ed., by H. Abelson, G. J. Sussman, J. Sussman, The MIT Press, 1996

A jobbrekurzió mindig *nagyságrendekkel* előnyösebb? Nem!

- A jobbrekurzív `sumi(L1)` össz-tárigénye konstans (azaz  $\Theta(1)$ ), a lineáris-rekurzív `sum(L1)` össz-tárigénye  $\Theta(\text{length}(L1))$
- Melyiknek alacsonyabb az össz-tárigénye?
  - `bevezeto:append(L1,L2)`
  - `R1=list:reverse(L1)`, `bevezeto:revapp(R1,L2)` *% jobbrek.*
- `append` kiteríti `L1` elemeit a vermen, ennek tárigénye  $\Theta(\text{length}(L1))$ , majd ezeket `L2` elé fűzi, így tárigénye  $\Theta(\text{length}(L1)+\text{length}(L2))$
- `revapp(R1,L2)` iteratív számítási folyamat, nem mélyíti a vermet, *de* `revapp` felépíti az `L1++L2` akkumulátort, ennek tárigénye szintén  $\Theta(\text{length}(L1)+\text{length}(L2))$
- A jobbrekurzív `revapp` össz-tárigénye *nagyságrendileg* hasonló, mint a lineáris-rekurzív `append` függvényé!
- Ha az akkumulátor mérete nem konstans (azaz  $\Theta(1)$ ), meggondolandó a jobbrekurzió...

## Elágazó rekurzió példa: Fibonacci-sorozat (2)



- Elágazóan rekurzív folyamat hívási fája `fib(5)` kiszámításakor
- Alkalmatlan a Fibonacci-számok előállítására
- A  $F(n)$  meghatározásához pontosan  $F(n+1)$  levélből álló fát kell bejárni, azaz ennyiszer kell meghatározni  $F(0)$ -at vagy  $F(1)$ -et

- $F(n)$  exponenciálisan nő  $n$ -nel:  $\lim_{n \rightarrow \infty} \frac{F(n+1)}{F(n)} = \varphi$ , ahol  $\varphi = (1 + \sqrt{5})/2 \approx 1.61803$ , az *aranymetszés* arányszáma

## Elágazó rekurzió példa: Fibonacci-sorozat (3)

- A lépések száma tehát  $F(n)$ -nel együtt exponenciálisan nő  $n$ -nel
- A tárigény ugyanakkor csak lineárisan nő  $n$ -nel, mert csak azt kell nyilvántartani, hogy hányadik szinten járunk a fában
- A Fibonacci-számok azonban lineáris-iteratív folyamattal is előállíthatók: ha az  $A$  és  $B$  változók kezdőértéke rendre  $F(1) = 1$  és  $F(0) = 0$ , és ismétlődően alkalmazzuk az  $A \leftarrow A + B$ ,  $B \leftarrow A$  transzformációkat, akkor  $N$  lépés után  $A = F(N + 1)$  és  $B = F(N)$  lesz

```
fibi(0) -> 0; % fibi(N) az N. Fibonacci-szám.
fibi(N) -> fibi(N-1, 1, 0).
```

```
% fibi(N,A,B) az A←A+B, B←A trafó N-szeri ismétlése utáni A.
fibi(0, A, _B) -> A;
fibi(I, A, B) -> fibi(I-1, B+A, A).
```

- A Fibonacci-példában a lépések száma elágazó rekurzióval  $n$ -nel exponenciálisan, lineáris rekurzióval  $n$ -nel arányosan nőtt!
- Pl. a `tree:leaves/1` függvény is lineáris rekurzióvá alakítható, de ezzel már nem javítható a hatékonysága: valamilyen LIFO tárolót kellene használni a mélységi bejáráshoz a rendszer stackje helyett

## Programhelyesség informális igazolása

- Egy rekurzív programról is be kell látnunk, hogy
  - funkcionálisan helyes (azaz azt kapjuk eredményül, amit várunk),
  - a kiértékelése biztosan befejeződik (nem „végtelen” a rekurzió).
- Ellenpélda: `fib(-1)` végtelen ciklus, bár a paraméter „csökken”!
- Bizonyítása rekurzió esetén egyszerű, *strukturális indukcióval* lehetséges, azaz visszavezethető a teljes indukcióra valamilyen *strukturális tulajdonság* szerint
- Csak meg kell választanunk a strukturális tulajdonságot, amire vonatkoztatjuk az indukciót; pl. a `fib/1` az  $N = 0$  paraméterre leáll, de a 0 nem a legkisebb egész szám: a *nemnegatív számok halmazában* viszont a legkisebb  $\rightarrow$  módosítani kell az értelmezési tartományt
- A `map` példáján mutatjuk be

