

Deklaratív Programozás

Szeredi Péter¹ Hanák Péter²

¹szeredi@cs.bme.hu

BME Számítástudományi és Információelméleti Tanszék

²hanak@emt.bme.hu

BME Egészségipari Mérnöki Tudásközpont

2018. ősz

Az előadók köszönetüket fejezik ki Kápolnai Richárdnak

I. rész

Bevezetés

- 1 Bevezetés
- 2 Célja: deklaratív programozás C++-ban
- 3 Prolog alapok
- 4 Erlang alapok
- 5 Haladó Prolog
- 6 Haladó Erlang

A tárgy témája

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

Tartalom

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

Honlap, ETS, levelezési lista

- Honlap: <https://dp.iit.bme.hu>
a jelen félév honlapja: <https://dp.iit.bme.hu/dp-current>
- ETS, az Elektronikus TanárSegéd
<https://dp.iit.bme.hu/ets>
- Levelezési lista:
<http://lists.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 (egy kivételével angolul)

- Simon St. Laurent: *Introducing Erlang. Getting Started in Functional Programming*. O'Reilly, 2013.
<http://shop.oreilly.com/product/0636920025818.do>
- Learn You Some Erlang for great good! (online is olvasható)
<http://learnyousomeerlang.com>
- Joe Armstrong: *Programming Erlang. Software for a Concurrent World. Second Edition*. The Pragmatic Programmers, 2013.
<http://www.pragprog.com/book/jaerlang2/programming-erlang>
- Francesco Cesarini, Simon Thompson: *Erlang Programming*. O'Reilly, 2009.
<http://oreilly.com/catalog/9780596518189/>

További irodalom:

- On-line Erlang documentation
<http://erlang.org/doc.html> vagy `erl -man <module>`
- Wikibooks on Erlang Programming
http://en.wikibooks.org/wiki/Erlang_Programming
- ERLANG összefoglaló magyarul
<http://nyelvek.inf.elte.hu/leirasok/Erlang/>

Fordító- és értelmezőprogramok

- SICStus Prolog – 4.4 verzió (licenz az ETS-en keresztül kérhető)
A kiegészítő komponensek (Jasper, Tcl/Tk és ODBC) installálására nincs szükség, glibc esetén a megadottnál frissebb verzió is jó
- 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, Windows)
- Eclipse fejlesztői környezet (SPIDER, erlIDE)

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 PDF-ben
- 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)

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 teszt eset 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 KHF-k egyre összetettebbek és **egymásra épülnek** – érdemes **minél előbb** elkezdni a KHF-ek beadását!
- 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.)

Gyakorlatok

- 2. héttől kezdődően 2 órás gyakorlatok, az időpontok (hamarosan) olvashatók a honlapon
- Laptop használata megengedett
- További Prolog gyakorlási lehetőség az ETS rendszerben (gyakorló feladatok, lásd honlap)

Deklaratív programozás: követelmények (folyt.)

Nagyzárthelyi, pótzárthelyi (NZH, PZH, PPZH)

- A zárthelyi **kötelező**, kivéve megajánlott jegy esetén.
A megajánlott jegy feltételei:
 - Alapfeltételek: KHF-ek 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
- A zárthelyin semmilyen jegyzet, segédlet nem használható
- 40%-os szabály (nyelvenként a maximális részpontszám 40%-a kell az eredményességhez)
- NZH, PZH: várhatóan a 12.-13. héten, később meghirdetendő időpontban
- A PPZH-ra 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: 85% (a 100 pontból 85)

Tartalom

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

Bevezető példa: adott értékű kifejezés előállítása

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

Hogyan ábrázoljuk az aritmetikai kifejezéseket?

- Első ötlet: füzér (string)
- Egy könnyebben kezelhető, strukturált ábrázoláshoz írjuk fel az aritmetikai kifejezés – $\langle \text{akif} \rangle$ – szintaxisát:

$$\begin{aligned} \langle \text{akif} \rangle & ::= \langle \text{szám} \rangle \mid \\ & \quad (\langle \text{akif} \rangle \langle \text{művelet} \rangle \langle \text{akif} \rangle) \\ \langle \text{művelet} \rangle & ::= + \mid - \mid * \mid / \end{aligned}$$

(az egyértelműség kedvéért minden részkifejezést zárójelbe teszünk)

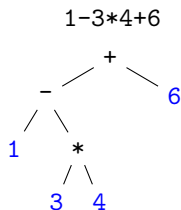
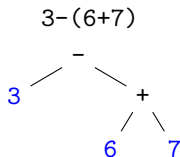
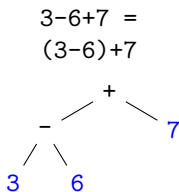
- Az $\langle \text{akif} \rangle$ adatstruktúra egy lehetséges megvalósítása C nyelven:

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

Az aritmetikai kifejezések matematikai absztrakciója

Milyen matematikai struktúra feleltethető meg $\langle \text{akif} \rangle$ -nek?

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



A bevezető példa megoldásának terve

- „Generate-and-test”: generáljuk le az adott levelekkel bíró összes $\langle \text{akif} \rangle$ -et, majd válogassuk ki azokat, amelyek értéke az adott szám
- Leegyszerűsítve, pl. levelek: 1, 2, 3; műveletek: csak a $-$ és a $*$
- Az összes $\langle \text{akif} \rangle$ előállítás (jelölje n a levelek, m a műveletek számát):
 - 1 Állítsuk elő az összes adott levélszámú címkézetlen bináris fát (jelölje f ezek számát)
 - 2 A csomópontokba minden lehetséges módon helyezzünk el műveleti jeleket ($f \cdot m^{n-1}$ fa)
 - 3 Állítsuk elő a levelek összes permutációját ($n!$ db.)
 - 4 Minden csomópont-címkézett fa leveleibe írjunk be minden permutációt ($f \cdot m^{n-1} \cdot n!$ darab $\langle \text{akif} \rangle$)
- Számítsuk ki az így kapott összes $\langle \text{akif} \rangle$ értékét, adjuk vissza azokat, amelyekre ez a megadott számmal egyezik

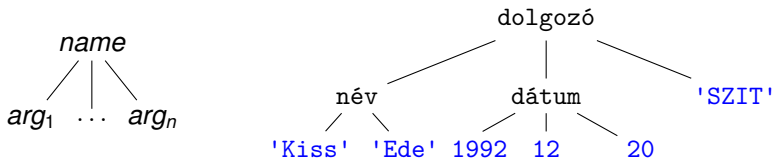
Tartalom

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

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(arg1, ..., argn)`
 - `name` egy névkonstans, az `argi` mezők tetsz. Prolog kifejezések
 - példa: `dolgozó(név('Kiss', 'Ede'), dátum(1992, 12, 20), 'SZIT')`.
 - Az összetett kifejezések valójában fastruktúrát alkotnak:

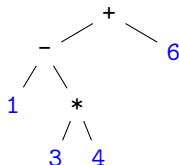


- a Prolog változó a matematikai változónak felel meg: **egy**, esetleg még ismeretlen adatot jelent, (legfeljebb) egyszer kaphat értéket; de megjelenhet összetett kifejezés részeként (pointer)

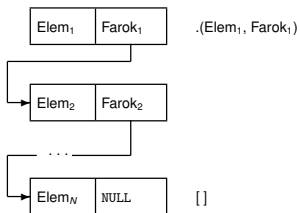
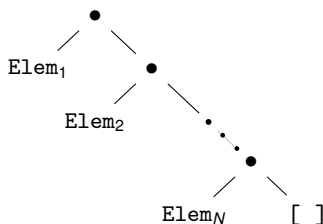
Szintaktikus „édesítőszerek” Prologban

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

```
| ?- 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 kifejezés, amely számokból a négy alpművelet (+, -, \*, /) segítségével épül fel (röviden, ha `X` **helyes**).

- 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 kifejezések ellenőrzése – továbbfejlesztett változat

- A kif Prolog eljárás segédeljárást használó változata:

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

- Az alap4 segédeljárás:

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

- Ekvivalens, ún. diszjunkciót használó változat („;”  $\equiv$  „vagy”):

```
alap4(X, Y, Kif) :- (Kif = X+Y ; Kif = X-Y
 ; Kif = X*Y ; Kif = X/Y
).
```

A=B infix alakban írható beépített eljárás, jelentése:

A és B azonos alakra hozható, esetleges változóbehelyettesítésekkel.

## Aritmetikai kifejezés levéllistájának előállítás

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

```
% kif_levelek(Kif, L): A számokból alapműveletekkel felépülő Kif
%
 kifejezés leveleiben levő számok listája L.
kif_levelek(Kif, L) :-
 number(Kif), L = [Kif]. % L egyelemű, Kif-ből álló lista
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, fejkommentje 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]
```

## Az append eljárás többirányú használata

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

*% append(L1, L2, 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] 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 előállítás

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

```

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

```

```

| ?- 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) ? ; ...

```

## 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 csak akkor 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.
```

```
levelek_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
```

## Adott értékű kifejezés előállítás – 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

```

# Tartalom

## 1 Bevezetés

- Követelmények, tudnivalók
- Egy kedvcsináló példa
- A példa Prolog változata
- 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:
 

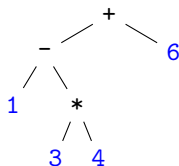
```
4> [X || X <- [1,2,3]]. % {x|x ∈ {1,2,3}}
[1,2,3]
5> [X || X <- [1,2,3], X*X > 5]. % {x|x ∈ {1,2,3}, x2 > 5}
[3]
6> [{X,Y} || X <- [1,2,3], Y <- lists:seq(1,X)].
% {(x,y)|x ∈ {1,2,3}, y ∈ {1..x}}
[{1,1},{2,1},{2,2},{3,1},{3,2},{3,3}]
```



## Aritmetikai kifejezések ábrázolása

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

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



- Az 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 hármassokkal ábrázoljuk:

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

## Adott méretű fák felsorolása

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

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

### 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 tartalmazóké:
 
$$\left\{ (b, '+', j) \mid \begin{array}{l} i \in [1 .. n - 1], \\ b \in \text{fak}(i), \\ j \in \text{fak}(n - i) \end{array} \right\}$$

## 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() = '+' | '-' | '*' | '/'.
% kifek(L) = L levéllistájú kifejezések 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 kifejezések:  $\{h\}$
- $L$  levéllistájú kifejezések:
 
$$\{(b, m, j) \mid \begin{array}{l} L_B \oplus L_J = L, \\ b \in \text{kifek}(L_B), \\ j \in \text{kifek}(L_J), \\ m \in \{+, -, *, /\} \end{array}\}$$

## 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]]
```

## Adott értékű kifejezések felsorolása – teljes kód

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

```
-module(kif).
-export([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])].
```

## II. rész

# Cékla: deklaratív programozás C++-ban

- 1 Bevezetés
- 2 Cékla: deklaratív programozás C++-ban
- 3 Prolog alapok
- 4 Erlang alapok
- 5 Haladó Prolog
- 6 Haladó Erlang

# Tartalom

2

## Cékla: deklaratív programozás C++-ban

- Imperatív és deklaratív programozás C nyelven
- Jobbrekurzió
- A Cékla programozási nyelv
- Listakezelés Céklában
- Magasabbrendű függvények (kiegészítő anyag)

# Imperatív és deklaratív programozási szemlélet

- Imperatív program

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

```
int pow(int A, int N) { // pow(A,N) = AN
 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!
 }
 return P; // Add vissza P végértékét
```

- 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*AN-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 tárígénnnyel :- (

```
powd(10,3) : 10*powd(10,2) : 10*(10*powd(10,1)) :
 10 * (10 * (10 * 1))
 └──────────┘
veremben tárolva
```

# Tartalom

## 2 Cékla: deklaratív programozás C++-ban

- Imperatív és deklaratív programozás C nyelven
- **Jobbrekurzió**
- A Cékla programozási nyelv
- Listakezelés Céklában
- Magasabbrendű függvények (kiegészítő anyag)

## Hatékony deklaratív programozás

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

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

|                                                                                                                             |   |                                                                                                                                       |
|-----------------------------------------------------------------------------------------------------------------------------|---|---------------------------------------------------------------------------------------------------------------------------------------|
| <pre>int ispow(int A, int B) {     if (A == 1) return true;     if (A%B==0) return ispow(A/B, B);     return false; }</pre> | } | <pre>int ispow(int A, int B) {     again:     if (A == 1) return true;     if (A%B==0) {A=A/B; goto again;}     return false; }</pre> |
|-----------------------------------------------------------------------------------------------------------------------------|---|---------------------------------------------------------------------------------------------------------------------------------------|

- 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 **farokrekurzió**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:  $\text{powi}(A, N, P) = P * A^N$*

```
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);
}
```

# Tartalom

## 2 Cékla: deklaratív programozás C++-ban

- Imperatív és deklaratív programozás C nyelven
- Jobbrekurzió
- **A Cékla programozási nyelv**
- Listakezelés Céklában
- Magasabbrendű függvények (kiegészítő anyag)

## 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
    - `<aritmetikai-op>`: `+` | `-` | `*` | `/` | `%` |
    - `<hasonlító-op>`: `<` | `>` | `==` | `!=` | `>=` | `<=`
- C++ fordítóval is fordítható a `cekla.h` fájl birtokában: láncolt lista kezelése, függvénytípusok és kiírás
- Kiíró függvények: főleg nyomkövetéshez, ugyanis *mellékhátásuk* van!
  - `write(X);`      Az `x` kifejezés kiírása a standard kimenetre
  - `writeln(X);`    Az `x` kifejezés kiírása és soremelés
- A (Prologban írt) Cékla fordító és a `cekla.h` letölthető a tárgy honlapjáról

## Cékla Hello world!

hello.cpp

```
#include "cekla.h" // így C++ fordítóval is fordítható
int main() { // bárhogy 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 Cekla 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

- Imperatív és deklaratív programozás C nyelven
- Jobbrekurzió
- A Cékla programozási nyelv
- **Listakezelés Céklában**
- Magasabbrendű függvények (kiegészítő anyag)

## 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 Cékla 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!)
 }
}

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

## Sztringek Céklaiban

- 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(nil == ""); // 1, true int-értéke
 writeln(nil); // []
 writeln(L5); // abc
 writeln(cons(10, L5)); // [10,97,98,99]
 writeln(tl(L4)); // bc
}
```

## Listák összefűzése: append

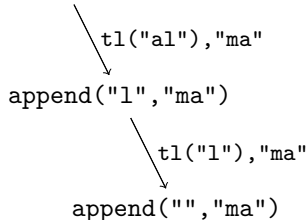
- `append(L1, L2)` visszaadja `L1` és `L2` elemeit egymás után fűzve

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

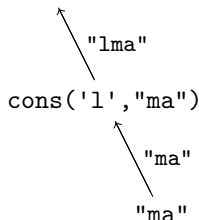
```
list append(const list L1, const list L2) {
 if (L1 == nil) return L2;
 return cons(hd(L1), append(tl(L1), L2)); }
```

- Például `append("al", "ma") == "alma"` (vagyis `[97,108,109,97]`).

`append("al", "ma")`



`cons('a', "lma")`



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

## Lista megfordítása: nrev, reverse

- Naív (négyzetes lépésszámú) 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 lépésszámú 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)



# Tartalom

2

## Cékla: deklaratív programozás C++-ban

- Imperatív és deklaratív programozás C nyelven
- Jobbrekurzió
- A Cékla programozási nyelv
- Listakezelés Céklában
- **Magasabbrendű függvények (kiegészítő anyag)**

## Magasabbrendű függvények Céklaiban (kiegészítő anyag)

- Magasabbrendű 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 számjegyek karakterek 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); }
```

## Magasabbrendű függvények: `map`, `filter` (kiegészítő anyag)

- `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 `map(incr, L)` kifejezés értéke  $[11,21,31]$ .

- `filter(P,L)`: az  $L$  lista azon  $X$  elemei, amelyekre  $P(X)$  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 `filter(digit, "X=100;")` kifejezés értéke "100" (azaz  $[49,48,48]$ )

## Magasabbrendű függvények: foldl (kiegészítő anyag)

- Hajtogatás balról

```
// 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));
}
```

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

## Magasabbrendű függvények: `foldr` (kiegészítő anyag)

- Hajtogatás jobbról

```
// 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 példák,  $L = [1, 5, 3, 8]$

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

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

## III. rész

# Prolog alapok

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

## Deklaratív programozási nyelvek

- A matematika függvényfogalmán alapuló **funkcionális prog.** nyelvek: LISP, SML, Haskell, Erlang, ...
- A matematika relációfogalmán alapuló **logikai programozási (LP)** nyelvek: Prolog, SQL, Mercury, Korlátnyelvek (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!

# A kurzus Logikai Programozás (LP) része

- **1. blokk:** A Prolog LP 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



# Tartalom

3

## Prolog alapok

- Prolog bevezetés – néhány példa
- A Prolog nyelv alapszintaxisa
- Nyomkövetés: 4-kapus doboz modell
- Listakezelő eljárások Prologban
- További vezérlési szerkezetek
- Operátorok
- Prolog végrehajtás – összefoglalás, pontosítás
- Ízelítő a „Haladó Prolog” részből

## A Prolog alapelemei: a családi kapcsolatok példája

- Adatok
  - Adottak személyekre vonatkozó állítások, pl.

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

| férfi           |
|-----------------|
| Imre            |
| István          |
| Géza            |
| Civakodó Henrik |
| ...             |

- 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)
```

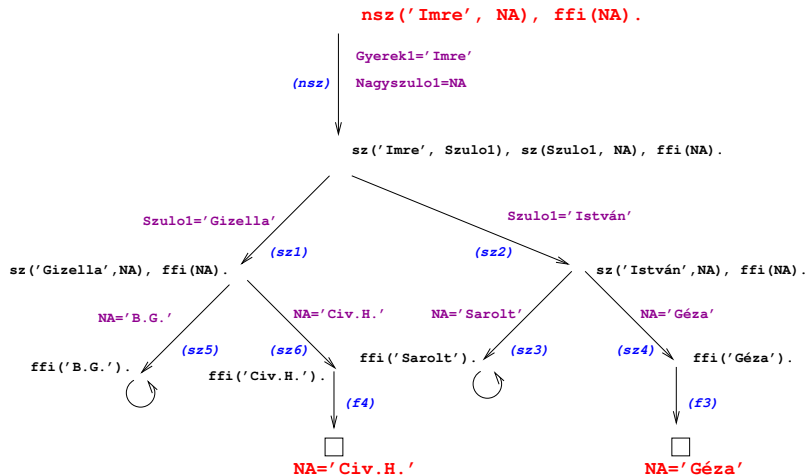
```
% Ki Imre nagyapja?
| ?- nagyszuloje('Imre', NA),
 ffi(NA).
NA = 'Civakodó Henrik' ? ;
NA = 'Géza' ? ;
no
% Ki Géza unokája?
| ?- nagyszuloje(U, 'Géza').
U = 'Imre' ? ;
no
% Ki Imre nagyszülője?
| ?- nagyszuloje('Imre', NSz).
NSz = 'Burgundi Gizella' ? ;
NSz = 'Civakodó Henrik' ? ;
NSz = 'Sarolt'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

## Deklaratív szemantika – klózok 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 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 meghiúsul.
  - 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 (esetleg változó-behelyettesítésekkel), vagy lehet sikertelen.
  - Siker esetén a behelyettesítéseket elvégezzük a célsorozat maradékán, ez lesz az új célsorozat.
  - Sikertelenség esetén a redukciós lépés is sikertelenül végződik (meghiúsul).

## 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 klózatát, amelynek feje egyesíthető a hívással, és végrehajtjuk a redukciót.
3. Ha a redukció sikeres (találunk egyesíthető fejű klózt), folytatjuk a végrehajtást 1.-től az új célsorozattal.
4. Ha a redukció meghiúsul, akkor visszalépés következik:
  - visszatérünk a legutolsó, felhasználói eljárással történt (sikeres) redukciós lépéshez,
  - annak *bemeneti* célsorozatát megpróbáljuk *újabb* klózzal redukálni – ugrás a 2. lépésre  
(Ennek meghiúsulása értelemszerűen újabb visszalépést okoz.)