## Programhelyesség informális igazolása (folyt.)

```
% @spec map(fun(A) -> B, [A]) -> [B].
map(_F, []) -> [];
map(F, [X|Xs]) -> [F(X)|map(F, Xs)].
```

- 1 A strukturális tulajdonság itt a lista hossza
- 2 A függvény funkcionálisan helyes, mert
  - belátjuk, hogy a függvény jól transzformálja az üres listát;
  - belátjuk, hogy az  $F$  jól transzformálja a lista első elemét (a fejét);
  - indukciós feltevés: a függvény jól transzformálja az eggyel rövidebb listát (a lista farkát);
  - belátjuk, hogy a fej transzformálásával kapott elem és a fark transzformálásával kapott lista összefűzése a várt listát adja.
- 3 A kiértékelés véges számú lépésben befejeződik, mert
  - a lista (mohó kiértékelés mellett!) véges,
  - a függvényt a *rekurzív ágba*n minden lépésben egyre rövidülő listára alkalmazzuk (strukturális tulajdonság csökken), és
  - a rekurziót előbb-utóbb leállítjuk (ui. kezeljük az *alapesetet*, ahol a strukturális tulajdonság zérus, van rekurziót nem tartalmazó klóz).

## Tartalom

- 7 Haladó Erlang
  - Kivételkezelés
  - Rekord
  - Rekurzív adatstruktúrák
  - Rekurzió fajtái
  - Halmazműveletek (rendezetlen listával)
  - Generikus keresőfák
  - Lusta farkú lista Erlangban

## Tagsági vizsgálat

- A halmazt itt egy rendezetlen listával ábrázoljuk
- A műveletek sokkal hatékonyabbak volnának rendezett adatszerkezettel (pl. rendezett lista, keresőfa, hash)
- Erlang STDLIB: sets, ordsets, gb\_sets modulok

### set.erl – Halmazkezelő függvények

```
% @type set() = list().
```

```
% empty() az üres halmaz.
```

```
empty() -> [].
```

% Az absztrakció miatt szükséges:  
% ábrázolástól független interfész.

```
% isMember(X, Ys) igaz, ha az X elem benne van az Ys halmazban.
```

```
isMember(_, []) -> false;
isMember(X, [Y|Ys]) ->
 X:=Y orelse isMember(X, Ys).
```

- **Megjegyzés:** orelse lusta kiértékelésű

## Halmazműveletek

- Öt ismert halmazműveletet definiálunk a továbbiakban (rendezetlen listákkal ábrázolt halmazokon):
  - unió (union,  $S \cup T$ )      vö. lists:fold\*/3
  - metszet (intersect,  $S \cap T$ )      vö. lists:filter/2
  - részhalmaz-e (isSubset,  $T \subseteq S$ )      vö. lists:all/2
  - egyenlők-e (isEqual,  $S \equiv T$ )
  - hatványhalmaz (powerSet,  $2^S$ )
- Otthoni gyakorlásra: halmazműveletek megvalósítása rendezett listákkal, illetve fákkal.  
A vizsgán lehetnek ilyen feladatok...

## Új elem berakása egy halmazba, listából halmaz

- newMember új elemet rak egy halmazba, *ha még nincs benne*

### set.erl – folytatás

```
% @spec newMember(X::any(), Xs::set()) -> Xs2::set().
% Xs2 halmaz az Xs halmaz és az [X] halmaz uniója.
```

```
newMember(X, Xs) ->
 case isMember(X, Xs) of
 true -> Xs;
 false -> [X|Xs]
 end.
```

- listToSet listát halmazzá alakít a duplikátumok törlésével; naív (lassú)

```
% @spec listToSet(list()) -> set().
% listToSet(Xs) az Xs lista elemeinek halmaza.
```

```
listToSet([]) -> [];
listToSet([X|Xs]) ->
 newMember(X, listToSet(Xs)).
```