### A végrehajtás nem „intelligens”

- Pl. | ?- *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 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:  $\langle \text{struktúranév} \rangle \langle \langle \text{arg}_1 \rangle, \dots, \langle \text{arg}_n \rangle \rangle$ 
    - 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:  

$$X \text{ is } Y+1 \equiv \text{is}(X, +(Y,1))$$
- 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 – **dinamikus típusfogalom**
  - ha visszalépünk egy redukciós lépésen keresztül, akkor az ott behelyettesített változók behelyettesítése megszűnik



## Aritmetikai beépített eljárások

### Aritmetikai beépített predikátumok (eljárások)

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

### A faktoriális függvény definíciója Prologban

- funkció 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.
```

## Néhány beépített predikátum

- Kifejezések egyesítése
  - $X = Y$ : az  $X$  és  $Y$  **szimbolikus** kifejezések egyesítése  $\equiv$  azonos alakra hozása változók esetleges behelyettesítésével
  - $X \neq Y$ : az  $X$  és  $Y$  kifejezések **nem** egyesíthetőek (nem hozhatók azonos alakra)
- Típusvizsgálatot végző beépített predikátumok
  - `var(X)`:  $X$  változó
  - `nonvar(X)`:  $X$  nem változó
    - `atomic(X)`:  $X$  konstans;  
`atom(X)`:  $X$  névkonstans, `number(X)`:  $X$  szám  
`integer(X)`:  $X$  egész szám, `float(X)`:  $X$  lebegőpontos szám
    - `compound(X)`:  $X$  összetett kifejezés
- További hasznos predikátumok
  - `true`, `fail`: Mindig sikerül ill. mindig megghiúsul.
  - `write(X)`: Az  $X$  Prolog kifejezést kiírja.
  - `write_canonical(X)`:  $X$  kanonikus (alapstruktúra) alakját írja ki.
  - `nl`: Kiír egy újsort.

## Programfejlesztési beépített predikátumok

- `consult(File)`: A `File` állományban levő programot beolvassa és értelmezendő alakban eltárolja. (`File = user`  $\Rightarrow$  terminálról olvas.)
- `trace`, `notrace`: A (teljes) nyomkövetést be- ill. kikapcsolja.
- `listing` vagy `listing(Predikátum)`: Az értelmezendő alakban eltárolt összes ill. adott nevű predikátumokat kilistázza.
- `halt`: A Prolog rendszer befejezi működését.

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

## Adatstruktúrák Prologban – a bináris fák példája

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

### Binárisfa-struktúra C-ben

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

A Prolog dinamikusan típusos nyelv –  
nincs szükség explicit típusdefinícióra

- Mercury típusleírás (komment)

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

- A típushoz tartozás ellenőrzése

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

## Bináris fák összegzése

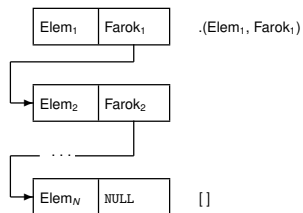
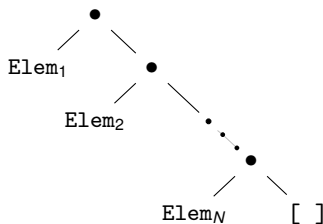
- Egy bináris fa levélösszegének kiszámítása:
  - levél esetén a levélben tárolt egész
  - csomópont esetén a két részfa levélösszegének összege

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

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

## 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:  
 $[Elem_1 | \dots | [Elem_N | Farok] \dots] \implies [Elem_1, \dots, Elem_N | Farok]$
- Ha a farok [], a „| []” jelsorozat elhagyható:  
 $[Elem_1, \dots, Elem_N | []] \implies [Elem_1, \dots, Elem_N]$

| ?- [1,2] = [X|Y].  $\implies$  X = 1, Y = [2] ?

| ?- [1,2] = [X,Y].  $\implies$  X = 1, Y = 2 ?

| ?- [1,2,3] = [X|Y].  $\implies$  X = 1, Y = [2,3] ?

| ?- [1,2,3] = [X,Y].  $\implies$  no

| ?- [1,2,3,4] = [X,Y|Z].  $\implies$  X = 1, Y = 2, Z = [3,4] ?

| ?- L = [1|\_], L = [\_|2|\_].  $\implies$  L = [1,2|\_A] ? % nyílt végű

| ?- L = .(1,[2,3|[]]).  $\implies$  L = [1,2,3] ?

| ?- L = [1,2|. (3,[])].  $\implies$  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 skalár (szám) szorzatának, és két vektor skalárszorzatának kiszámítására. Feltételezhető, hogy egy hívásban a vektorok azonos hosszúságúak.

```
% v_ossz(+A, +B, ?C): C az A és B vektorok összege
```

```
v_ossz([], [], []).
```

```
v_ossz([A|AL], [B|BL], [C|CL]) :-
```

```
 C is A+B,
```

```
 v_ossz(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

3

## Prolog alapok

- Prolog bevezetés – néhány példa
- **A Prolog nyelv alapszintaxisa**
- Nyomkövetés: 4-kapus doboz modell
- Listakezelő eljárások Prologban
- További vezérlési szerkezetek
- Operátorok
- Prolog végrehajtás – összefoglalás, pontosítás
- Ízelítő a „Haladó Prolog” részből

## 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
% -----
% | | |
% struktúranév \ argumentum, változó
% \- argumentum, összetett kif.
```

- Szintaxis:

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

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

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

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

## Lexikai elemek: példák és szintaxis

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

```
<változó> ::= <nagybetű><alfanum. jel>...|
 _ <alfanum. jel>...
<névkonstans> ::= ' <idézett kar.>... ' |
 <kisbetű><alfanum. jel>...|
 <tapadó jel>...| ! | ; | [] | {}
<egész szám> ::= {előjeles vagy előjeltelen számjegysorozat}
<lebegőp.szám> ::= {belsejében tizedespontot tartalmazó
 számjegysorozat esetleges exponenssel}
<idézett kar.> ::= {tetszőleges nem ' és nem \ karakter} |
 \ <escape szekvencia>
<alfanum. jel> ::= <kisbetű> | <nagybetű> | <számjegy> | _
<tapadó jel> ::= + | - | * | / | \ | $ | ^ | < | > | = | ` | ~ | : | . | ? | @ | # | &
```

## Prolog programok formázása

- Megjegyzések (comment)
  - A % százalékjeltől a sor végéig
  - A /\* jelpártól a legközelebbi \*/ jelpárig.
- Formázó elemek (komment, szóköz, újsor, tabulátor stb.) szabadon használhatók
  - 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 klózek legyenek egymás mellett a programban, közéjük ne tegyünk üres sort.
  - A predikátum elé tegyünk egy üres sort és egy fejkommentet:  
`% predikátumnév(A1, ..., An): A1, ..., An közötti`  
`% összefüggést leíró kijelentő mondat.`
  - A klózfejet írjuk 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-klózok** nyelve,
    - végrehajtási mechanizmusa: **mintaillesztéses** eljáráshíváson alapuló **visszalépéses** keresés.

# Tartalom

3

## Prolog alapok

- Prolog bevezetés – néhány példa
- A Prolog nyelv alapszintaxisa
- **Nyomkövetés: 4-kapus doboz modell**
- Listakezelő eljárások Prologban
- További vezérlési szerkezetek
- Operátorok
- Prolog végrehajtás – összefoglalás, pontosítás
- Ízelítő a „Haladó Prolog” részből

## A Prolog nyomkövető által használt eljárás-doboz modell

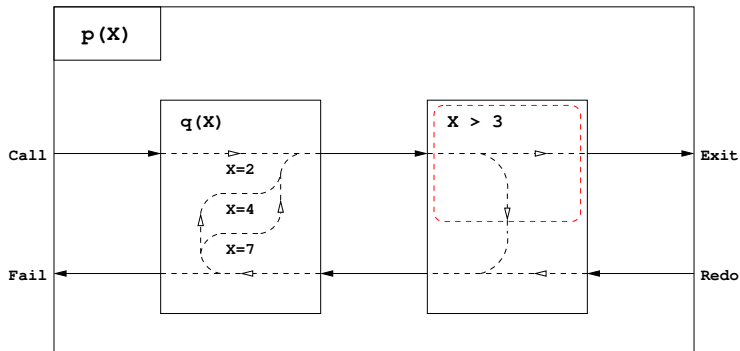
- A Prolog eljárás-végrehajtás két fázisa
  - előre menő: egymásba **skatulyázott eljárás-be** és **-kilépések**
  - visszafelé menő: **új megoldás** kérése egy már lefutott eljárástól
- Egy egyszerű példaprogram, hívása  $|- ?- p(X)$ .  
 $q(2)$ .  $q(4)$ .  $q(7)$ .  $p(X) :- q(X), X > 3$ .
- Példafutás: belépünk a  $p/1$  eljárásba (Hívási kapu, Call port)
  - Belépünk a  $q/1$  eljárásba (Call port)
  - $q/1$  sikeresen lefut,  $q(2)$  eredménnyel (Kilépési kapu, Exit port)
    - $A > /2$  eljárásba belépünk a  $2>3$  hívással (Call)
    - $A > /2$  eljárás sikertelenül fut le (Meghiúsulási kapu, Failport)
  - (visszafelé menő futás): visszatérünk (a már lefutott)  $q/1$ -be, újabb megoldást kérve (Újra kapu, Redo Port)
  - A  $q/1$  eljárás újra sikeresen lefut a  $q(4)$  eredménnyel (Exit)
    - A  $4>3$  hívással a  $> /2$ -be belépünk majd kilépünk (Call, Exit)
- A  $p/1$  eljárás sikeresen lefut  $p(4)$  eredménnyel (Exit)



# Eljárás-doboz modell – grafikus szemléltetés

$q(2).$   $q(4).$   $q(7).$

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



## Eljárás-doboz modell – egyszerű nyomkövetési példa

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

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

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

`| ?- trace, p(X).`

```

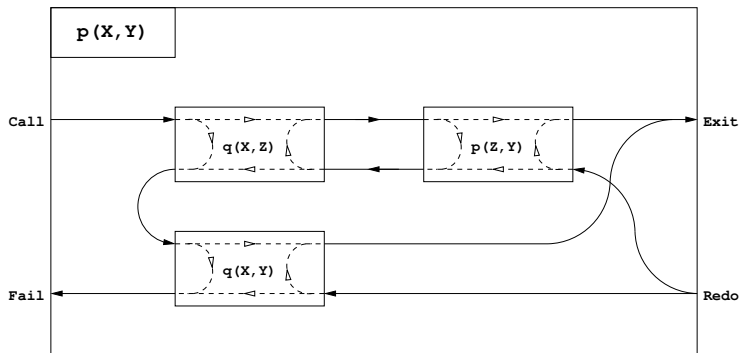
1 1 Call: p(_463) ?
2 2 Call: q(_463) ?
? 2 2 Exit: q(2) ? % ? ≡ maradt választási pont q-ban
3 2 Call: 2>3 ?
3 2 Fail: 2>3 ?
2 2 Redo: q(2) ?
? 2 2 Exit: q(4) ?
4 2 Call: 4>3 ?
4 2 2 Exit: 4>3 ? % nincs ? ⇒ a doboz törlődik (*)
? 1 1 Exit: p(4) ?
X = 4 ? ;
1 1 1 Redo: p(4) ?
 % (*) miatt nem látjuk a Redo-Fail kapukat a 4>3 hívásra
2 2 Redo: q(4) ?
2 2 Exit: q(7) ?
5 2 Call: 7>3 ?
5 2 Exit: 7>3 ?
1 1 Exit: p(7) ? % nincs ? ⇒ a doboz törlődik (*)
X = 7 ? ; no

```

## 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: érjük 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` parancssal indítja a nyomkövetést
  - `notrace`, `nodebug`, `nozip` – kikapcsolja a nyomkövetést
  - `spy(P)`, `nospyp(P)`, `nospya11` – töréspont be/ki a `P` eljárásra,  $\forall$  ki.
- Alapvető nyomkövetési parancsok, ujsorral (<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`) – újratekint 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 részjeljárás Hívás kapujához érkezünk, **létrehozunk** egy példányt a meghívandó eljárásból, majd
- meghívjuk az eljáráspéldány „következő megoldás” metódusát (\*)
  - Ha ez sikerül, akkor a vezérlés átkerül a következő hívás Hívás kapujára, vagy a szülő Kilépési kapujára
  - Ha ez meghiúsul, **megszüntetjük** az eljáráspéldányt majd ugrunk az előző hívás Újra kapujára, vagy a következő klóz elejére, stb.
- Amikor egy Újra kapuhoz érkezünk, a (\*) lépésnél folytatjuk.
- A szülő Újra kapuja (a „következő megoldás” nem első hívása) a tárolt klózsorszámnak megfelelő klózban az utolsó Újra kapura adja a vezérlést.

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

A p/2 eljárás (91. 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).
 qbptr = new q(x, py); // create a new instance of subgoal q(X,Z)
 case 2: // (enter here for Redo port if clno==2)
 /* redo21: */
 if(!qbptr->next()) { // if q(X,Y) fails
 delete qbptr; // destroy it,
 return FALSE; // and exit via the Fail port
 }
 return TRUE; // otherwise, exit via the Exit port
 } }

```

# Tartalom

3

## Prolog alapok

- Prolog bevezetés – néhány példa
- A Prolog nyelv alapszintaxisa
- Nyomkövetés: 4-kapus doboz modell
- **Listakezelő eljárások Prologban**
- További vezérlési szerkezetek
- Operátorok
- Prolog végrehajtás – összefoglalás, pontosítás
- Ízelítő a „Haladó Prolog” részből



## 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 kiküszöbölhetőek, 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.

## Lista építése *előlről* – nyílt végű listákkal

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

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

- 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!

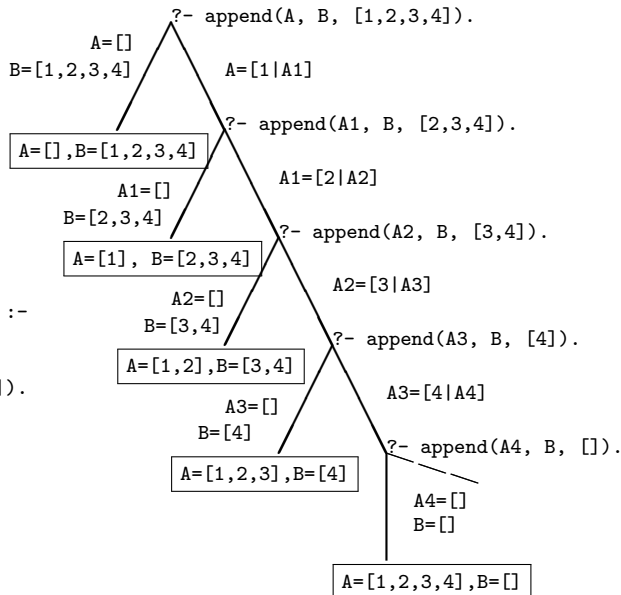
- Célok (pl.): `append([1,2,3], [4], Ered), write(Ered).`
- Fej: `append([X|L1], L2, [X|L3])`
- Behelyettesítés:  $X = 1, L1 = [2,3], L2 = [4], \text{Ered} = [1|L3]$
- Új célsorozat: `append([2,3], [4], L3), write([1|L3]).`  
( $Ered$  nyílt végű lista, farka még behelyettesítetlen.)
- A további redukciós lépések behelyettesítése és eredménye:
 

|                 |                                                          |
|-----------------|----------------------------------------------------------|
| $L3 = [2 L3a]$  | <code>append([3], [4], L3a), write([1 [2 L3a]]).</code>  |
| $L3a = [3 L3b]$ | <code>append([], [4], L3b), write([1,2 [3 L3b]]).</code> |
| $L3b = [4]$     | <code>write([1,2,3 [4]]).</code>                         |

# Listák szétbontása az append/3 segítségével

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

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



## Nyílt végű listák az append változatokban

$$\begin{array}{l} \text{app0}([], L, L). \\ \text{app0}([X|L1], L2, R) :- \\ \quad \text{app0}(L1, L2, L3), R = [X|L3]. \end{array} \quad \left\| \right. \quad \begin{array}{l} \text{append}([], L, L). \\ \text{append}([X|L1], L2, [X|L3]) :- \\ \quad \text{append}(L1, L2, L3). \end{array}$$

- Ha az 1. argumentum zárt végű ( $n$  hosszú), mindkét változat legfeljebb  $n + 1$  lépésben egyértelmű választ ad, amely lehet nyílt végű:
  - | ?-  $\text{app0}([1,2], L2, L3)$ .  $\implies L3 = [1,2|L2]$  ? ; 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  $\infty$  sok Prolog kifejezéssel fedhető le, pl.
  - $\_ \oplus [1] = L$  ( $\equiv L$  utolsó eleme 1):  $L = [1]; \_ , [1]; \_ , \_ , [1]; \dots$
- `app0` szétszedésre nem jó, pl.  $\text{app0}(L, [1,2], []) \implies \infty$  ciklus, mert redukálva a 2. klózzal  $\implies \text{app0}(L1, [1,2], L3), [X|L3] = []$ .
- Az `append` eljárás jobbrekurzív, hála a logikai változó használatának

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

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

## Listák megfordítása

- Naív (négyzetes lépésszámú) megoldás

```

% nrev(L, R): R = L megfordítása.
nrev([], []).
nrev([X|L], R) :-
 nrev(L, RL),
 append(RL, [X], R).

% nrev(L) = L megfordítása (Cékla)
list nrev(const list XL)
{
 if (XL == nil) return nil;
 int X = hd(XL); list L = tl(XL);
 list RL = nrev(L);
 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]). => yes DE
| ?- member(2, [1,2,3,2]), R=yes. => R=yes ? ; R=yes ? ; no
```

- Lista elemeinek felsorolása:

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

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

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

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

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

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



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

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

- A `lists` könyvtárban a fenti módon definiált `select/3` eljárás keresési tere **véges**, ha vagy a 2., vagy a 3. argumentuma zárt végű lista.

## Listák permutációja (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).
```

```
⇒ L = [1,2] ? ; L = [2,1] ? ; no
```

```
| ?- perm([a,b,c], L).
```

```
⇒ L = [a,b,c] ? ; L = [a,c,b] ? ; L = [b,a,c] ? ;
 L = [b,c,a] ? ; L = [c,a,b] ? ; L = [c,b,a] ? ; no
```

```
| ?- perm(L, [1,2]).
```

```
⇒ 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 fejkommentben:  
% `perm(+Lista, ?Perm)`: Lista permutációja a Perm lista.
- A `lists` könyvtár tartalmaz egy kétirányban is működő `permutation/2` eljárást.

# Tartalom

3

## Prolog alapok

- Prolog bevezetés – néhány példa
- A Prolog nyelv alapszintaxisa
- Nyomkövetés: 4-kapus doboz modell
- Listakezelő eljárások Prologban
- **További vezérlési szerkezetek**
- Operátorok
- Prolog végrehajtás – összefoglalás, pontosítás
- Ízelítő a „Haladó Prolog” részből

# Diszjunkció

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

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

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

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

## A diszjunkció mint szintaktikus édesítőszer

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

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

- Kigyűjtjük azokat a változókat, amelyek a diszjunkcióban és azon kívül is előfordulnak
- 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  $X = \text{István}$  vagy  $X = \text{Gizella}$ .

- Az (1) predikátum átalakítható egyetlen, diszjunkciós klózzá:

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

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

- Általánosan: tetszőleges predikátum egyklózossá alakítható:

- a klózeket azonos fejlűvé alakítjuk, új változók és =-ek bevezetésével:

```
szuloje('Imre', Sz) :- Sz = 'István'.
```

```
szuloje('Imre', Sz) :- Sz = 'Gizella'.
```

- a klóztörzseket egy diszjunkcióvá fogjuk össze, lásd (2).