## Unió, metszet

### set.erl – folytatás

```
% @spec union(Xs::set(), Ys::set()) -> Zs::set().
% Zs az Xs és Ys halmazok uniója.
```

```
union([], Ys) -> Ys;
union([X|Xs], Ys) ->
 newMember(X, union(Xs, Ys)).
```

```
union2(Xs, Ys) ->
 foldr(fun newMember/2, Ys, Xs).
```

```
% @spec intersect(Xs::set(), Ys::set()) -> Zs::set().
% Zs az Xs és Ys halmazok metszete.
```

```
intersect([], _) -> [];
intersect([X|Xs], Ys) ->
 Zs = intersect(Xs, Ys),
 case isMember(X, Ys) of
 true -> [X|Zs];
 false -> Zs
 end.
```

```
intersect3(Xs, Ys) ->
 [X || X <- Xs,
 isMember(X, Ys)
].
```

## Részhalmaza-e, egyenlők-e

## set.erl – folytatás

```
% @spec isSubset(Xs::set(), Ys::set()) -> B::bool().
% B igaz, ha Xs részhalmaza Ys-nek.
isSubset([], _) ->
 true;
isSubset([X|Xs], Ys) ->
 isMember(X, Ys) andalso isSubset(Xs, Ys).

% @spec isEqual(Xs::set(), Ys::set()) -> B::bool().
% B igaz, ha Xs és Ys elemei azonosak.
isEqual(Xs, Ys) ->
 isSubset(Xs, Ys) andalso isSubset(Ys, Xs).
```

- isSubset lassú a rendezetlenség miatt
- andalso lusta kiértékelésű
- A listák egyenlőségének vizsgálata ugyan beépített művelet az Erlangban, halmazokra mégsem használható, mert pl. [3,4] és [4,3] listaként különböznek, de halmazként egyenlők.

## Halmaz hatványhalmaza

- Az  $S$  halmaz hatványhalmazának nevezzük az  $S$  összes részhalmazának a halmazát, jelölés itt:  $2^S$
- $S$  hatványhalmazát *rekurzívan* például úgy állíthatjuk elő, hogy kiveszünk  $S$ -ből egy  $x$  elemet, majd előállítjuk az  $S \setminus \{x\}$  hatványhalmazát
- Például  $S = \{10, 20, 30\}$ ,  $x \leftarrow 10$ ,  $2^{S \setminus \{x\}} = \{\{\}, \{20\}, \{30\}, \{20, 30\}\}$
- Ha tetszőleges  $T$  halmazra  $T \subseteq S \setminus \{x\}$ , akkor  $T \subseteq S$  és  $T \cup \{x\} \subseteq S$ , azaz mind  $T$ , mind  $T \cup \{x\}$  eleme  $S$  hatványhalmazának
- Vagyis  $2^{\{10, 20, 30\}} =$

$$\{\{\}, \{20\}, \{30\}, \{20, 30\}\} \cup \{\{\}\cup\{10\}, \{20\}\cup\{10\}, \{30\}\cup\{10\}, \{20, 30\}\cup\{10\}\}$$

## % powerSet\*(S) az S halmaz hatványhalmaza.

```
powerSet1([]) ->
 [[]];
powerSet1([X|Xs]) ->
 P = powerSet1(Xs),
 P ++ [[X|Ys] || Ys <- P].

powerSet2(Xs) -> % jobbrekurzívan
 foldl(fun(X, P) ->
 P ++ [[X|Ys] || Ys <- P],
 [[]],
 Xs).
```

## Halmaz hatványhalmaza – hatékonyabb változat

- $A \text{ P } ++ [ [X|Ys] || Ys <- P ]$  művelet hatékonyabbá tehető

## set.erl – folytatás

```
% @spec insAll(X::any(), Yss::[[any()]], Zss::[[any()]]) -> Xss::[[any()]].
% Xss az Yss lista Ys elemeinek Zss elé fűzött
% listája, amelyben minden Ys elem elé X van beszúrva.
insAll(_X, [], Zss) ->
 Zss;
insAll(X, [Ys|Yss], Zss) ->
 insAll(X, Yss, [[X|Ys]|Zss]).

powerSet3([]) ->
 [[]];
powerSet3([X|Xs]) ->
 P = powerSet3(Xs),
 insAll(X, P, P). % [[X|Ys] || Ys <- P] ++ P kiváltására
```

## Tartalom

- 7 Haladó Erlang
  - Kivételkezelés
  - Rekord
  - Rekurzív adatstruktúrák
  - Rekúzió fajtái
  - Halmazműveletek (rendezetlen listával)
  - Generikus keresőfák
  - Lusta farkú lista Erlangban

## Generikus keresőfák Erlangban

## Tartalom

Lásd a leírást a <http://dp.iit.bme.hu/dp08a/gtree.pdf> (1–5. oldalak), a futtatható példaprogramokat a `gtree.erl` fájlban.

Megjegyzések:

- `gtree:set_to_list/1` funkcióban azonos `tree:to_list_infix/1` függvénnyel, de hatékonyabb: nincs benne összefűzés ( $L1++L2$ ), csak építés ( $[H|T]$ )
- ```
1> gtree:list_to_set([3,1,5,4,2,1]).
{3,{1,leaf,{2,leaf,leaf}},{5,{4,leaf,leaf},leaf}}
2> io:format("~10p~n",
    [gtree:list_to_map([{{3,a},{1,a},{5,a},{4,b},{2,b},{1,x}}])]).
{{3,a},
 {{1,x},
  leaf,
  {{2,b},
   leaf,
   leaf}}},
 ...
```

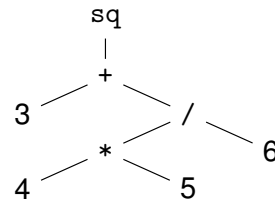
7 Haladó Erlang

- Kivételkezelés
- Rekord
- Rekurzív adatstruktúrák
- Rekurzív fajtái
- Halmazműveletek (rendezetlen listával)
- Generikus keresőfák
- Lusta farkú lista Erlangban

Összetett kifejezés kiértékelése

- Egy összetett kifejezést az Erlang két lépésben értékeli ki, **mohó** kiértékeléssel; az alábbi rekurzív kiértékelési szabállyal:
 - Először** kiértékeli az operátort (műveleti jelet, függvényjelet) és az argumentumait,
 - majd **ezután** alkalmazza az operátort az argumentumokra.
- A kifejezéseket *kifejezésfával* ábrázolhatjuk
- Hasonló a Prolog-kifejezés ábrázolásához:


```
| ?- write_canonical(sq(3+4*5/6)).
sq(+ (3, / (* (4, 5), 6)))
```
- A mohó kiértékelés során az operandusok alulról fölfelé „terjednek”
- Felhasználói függvény mohó alkalmazása (fent 2. pont):
 - a függvény törzsében a formális paraméterek összes előfordulását lecseréli a megfelelő aktuális paraméterre,
 - majd kiértékeli a függvény törzsét.



Függvényalkalmazás mohó kiértékelése

Tekintsük a következő egyszerű függvények definícióját:

```
sq(X) -> X * X.
sumsq(X, Y) -> sq(X) + sq(Y).
f(A) -> sumsq(A+1, A*2).
```

Mohó kiértékelés esetén minden lépésben egy részkifejezést egy vele egyenértékű kifejezéssel helyettesítünk. Pl. az $f(5)$ mohó kiértékelése:

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

- A függvényalkalmazás itt bemutatott *helyettesítési modellje*, az „egyenlők helyettesítése egyenlőkkel” (*equals replaced by equals*) segíti a függvényalkalmazás *jelentésének* megértését
- Olyan esetekben alkalmazható, amikor egy függvény *jelentése független* a környezetétől (pl. ha minden mellékhatás kizárva)
- Az fordítók rendszerint bonyolultabb modell szerint működnek

Lusta kiértékelés

- Az Erlang tehát először kiértékeli az operátort és az argumentumait, majd alkalmazza az operátort az argumentumokra
- Ezt a kiértékelési sorrendet *mohó* (eager) vagy *applikatív sorrendű* (applicative order) kiértékelésnek nevezzük
- Van más lehetőség is: a kiértékelést addig halogatjuk, ameddig csak lehetséges: ezt *lusta* (lazy), *szükség szerinti* (by need) vagy *normál sorrendű* (normal order) kiértékelésnek nevezzük
- Példa: az $f(5)$ lusta kiértékelése:

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

- Példa: a `false andalso f(5) > 100` lusta kiértékelése:

```
false andalso f(5) > 100 → false
```

Mohó és lusta kiértékelés

- Igazolható, hogy olyan függvények esetén, amelyek jelentésének megértésére a helyettesítési modell alkalmas, a kétféle kiértékelési sorrend azonos eredményt ad
- Vegyük észre, hogy lusta (szükség szerinti) kiértékelés mellett egyes részkifejezéseket néha többször is ki kell értékelni
- A többszörös kiértékelést jobb értelmezők/fordítók (pl. Alice, Haskell) úgy kerülik el, hogy az azonos részkifejezéseket megjelölik, és amikor egy részkifejezést először kiértékelnek, az *eredményét megjegyzik*, a többi előfordulásakor pedig ezt az eredményt veszik elő. E módszer hátránya a nyilvántartás szükségessége. Ma általában ezt nevezik *lusta* kiértékelésnek.