## A meghíúsulásos negáció (NF – Negation by Failure)

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

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

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

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

## Gondok a megghiúsulások negációjával

- 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). \implies no
| ?- \+ szuloje('Géza', X). \implies 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  $\implies (*)$  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) :-
 number(K1),
 egyhat(K2, E0),
 E is K1*E0.
egyhat(K1*K2, E) :-
 number(K2),
 egyhat(K1, E0),
 E is K2*E0.
% (4)
% (5)
```

- A fenti megoldás hibás – többszörös megoldást kaphatunk:

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

- A többszörös megoldás oka:

az egyhat(2\*3, E) hívás esetén a (4) és (5) klóz egyaránt sikeres!

## Többszörös megoldások kiküszöbölése

- El kell 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 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 zárójelk feleslegesek (‘;’ jobbról-balra zárójeleződik).

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

## Feltételes kifejezés – példák

- Faktoriális

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

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

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

## Példa: intervallum 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.

## A vágó eljárás – a feltételes szerkezet megvalósítási alapja

- 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, !.
```
- Segédfogalom: egy cél **szülőjének** az őt tartalmazó klóz fejével illetett hívást nevezzük
  - A 4-kapus modellben a szülő a körülvevő dobozhoz rendelt cél.
  - A fenti vágók szülője lehet pl. a `első_poz_elem([-1,0,3,0,2], P)` cél
- A vágó végrehajtása (a fentivel ekvivalens definíció):
  - mindig sikerül; de mellékhatásként a végrehajtás adott állapotától visszafelé egészen a szülő célíg – azt is beleértve – megszünteti a választási pontokat.
- A vágó szemléltetése a 4-kapus doboz modellben: a vágó `Fail` kapujából a körülvevő (szülő) doboz `Fail` kapujára megyünk.

# Tartalom

3

## Prolog alapok

- Prolog bevezetés – néhány példa
- A Prolog nyelv alapszintaxisa
- Nyomkövetés: 4-kapus doboz modell
- Listakezelő eljárások Prologban
- További vezérlési szerkezetek
- **Operátorok**
- Prolog végrehajtás – összefoglalás, pontosítás
- Ízelítő a „Haladó Prolog” részből



# Operátor-kifejezések

- Példa:  $S \text{ is } -S1+S2$  ekvivalens az  $\text{is}(S, +(-(S1), S2))$  kifejezéssel

- Operátoros kifejezések

$\langle \text{összetett kif.} \rangle ::=$

|                                        |                                                |                                     |                      |
|----------------------------------------|------------------------------------------------|-------------------------------------|----------------------|
| $\langle \text{struktúranév} \rangle$  | $( \langle \text{argumentum} \rangle, \dots )$ | {eddig csak ez volt}                |                      |
| $\langle \text{argumentum} \rangle$    | $\langle \text{operátornév} \rangle$           | $\langle \text{argumentum} \rangle$ | {infix kifejezés}    |
| $\langle \text{operátornév} \rangle$   | $\langle \text{argumentum} \rangle$            |                                     | {prefix kifejezés}   |
| $\langle \text{argumentum} \rangle$    | $\langle \text{operátornév} \rangle$           |                                     | {posztfix kifejezés} |
| $( \langle \text{kifejezés} \rangle )$ |                                                |                                     | {zárójeles kif.}     |

$\langle \text{operátornév} \rangle ::= \langle \text{struktúranév} \rangle$  {ha operátorként lett definiálva}

- Operátor(ok) definiálása

$\text{op}(\text{Prio}, \text{Fajta}, \text{OpNév})$  vagy  $\text{op}(\text{Prio}, \text{Fajta}, [\text{OpNév}_1, \dots, \text{OpNév}_n])$ , ahol

- Prio (prioritás): 1–1200 közötti egész
- Fajta: az  $\text{yfx}$ ,  $\text{xfy}$ ,  $\text{xfx}$ ,  $\text{fy}$ ,  $\text{fx}$ ,  $\text{yf}$ ,  $\text{xf}$  névkonstansok egyike
- $\text{OpNév}_i$  (az operátor neve): tetszőleges névkonstans
- Az  $\text{op}/3$  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 végrehajthatódnak.

## 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       | <b>infix</b>    | $A \ f \ B \equiv f(A, B)$ |
|           | fy         | fx        | <b>prefix</b>   | $f \ A \equiv f(A)$        |
| yf        |            | xf        | <b>posztfix</b> | $A \ f \equiv f(A)$        |

- A zárójelést 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 zárójel (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 zárójel)
  - $a=b=c$  szintaktikusan hibás, mert az = operátor fajtája xfx, azaz **nem-asszociatív**

# Szabványos, beépített operátorok

## Szabványos operátorok

Színkód: már ismert, új aritmetikai

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

## További beépített operátorok SICStus Prologban

|      |     |                             |  |
|------|-----|-----------------------------|--|
| 1150 | fx  | <code>mode public</code>    |  |
|      |     | <code>dynamic block</code>  |  |
|      |     | <code>volatile</code>       |  |
|      |     | <code>discontiguous</code>  |  |
|      |     | <code>initialization</code> |  |
|      |     | <code>multifile</code>      |  |
|      |     | <code>meta_predicate</code> |  |
| 1100 | xfy | <code>do</code>             |  |
| 900  | fy  | <code>spy nospy</code>      |  |
| 550  | xfy | <code>:</code>              |  |
| 500  | yfx | <code>\</code>              |  |
| 200  | fy  | <code>+</code>              |  |

## Operátorok implicit zárójelezése – általános szabályok

- Egy  $X \text{ op}_1 Y \text{ op}_2 Z$  zárójelezése, ahol  $\text{op}_1$  és  $\text{op}_2$  prioritása  $n_1$  és  $n_2$ :
  - ha  $n_1 > n_2$  akkor  $X \text{ op}_1 (Y \text{ op}_2 Z)$ ;
  - ha  $n_1 < n_2$  akkor  $(X \text{ op}_1 Y) \text{ op}_2 Z$ ; (kisebb prio.  $\Rightarrow$  erősebb kötés)
  - ha  $n_1 = n_2$  és  $\text{op}_1$  jobb-asszociatív ( $xfy$ ), akkor  $X \text{ op}_1 (Y \text{ op}_2 Z)$ ;
  - **egyébként**, ha  $n_1 = n_2$  és  $\text{op}_2$  bal-assz. ( $yfx$ ), akkor  $(X \text{ op}_1 Y) \text{ op}_2 Z$ ;
  - 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 zárójelez~~etlen~~ operandusaként
  - legfeljebb  $n - 1$  prioritású operátort fogadunk el az  $x$  oldalon
  - legfeljebb  $n$  prioritású operátort fogadunk el az  $y$  oldalon
- A zárójelezett kifejezéseket és az alapstruktúra-alakú kifejezéseket feltétel nélkül elfogadjuk operandusként
- Az alapszabály a prefix és posztfix operátorokra is alkalmazandó

## Operátorok – 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 `, ,` **operátorként** 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)).`  $\implies$  `' , '(a , '(b,c))`
      - | `?- write_canonical(a,b,c).`  $\implies$  `! write_canonical/3 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  $fy$  fajtával 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(_, xfy, Op), write_canonical(Op), write(' '), fail.
```

```
; do -> ', ' : ^
```

```
no
```

# Operátorok felhasználása

- Mire jók az operátorok?
  - aritmetikai eljárások kényelmes irására, pl.  $X \text{ is } (Y+3) \bmod 4$
  - 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.

## 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
```



## Klasszikus szimbolikuskifejezés-feldolgozás: deriválás

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

```
% deriv(Kif, D): Kif-nek az x szerinti deriváltja D.
```

```
deriv(x, 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

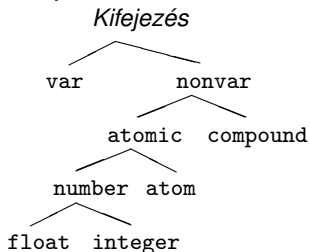
3

## Prolog alapok

- Prolog bevezetés – néhány példa
- A Prolog nyelv alapszintaxisa
- Nyomkövetés: 4-kapus doboz modell
- Listakezelő eljárások Prologban
- További vezérlési szerkezetek
- Operátorok
- **Prolog végrehajtás – összefoglalás, pontosítás**
- Ízelítő a „Haladó Prolog” részből

# Az egyesítési algoritmus – a Prolog adatfogalma

- Prolog kifejezések osztályozása – kanonikus alak



|                          |                     |
|--------------------------|---------------------|
| <code>var(X)</code>      | X változó           |
| <code>nonvar(X)</code>   | X nem változó       |
| <code>atomic(X)</code>   | X konstans          |
| <code>compound(X)</code> | X struktúra         |
| <code>number(X)</code>   | X szám              |
| <code>atom(X)</code>     | X névkonstans       |
| <code>float(X)</code>    | X lebegőpontos szám |
| <code>integer(X)</code>  | X egész szám        |

## 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 feje és egy célsorozat első tagja)
  - feladata: a két kifejezés egyesíthetőségének eldöntése
  - matematikailag az eredménye: megghiúsulás, vagy siker és a legáltalánosabb egyesítő – *most general unifier*,  $mgu(A, B)$  – előállítása
  - praktikusán 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:  $V \leftarrow 1, T \leftarrow leaf(1), S \leftarrow 1$
    - legáltalánosabb egyesítő behelyettesítés:  $T \leftarrow leaf(V), S \leftarrow V,$   
vagy  $T \leftarrow leaf(S), V \leftarrow S$
  - 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 meghiúsulással;
  - $A_2$  és  $B_2$  egyesítését elvégzi, ha ez sikertelen, akkor kilép meghiúsulással;
  - ...
  - $A_N$  és  $B_N$  egyesítését elvégzi, ha ez sikertelen, akkor kilép meghiúsulássalKilép sikerrel
- 5 Minden más esetben kilép meghiúsulással ( $A$  és  $B$  nem egyesíthető)

## Egyesítési példák a gyakorlatban

- Az egyesítéssel kapcsolatos beépített eljárások:
  - $X = Y$  egyesíti a két argumentumát, megghiúsul, ha ez nem lehetséges.
  - $X \neq Y$  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 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 ún. ciklikus kifejezések keletkezhetnek)
  - Szabványos eljárásként rendelkezésre áll:  
`unify_with_occurs_check/2`
  - Kiterjesztés (pl. SICStus): az előfordulás-ellenőrzés elhagyása miatt keletkező ciklikus kifejezések tisztességes kezelése.

- Példák:

```
| ?- X = s(1,X).
```

```
 X = s(1,s(1,s(1,s(1,s(...))))).
```

```
| ?- unify_with_occurs_check(X, s(1,X)).
```

```
 no
```

```
| ?- X = s(X), Y = s(s(Y)), X = Y.
```

```
 X = s(s(s(s(s(...))))), Y = s(s(s(s(s(...))))).
```



## Az egyesítési alg. matematikai megfogalmazása (kieg. anyag)

- A *behelyettesítés* egy olyan  $\sigma$  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:
  - $K\sigma$ :  $\sigma$  alkalmazása egy *tetszőleges*  $K$  kifejezésre:  $\sigma$  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  $\theta$  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 rendel (  $Dom(\sigma \otimes \theta) = Dom(\sigma) \cup Dom(\theta)$  ):

$$\sigma \otimes \theta = \{x \leftarrow (x\sigma)\theta \mid x \in Dom(\sigma)\} \cup \{y \leftarrow y\theta \mid y \in Dom(\theta) \setminus Dom(\sigma)\}$$

- 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  $\rho$  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 egyesíthetőek ha létezik egy olyan  $\sigma$  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.  $\sigma_2$  és  $\sigma_3$  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  $\sigma_1$ ,
    - b.  $A_2\sigma_1$  és  $B_2\sigma_1$  legáltalánosabb egyesítője  $\sigma_2$ ,
    - c.  $A_3\sigma_1\sigma_2$  és  $B_3\sigma_1\sigma_2$  legáltalánosabb egyesítője  $\sigma_3$ ,
    - 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.)  $\text{mgu}(\text{leaf}(V), \text{leaf}(5))$  (4., majd 2. szerint) =  $\{V \leftarrow 5\} = \sigma_1$
    - (b.)  $\text{mgu}(V\sigma_1, S) = \text{mgu}(5, S)$  (3. szerint) =  $\{S \leftarrow 5\} = \sigma_2$
  - tehát  $\text{mgu}(A, B) = \sigma_1 \otimes \sigma_2 = \{V \leftarrow 5, S \leftarrow 5\}$
- $A = \text{node}(\text{leaf}(X), T)$ ,  $B = \text{node}(T, \text{leaf}(3))$ 
  - (4.)  $A$  és  $B$  neve és argumentumszáma megegyezik
    - (a.)  $\text{mgu}(\text{leaf}(X), T)$  (3. szerint) =  $\{T \leftarrow \text{leaf}(X)\} = \sigma_1$
    - (b.)  $\text{mgu}(T\sigma_1, \text{leaf}(3)) = \text{mgu}(\text{leaf}(X), \text{leaf}(3))$  (4., majd 2. szerint) =  $\{X \leftarrow 3\} = \sigma_2$
  - tehát  $\text{mgu}(A, B) = \sigma_1 \otimes \sigma_2 = \{T \leftarrow \text{leaf}(3), X \leftarrow 3\}$

## Az eljárás-redukciós végrehajtási modell

- A Prolog végrehajtás (ismétlés):
  - egy adott célsorozat futtatása egy adott programra vonatkozóan,
  - eredménye lehet:
    - siker – változó-behelyettesítésekkel
    - megüszülés (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  $\Rightarrow$ meghiúsulás
  - 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  $\Rightarrow$ meghiúsulás

## 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

- 1 (Kezdeti beállítások:) A verem üres,  $CS :=$  kezdeti célsorozat
- 2 (Beépített elj.): Ha  $CS$  első hívása beépített akkor hajtsuk végre a red. lépést.
  - a. Ha ez sikertelen  $\Rightarrow$  6. lépés.
  - b. Ha ez sikeres,  $CS :=$  a redukciós lépés eredménye, és  $\Rightarrow$  5. lépés.
- 3 (Klózszámláló kezdőértékezése:)  $I = 1$ .
- 4 (Redukciós lépés:) Tekintsük  $CS$  első hívására alkalmazható klózek 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.
  - a. Ha  $I > N \Rightarrow$  6. lépés.
  - b. Redukciós lépés a lista  $I$ -edik klóza és a  $CS$  célsorozat között.
  - c. Ha ez sikertelen, akkor  $I := I+1$ , és  $\Rightarrow$  4a. lépés.
  - d. Ha  $I < N$  (nem utolsó), akkor veremljük  $\langle CS, I \rangle$ -t.
  - e.  $CS :=$  a redukciós lépés eredménye.
- 5 (Siker:) Ha  $CS$  üres, akkor sikeres vég, egyébként  $\Rightarrow$  2. lépés.
- 6 (Sikertelenség:) Ha a verem üres, akkor sikertelen vég.
- 7 (Visszalépés:) Leemeljük a (nem üres) verem tetejéről a  $\langle CS, I \rangle$ -párt,  $I := I+1$ , és  $\Rightarrow$  4. lépés.



## A faösszegző program többirányú aritmetikával

- A 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 működé.

```
% 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.
```

```
% 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.
```

```
% tree_sum2(?Tree, ?S):
tree_sum2(leaf(Value), Value).
tree_sum2(node(Left,Right), S) :-
 plus(SL, SR, S),
 tree_sum2(Left, SL),
 tree_sum2(Right, SR).
```

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

# A többirányú faösszegző program keresési tere

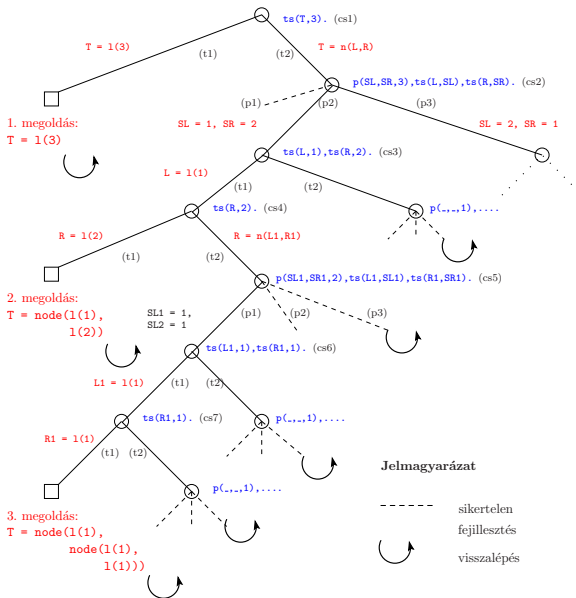
A program:

```

p(1, 1, 2). (p1)
p(1, 2, 3). (p2)
p(2, 1, 3). (p3)

ts(1(V), V). (t1)
ts(n(L,R), S) :-
 p(SL,SR,S),
 ts(L, SL),
 ts(R, SR). (t2)

```



# Tartalom

3

## Prolog alapok

- Prolog bevezetés – néhány példa
- A Prolog nyelv alapszintaxisa
- Nyomkövetés: 4-kapus doboz modell
- Listakezelő eljárások Prologban
- További vezérlési szerkezetek
- Operátorok
- Prolog végrehajtás – összefoglalás, pontosítás
- Ízelítő a „Haladó Prolog” részből

## Az „univ” beépített eljárás (=.. /2)

- Formula (Form): az ‘x’ atom; szám; Form1 + Form2; Form1 \* Form2
  - Számoljuk ki egy formula értékét egy adott x behelyettesítés mellett!
- % value\_of(+Form, +XE, ?E): az x=XE helyettesítéssel Form értéke E.

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

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

```
value_of0(Frm, X, V) :-
 Frm = *(P1,P2),
 value_of0(P1, X, V1),
 value_of0(P2, X, V2),
 FrmV = *(V1,V2),
 V is FrmV.
```

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

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

- value\_of/3 minden az is/2 által elfogadott bináris függvényre működik!
- | ?- value\_of(exp(100,min(x,1/x)), 2, V).       $\implies$       V = 10.0 ? ; no

## 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.           | ⇒ | <b>hiba</b>        |
| ?- 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]]  |

## 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 egy másik alkalmazása:

```
% 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).
```

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

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

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

```
| ?- findall(X, (member(X, [1,7,8,3,2,4]), X>3), L).
```

```
 => L = [7,8,4] ? ; no
```

```
| ?- findall(Y, member(X-Y, [a-c,a-b,b-c,c-e,b-d]), L).
```

```
 => L = [c,b,c,e,d] ? ; no
```

- Az eljárás jelentése (deklaratív szemantikája):

$$\text{Lista} = \{ \text{Gyűjtő } \mathbf{m\acute{a}solat} \mid (\exists X \dots Z) \text{Cél igaz} \}$$

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.



## Jobbrekurzív predikátumok akkumulátorok segítségével

- 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. ($\equiv \sum L = S-S0$)
```

```
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.

# Hogyan írunk 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

- Kétagumentumú C függvény  $\implies$  2+1-argumentumú Prolog eljárás.
- A függvény eredménye  $\implies$  utolsó arg.:  $\text{hatv}(+A, +H, ?E): A^H = E$ .
- Ciklus  $\implies$  segédeljárás:  $\text{hatv}(+A0, +H0, +E0, ?E): A0^{H0} * E0 = E$ .
- »a« és »h« C változók  $\implies$  »+A0« és »+H0« bemenő *paraméterek* (nem kell végérték),  
»e« C változó  $\implies$  »+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;
}

```

## IV. rész

# Erlang alapok

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

# Tartalom

4

## Erlang alapok

- **Bevezetés**
- Típusok
- Az Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabb rendű függvények, függvényérték
- Műveletek, beépített függvények
- Örök
- Válogatás a könyvtári függvényekből (OTP 18 V7.3 STDLIB)
- Kivételkezelés
- Típus-specifikáció
- Keresési feladat pontos megoldása funkcionális megközelítésben
- Listák használata: futamok
- Prolog és Erlang összehasonlítása
- Rekurzív adatstruktúrák

## Funkcionális programozás (FP): 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*.
- Az FP fő jellemzői: függvények, nem frissíthető változók, rekurzió.

Példák funkcionális programozási nyelvekre, nyelvcsaládokra:

- Lisp (Common Lisp), Scheme, Clojure (JVM-en fut)
- SML, Caml, Caml Light, OCaml, Alice, F# (.NET)
- Clean, Haskell
- Erlang, Elixir (Erlang VM-en fut)

# Az Erlang nyelv

- 1985: megszületik „Ellemtelben” (Ericsson–Televerket labor)
  - Agner Krarup Erlang dán matematikus, ill. Ericsson language
  - 1985-86: első interpreter Prologban! (Joe Armstrong)
- 1991: első megvalósítás, első projektek
- 1997: első OTP (Open Telecom Platform)
- 1998-tól: nyílt forráskódú, szabadon használható  
<http://www.erlang.org/>
- Funkcionális alapú (functionally based)
- Párhuzamos programozást segítő (concurrency-oriented)
- Hibatűrő (fault-tolerant) – hatékony hibakezelés
- Skálázható (scalable)
- Gyakorlatban használt  
[http://en.wikipedia.org/wiki/Erlang\\_\(programming\\_language\)#Distribution](http://en.wikipedia.org/wiki/Erlang_(programming_language)#Distribution),  
<https://www.erlang-solutions.com/>

„Programming is fun!”