Lusta kiértékelés Erlangban: lusta farkú lista

- Ismétlés: `% @type erlang:list() = [] | [any()|list()].`
- A `[H|T]` egy speciális szintaxisú kételemű ennes, nemcsak listákra használhatjuk:


```
1> [1|[2]].
[1,2] % Lista, mert a / utáni rész lista.
2> [1|[2|[]]].
[1,2] % Lista, mint az előző.
3> [1|2].
[1|2] % Egy kételemű ennes, mert a / utáni rész nem lista.
```
- A következő főlíákon az átláthatóság kedvéért a listaszintaxist használjuk egy kételemű ennesre, a lusta listára

`lazy.erl` – Lusta farkú lista

```
% @type lazy:list() = [] | [any()|fun() -> lazy:list()].
```

- A fenti szerkezetben a második tag (farok) *késleltett kiértékelésű* (delayed evaluation)
- Teljesen lusta lista: `verylazy.erl` (nem tananyag)


```
% @type verylazy:list() = fun() -> ([] | [any()|verylazy:list()]).
```

Lusta farkú lista építése

- Végtelen számsorozat:

`lazy.erl` – folytatás

```
% @spec infseq(N::integer())
%       -> lazy:list().
infseq(N) ->
    [N|fun() -> infseq(N+1) end].
```

- Példák:

```
1> lazy:infseq(0).
[0|#Fun<lazy.1.65678590>]
2> T1 = tl(lazy:infseq(0)).
#Fun<lazy.1.65678590>
3> T1().
[1|#Fun<lazy.1.65678590>]
```

- Véges számsorozat:

`lazy.erl` – folytatás

```
% @spec seq(M::integer(),
%         N::integer())
%       -> lazy:list().
seq(M, N) when M =< N ->
    [M|fun() -> seq(M+1, N) end];
seq(_, _) ->
    [].
```

- Példák:

```
1> lazy:seq(1,1).
[1|#Fun<lazy.0.35745118>]
2> tl(lazy:seq(1,1)).
#Fun<lazy.0.35745118>
3> (tl(lazy:seq(1,1)))().
[]
```

Erlang-lista konvertálása

Erlang-listából lusta lista:

- Nagyon gyors: egyetlen függvényhívás

```
% @spec cons(erlang:list())
%       -> lazy:list().
cons([]) ->
    [];
cons([H|T]) ->
    [H|fun() -> cons(T) end].
```

```
1> lazy:cons([1,2]).
[1|#Fun<lazy.10.66878903>]
2> T2 = tl(lazy:cons([1,2])).
#Fun<lazy.10.66878903>
3> T2().
[2|#Fun<lazy.10.66878903>]
4> (tl(T2()))().
[]
```

Lusta listából Erlang-lista:

- Csak az első N elemét értékeljük ki: lehet, hogy végtelen

```
% @spec take(lazy:list(),
%           N::integer())
%       -> erlang:list().
take(_, 0) -> [];
take([], _) -> [];
take([H|_], 1) -> [H]; % optim.
take([H|T], N) ->
    [H|take(T(), N-1)].
```

```
1> lazy:take(lazy:infseq(0), 5).
[0,1,2,3,4]
2> lazy:take(lazy:seq(1,2), 5).
[1,2]
```

- Ha $N=1$, a $T()$ hívás felesleges

Gyakori függvények lusta listára adaptálva – map

- Motiváció: listanézet nem alkalmazható; a lusta szintaxis elrejtése

lazy.erl – folytatás

```
% @spec map(fun(), lazy:list()) -> lazy:list().
map(_, []) -> [];
map(F, [H|T]) -> [F(H)|fun() -> map(F, T()) end].
```

```
1> F = fun(X) -> io:format("Hivas: F(~p)~n", [X]), math:exp(X) end.
#Fun<erl_eval.6.80247286>
2> F(1).
Hivas: F(1)
2.718281828459045
3> L = lazy:map(F, lazy:infseq(1)).
Hivas: F(1)
[2.718281828459045|#Fun<lazy.5.87890739>]
4> lazy:take(L, 3).
Hivas: F(2)
Hivas: F(3)
[2.718281828459045, 7.38905609893065, 20.085536923187668]
```

Gyakori függvények lusta listára adaptálva – iteratív sum

- Lista összegzése (csak véges listára)

lazy.erl – folytatás

```
sum(L) -> sum(L, 0).
```

```
% @spec sum(lazy:list(), number()) -> number().
sum([], X) -> X;
sum([H|T], X) -> sum(T(), H+X). % jobbrekurzív!
```

- Összehasonlítás:

- `lists:sum(lists:seq(1, N=10000000))`
mohó, gyors⁹, tárigénye lineáris N -ben
- `lazy:sum(lazy:seq(1, N=10000000))`
lusta, lassabb, tárigénye kb. konstans (korlátos számok esetén)

- Általánosabban a lusta lista és a mohó Erlang-lista összehasonlítása:

- Tárigénye csak a kiértékelt résznek van
- Lusta lista *teljes* kiértékelése sokkal lassabb is lehet (késleltetés)
- De időigénye alacsonyabb *lehet*, ha nem kell teljesen kiértékelni

⁹ha nem itt kell létrehozni a listát; a példában `lists:sum` gyors, `lists:seq` lassú

Gyakori függvények lusta listára adaptálva – filter, append

- Motiváció: listanézet, ++ nem alkalmazható; a lusta szintaxis elrejtése

lazy.erl – folytatás

```
% filter(fun(any()) -> bool(), lazy:list()) -> lazy:list().
% Kicsit mohó, az eredménylista fejéig kiértékeli a listát
filter(_, []) ->
```

```
    [];
filter(P, [H|T]) ->
    case P(H) of
        true -> [H|fun() -> filter(P, T()) end];
        false -> filter(P, T()) % megkeressük az eredmény fejét
    end.
```

```
% @spec append(lazy:list(), lazy:list()) -> lazy:list().
append([], L2) -> L2;
append([H|T], L2) -> [H|fun() -> append(T(), L2) end].
```

Nevezetes számsorozatok

Fibonacci-sorozat

```
% @spec fibs(integer(), integer()) -> lazy:list.
fibs(Cur, Next) -> [Cur|fun() -> fibs(Next, Cur+Next) end].
```

```
1> lazy:take(lazy:fibs(0,1),10).
[0,1,1,2,3,5,8,13,21,34]
```

Eratoszteniési szita

```
% @spec sift(Prime::integer(), L::lazy:list()) -> L2::lazy:list().
% L2 lista L lista azon elemei, melyek nem oszthatóak Prime-mal.
sift(Prime, L) -> filter(fun(N) -> N rem Prime /= 0 end, L).
```

```
% @spec sieve(L1::lazy:list()) -> L2::lazy:list().
% L2 lista az L1 végtelen lista szitálása (üres listára hibát ad).
sieve([H|T]) -> [H|fun() -> sieve(sift(H, T())) end].
```

```
1> lazy:take(lazy:sieve(lazy:infseq(2)),10).
[2,3,5,7,11,13,17,19,23,29]
```

Lusta append alkalmazása: lusta qsort

```
% Csak emlékeztetőül: @spec eqsort(erlang:list()) -> erlang:list().
eqsort([]) -> [];
eqsort([Pivot|Xs]) -> eqsort([X || X <- Xs, X < Pivot])
++ [Pivot|eqsort([X || X <- Xs, X >= Pivot])].
```

```
% @spec qsort(lazy:list()) -> lazy:list().
qsort([]) -> [];
qsort([Pivot|Xs]) ->
  io:format("hivas: qsort(~w~n", [take([Pivot|Xs], 100)]),
  Low = fun(X) -> X < Pivot end, High = fun(X) -> X >= Pivot end,
  append(qsort(filter(Low, Xs())),
  [Pivot|fun() -> qsort(filter(High, Xs())) end])).
```

```
1> L=cons([5,3,6,8,1,7]).
[5|#Fun<lazy.10.7...>]
2> S = qsort(L).
Hivas: qsort([5,3,6,8,1,7])
Hivas: qsort([3,1])
Hivas: qsort([1])
[1|#Fun<lazy.12.1...>]
```

```
3> take(S, 1).
[1]
4> take(S, 3).
[1,3,5]
5> take(S, 4).
Hivas: qsort([6,8,7])
[1,3,5,6]
```

```
6> take(qsort(L), 6).
Hivas: qsort([5,3,...])
Hivas: qsort([3,1])
Hivas: qsort([1])
Hivas: qsort([6,8,7])
Hivas: qsort([8,7])
Hivas: qsort([7])
[1,3,5,6,7,8]
```