## Erlang-szakirodalom (egy kivételével angolul)

- Simon St. Laurent: *Introducing Erlang. Getting Started in Functional Programming*. O'Reilly, 2013.  
<http://shop.oreilly.com/product/0636920025818.do>
- Learn You Some Erlang for great good! (online is olvasható)  
<http://learnyousomeerlang.com>
- Joe Armstrong: *Programming Erlang. Software for a Concurrent World. Second Edition. The Pragmatic Programmers*, 2013.  
<http://www.pragprog.com/book/jaerlang2/programming-erlang>
- Francesco Cesarini, Simon Thompson: *Erlang Programming*. O'Reilly, 2009. <http://oreilly.com/catalog/9780596518189/>

### További irodalom:

- Online dokumentáció: <http://erlang.org/doc.html>
- Lokális dokumentáció (Csak Linuxon 'erlang-manpages' csomaggal):  
`erl -man <module>`, ahol `<module>` = `erlang`, `lists`, `dict`, `sets`, `io` stb.
- Wikibooks on Erlang Programming  
[http://en.wikibooks.org/wiki/Erlang\\_Programming](http://en.wikibooks.org/wiki/Erlang_Programming)
- ERLANG összefoglaló magyarul  
<http://nyelvek.inf.elte.hu/leirasok/Erlang/>



## Erlang runtime system (interpreter)

- Erlang shell (interaktív értelmező) indítása

```
$ erl
```

```
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:...
Eshell V7.3 (abort with ^G)
```

```
1> 3.2 + 2.1 * 2.
```

```
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
```

```
1> 3.2 + 2.1 * 2.
```

```
7.4
```

```
...
```

Lezárás és  
indítás „pont-  
bevitel”-lel.

```
...
```

```
7> % ctrl-G
```

```
User switch command
```

```
--> q % Itt nem kell a pont!
```

# 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(); ls(Dir) -- list files in the current folder; list files in folder <Dir>
m(), m(Mod) -- which modules are loaded; 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 fordítása, futtatása

### 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 sikertelen
```

```
1> c(bevezeto). % fordítás
```

```
{ok,bevezeto}
```

```
2> bevezeto:fac(5). % futtatás
```

```
120
```

```
3> fac(5). % futtatás
```

```
** exception error: undefined shell command fac/1
```

```
4> bevezeto:fac(5) % futtatás
```

```
4>
```

```
4> . % a pont (.) kell a kiértékelés elindításához
```

```
120
```

## Listakezelés – rövid példák (1)

```
1> L1 = [10,20,30]. % új változó kötése, '=' a mintaillesztés, kötés
[10,20,30]
2> H = hd(L1). % hd: Built-in function (BIF)
10
3> b(). % kötött változók kiírása, lásd help().
H = 10
L1 = [10,20,30]
ok
4> T = tl(L1). % tl: Built-in function
[20,30]
5> T := [20|[30|[[]]]]. % egyenlőségvizsgálat
true
6> hd(tl(L1)). % kifejezés közvetlenül is kiértékelhető
20
7> v(6). % a v() paranccsal egy bármely érték kiírható
20
8> tl([]). % mi az üres lista farka?
** exception error: bad argument
 in function tl/1
 called as tl([])
```

## Listakezelés – rövid példák (2)

### bevezeto.erl – folytatás

*% sum(L) az L számlista összege.*

sum([]) -> 0;

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

```
9> c(bevezeto).
```

```
ok,bevezeto
```

```
10> f(L1). % forget L1: szabadítsd fel L1-et
```

```
ok
```

```
11> L1 = [10,20.5,30.5].
```

```
[10,20.5,30.5]
```

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

```
61.0
```

```
13> bevezeto:sum(tl(L1)).
```

```
51.0
```

```
14> bevezeto:sum(tl(tl(tl((L1))))).
```

```
0
```

```
15> bevezeto:sum("abc").
```

```
294
```

## Listakezelés – rövid példák (3)

bevezeto.erl – folytatás

*% 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]).
```

```
10> c(bevezeto).
```

```
ok,bevezeto
```

```
11> L1.
```

```
[10,20.5,30.5]
```

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

```
[10,20.5,30.5,a,b,c,d]
```

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

```
[30.5,20.5,10,a,b,c,d]
```

# Tartalom

4

## Erlang alapok

- Bevezetés
- **Típusok**
- Az Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabb rendű függvények, függvényérték
- Műveletek, beépített függvények
- Örök
- Válogatás a könyvtári függvényekből (OTP 18 V7.3 STDLIB)
- Kivételkezelés
- Típuspecifikáció
- Keresési feladat pontos megoldása funkcionális megközelítésben
- Listák használata: futamok
- Prolog és Erlang összehasonlítása
- Rekurzív adatstruktúrák



# 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.  $2\#11111110 = 8\#376 = 16\#fe = 10\#254$
  - 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

# Atom

- Névkonstans (nem fűzér!)
- Kisbetűvel kezdődő, bővített alfanumerikus<sup>1</sup> karaktersorozat, pl. `sicstus`, `erlang_OTP`, `email@info_11`
- Bármilyen<sup>2</sup> karaktersorozat is az, ha egyszeres idézőjelbe tesszük, pl. `'SICStus'`, `'erlang OTP'`, `'35 May'`, `'síró ürom'`
- Hossza tetszőleges, vezérlőkaraktereket is tartalmazhat, pl. `'hosszú atom, á-val, é-vel, ó-val, ú-val, rövid ö-vel és ü-vel'`  
`'atom, formázókarakterekkel (\n\r\s\t)'`<sup>3</sup>
- Saját magát jelöli
- Hasonló a Prolog névkonstanshoz (atom)
- C++, Java nyelvekben a legközelebbi rokon: enum

---

<sup>1</sup>Bővített alfanumerikus: kis- vagy nagybetű, számjegy, aláhúzás (`_`), kukac (`@`).

<sup>2</sup>Latin-1 vagy a latin-1 készletbe tartozó, de utf-8 kódolású karakter lehet (R18).

<sup>3</sup>`\n`: new line, `\r`: return, `\s`: space, `\t`: horizontal tabulator

# Függvény

- A függvény is érték: változóhoz köthető, adatszerkezet eleme lehet, ...
- Példák:

```
1> F = fun bevezeto:fac/1.
```

```
#Fun<bevezeto.fac.1>
```

```
2> F(6).
```

```
720
```

```
3> L = [fun erlang:'+' /2, fun erlang:'-' /2].
```

```
[#Fun<erlang.+ .2>, #Fun<erlang.- .2>]
```

```
4> (hd(L))(4,3).
```

```
7
```

- Részletesen később, a „Magasabb rendű függvények”c. részben

## 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

## 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
  - Egy elemű lista: `[elem]`
  - Fejből-farokból létrehozott lista: `[elem|[]]`, `['elem1'|['elem2']]`
  - Több elemű 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]`

## Füzér (string)

- Csak rövidítés, tkp. karakterkódok listája, pl.  
`"erl" ≡ [$e,$r,$l] ≡ [101,114,108]`
- Az Erlang shell a nyomtatható karakterkódok listáját füzérként 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

4

## Erlang alapok

- Bevezetés
- Típusok
- **Az Erlang szintaxis alapjai**
- Mintaillesztés
- Listanézet
- Magasabb rendű függvények, függvényérték
- Műveletek, beépített függvények
- Örök
- Válogatás a könyvtári függvényekből (OTP 18 V7.3 STDLIB)
- Kivételkezelés
- Típusspecifikáció
- Keresési feladat pontos megoldása funkcionális megközelítésben
- Listák használata: futamok
- Prolog és Erlang összehasonlítása
- Rekurzív adatstruktúrák



# Term

- A *term* tetszőleges típusú adatszerkezetet jelent az Erlangban
- Minden termnek van *típusa*; néhány típusal (reference, port, pid és binary) nem foglalkozunk
- *Közelítő rekurzív definíció*: szám-, atom- és függvényértékekből, kötött változókból, ill. *termekből* ennes- és listakonstruktorokkal felépített, tovább nem egyszerűsíthető kifejezés
- Példák
  - Term (mert tovább nem egyszerűsíthető)  
123456789  
{'Diák Detti', [{khf, [cekla, prolog, erlang, prolog]}]}
  - Nem term (mert tovább egyszerűsíthető)  
5+6, mert műveletet tartalmaz  
fun erlang:'+' /2(5,6), mert függvényalkalmazást tartalmaz

# Változó

- Nagybetűvel kezdődő, bővített alfanumerikus karaktersorozat<sup>4</sup>, 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

---

<sup>4</sup>Bővített alfanumerikus: kis- vagy nagybetű, számjegy, aláhúzás (  ), kukac (@).

## Minta(kifejezés)

- Minta(kifejezés): olyan term, amelyben szabad változó is lehet
  - változók  $\subset$  minták
  - termek  $\subset$  minták

Igaz, néhány ritka ellenpéldától eltekintve.

Ez például hibás, mert függvényértékre nem lehet mintát illeszteni:

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

További példa:

```
1> A = fun (X) -> X+1 end.
#Fun<erl_eval.6.50752066>
2> {A, B} = {fun (X) -> X+1 end, 23.}
#Fun<erl_eval.6.50752066>,23
3> fun (X) -> X+1 end = A.
* 1: illegal pattern
4>
```

# Kifejezés

A kifejezés lehet:

- term (már tárgyaltuk)
- szekvenciális kifejezés
- összetett kifejezés
- függvényalkalmazás
- örökifejezés (később lesz róla szó)
- egyéb összetett kifejezés: `if`, `case`, `try...catch`, `catch stb.` (később lesz róluk szó)

A kifejezés *kiértékelése* alapvetően *mohó* (eager, strict evaluation).

```
4> Nevező = 0.
```

```
0
```

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

```
** exception error: an error occurred when evaluating an arithmetic expression
```

## Szekvenciális kifejezés

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

Összetett kifejezés, függvényalkalmazás

Összetett kifejezés

- Kiértékelhető műveleteket, függvényeket is tartalmazó kifejezés, pl. $X=2+3$, `[{5+6, math:sqrt(2), bevezeto:fac(X)}, alma]`
- Különbözik a termtől, ahol a műveletvégzés/függvényhívás tiltva van

Függvényalkalmazás

- Szintaxisa
 - `fnév(arg1, arg2, ..., argn)`
vagy
 - `modul:fnév(arg1, arg2, ..., argn)`
- Például

```
8> length([a,b,c]).
3
9> erlang:tuple_size({1,a,'A',"1aA"}).
4
10> erlang:'+'(1,2).
3
```

Függvénydeklaráció

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

- Alakja:

```
fnév(A11, ..., A1m) [when ŐrSzekv1] -> SzekvenciálisKif1;
... ;
fnév(An1, ..., Anm) [when ŐrSzekvn] -> SzekvenciálisKifn.
```

- A függvényt a neve, az „aritása” (paramétereinek száma), valamint a moduljának a neve azonosítja.
- Az azonos nevű, de eltérő aritású függvények nem azonosak!
- Példa:

```
fac(N) -> fac(N, 1).
```

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

- (Az öröket kicsit később vezetjük be.)

Tartalom

4

Erlang alapok

- Bevezetés
- Típusok
- Az Erlang szintaxis alapjai
- **Mintaillesztés**
- Listanézet
- Magasabb rendű függvények, függvényérték
- Műveletek, beépített függvények
- Örök
- Válogatás a könyvtári függvényekből (OTP 18 V7.3 STDLIB)
- Kivételkezelés
- Típuspecifikáció
- Keresési feladat pontos megoldása funkcionális megközelítésben
- Listák használata: futamok
- Prolog és Erlang összehasonlítása
- Rekurzív adatstruktúrák

Minta, mintaillesztés (egyesítés)

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

- Példák:

$\text{Pi} = 3.14159$	\rightsquigarrow^5	$\text{Pi} \mapsto^6 3.14159$
$3 = P$	\rightsquigarrow	hiba (jobb oldal nem tömör)
$[\text{H1} \text{T1}] = [1, 2, 3]$	\rightsquigarrow	$\text{H1} \mapsto 1, \text{T1} \mapsto [2, 3]$
$[1, 2 \text{T2}] = [1, 2, 3]$	\rightsquigarrow	$\text{T2} \mapsto [3]$
$[\text{H2} [3]] = [1, 2, 3]$	\rightsquigarrow	meghiúsulás, hiba
$\{\text{A1}, \text{B1}\} = \{\{a\}, \text{'Beta'}\}$	\rightsquigarrow	$\text{A1} \mapsto \{a\}, \text{B1} \mapsto \text{'Beta'}$
$\{\{a\}, \text{B2}\} = \{\{a\}, \text{'Beta'}\}$	\rightsquigarrow	$\text{B2} \mapsto \text{'Beta'}$

⁵ $\text{Kif} \rightsquigarrow$ jelentése: „Kif kiértékelése után”.

⁶ $X \mapsto V$ jelentése: „X a V értékhez van kötve”.

Mintaillesztés függvény klózaira – 1. példa

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

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. és 2. klóz, sikeres: 3. klóz  
3  
4> khf:kiadott(java). % három sikertelen illesztés után hiba  
** exception error: no function clause matching khf:kiadott(java) ...
```

Mintaillesztés függvény klózaira – 2. példa

- Hányszor szerepel egy elem egy listában? Első megoldásunk:

khf.erl – folytatás

```
-spec elofordulo(E::term(), L::[term()]) -> N::integer().
```

% Az E elem az L listában N-szer fordul elő.

```
elofordulo(E, []) -> 0; % 1.
```

```
elofordulo(E, [E|Farok]) -> 1 + elofordulo(E, Farok); % 2.
```

```
elofordulo(E, [Fej|Farok]) -> elofordulo(E, Farok). % 3.
```

```
5> khf:elofordulo(a, [a,b,a,1]). % 2. klóz, majd 3., 2., 3., 1.
```

```
2
```

```
6> khf:elofordulo(java, [cekla,prolog,prolog]). % 3., 3., 3., 1.
```

```
0
```

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

Kitérő: változók elnevezése

- Az előző függvény fordításakor figyelmeztetést kapunk:

Warning: variable 'E' is unused

Warning: variable 'Fej' is unused

- A figyelmeztetés kikapcsolható alulvonással (_) kezdődő nevű változóval

khf.erl – folytatás

```

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

```

- Ilyen esetekben a „névtelen” _ változót is használhatjuk, de jobb az _<változónév> használata, mert utal a szerepére
- A „névtelen” _ változó nem értékelhető ki, ezért tömör kifejezésben nem használható
- Több _ változó is lehet ugyanabban a mintában, például:
 $[H, _, _] = [1, 2, 3] \rightsquigarrow H \mapsto 1$
- Találós kérdés: miben különböznek az alábbi mintaillesztések, ha $L = [a]$?
 a) $A = \text{hd}(L)$. b) $[A | _] = L$. c) $[A, _ | _] = L$.

Mintaillesztés függvény klózaira – 3. példa

- Teljesítette-e egy hallgató a khf-követelményeket?

```
D1 = {'Diák Detti', [{khf,[cekla,prolog,erlang,prolog]}, {zh,59}]}.
```

```
D2 = {'Néma Levi', [{khf,[prolog,erlang]}, {zh,32}]}.
```

```
D3 = {'Laza Lali', [{khf,[erlang,prolog,erlang]}, {zh,22}]}.
```

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, [_|L]}) ->  
    megfelelt(K, {Nev, L});  
megfelelt(_, {_, []}) ->  
    false.
```

„Biztonságos” illesztés: ha egy mindig sikerül

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

khf.erl – folytatás

```
-spec bizt_kiadott(Ny::atom()) -> {ok, Db::integer()} | error.
% Az Ny nyelven Db darab kisházit adtak ki.
bizt_kiadott(cekla) -> {ok, 1};
bizt_kiadott(prolog) -> {ok, 3};
bizt_kiadott(erlang) -> {ok, 3};
bizt_kiadott(_Ny) -> error. % ez a klóz mindenre illeszkedik
```

- Az ok és az error atomokat konvenció szerint választottuk
- Kötés: ha a minta egyetlen szabad változó (_Ny), az illesztés sikeres
- Lássunk két példát!

```
7> khf:bizt_kiadott(cekla).
{ok,1}
8> khf:bizt_kiadott(java).
error
```

- De hogyan férünk hozzá az eredményhez?

Feltételes kifejezés mintaillesztéssel (case)

- `case` Kif of

```
Minta1 [when ŐrSzekv1] -> SzekvenciálisKif1;
```

```
...
```

```
Mintan [when ŐrSzekvn] -> SzekvenciálisKifn
```

```
end.
```

- Kiértékelés: balról jobbra, fölülről lefelé
- 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
```

További példa case használatára

- Az adott nyelven a kisházik hány százalékát adták be?

khf.erl – folytatás

```
-spec bizt_teljesitmeny(Nyelv::atom(), Beadott_db::integer()) ->
%           {ok, Teljesitmeny::float()} | error.
bizt_teljesitmeny(Nyelv, Beadott_db) ->
    case bizt_kiadott(Nyelv) of
        {ok, Kiadott_db} -> {ok, round(100 * (Beadott_db / Kiadott_db))};
        error             -> error
    end.
```

- Függvény klózai összevonhatók a case-zel:

```

bizt_kiadott(cekla)   -> {ok, 1};
bizt_kiadott(prolog) -> {ok, 3};
bizt_kiadott(erlang) -> {ok, 3};
bizt_kiadott(_Ny)    -> error.

```

≡

```

bizt_kiadott(Ny) ->
    case Ny of
        cekla -> {ok, 1};
        prolog -> {ok, 3};
        erlang -> {ok, 3};
        _Ny -> error
    end.

```


Tartalom

4

Erlang alapok

- Bevezetés
- Típusok
- Az Erlang szintaxis alapjai
- Mintaillesztés
- **Listanézet**
- Magasabb rendű függvények, függvényérték
- Műveletek, beépített függvények
- Örök
- Válogatás a könyvtári függvényekből (OTP 18 V7.3 STDLIB)
- Kivételkezelés
- Típuspecifikáció
- Keresési feladat pontos megoldása funkcionális megközelítésben
- Listák használata: futamok
- Prolog és Erlang összehasonlítása
- Rekurzív adatstruktúrák

Listanézet (List comprehension)

- Listanézet: `[Kif || Minta <- Lista, Feltétel]`

Közelítő definíció: A listanézet

- a `Minta` mintától függő `Kif` kifejezések *listája*,
 - ahol a `Minta` minta a `Lista` lista egy olyan eleme,
 - amelyre a `Feltétel` feltétel igaz.
- A `Feltétel` feltétel tetszőleges logikai (`true` v. `false` atom értékű) kifejezés lehet. A `Minta` mintában előforduló változónevek elfedik a listanézetten kívüli, azonos nevű változókat.
- A listanézet pontos szintaxisa:
`[X || Q1, Q2, ...]`, ahol `X` tetszőleges kifejezés, `Qi` pedig generátor (`Minta <- Lista`) vagy szűrőfeltétel (predikátum) lehet.
- A listanézet sokféle programozási nyelvben elérhető, lásd:
[https://en.wikipedia.org/wiki/Comparison_of_programming_languages_\(list_comprehension\)](https://en.wikipedia.org/wiki/Comparison_of_programming_languages_(list_comprehension))

Listanézet: kis példák

```
1> [X || X <- [1,2,3]].      % { x | x ∈ {1,2,3} }
[1,2,3]
2> [2*X+1 || X <- [1,2,3]]. % { 2·x | x ∈ {1,2,3} }
[3,5,7]
3> [2*X || X <- [1,2,3], X rem 2 /= 0, X > 2].
[6]
4> lists:seq(1,3).          % egészek 1-től 3-ig
[1,2,3]
5> [{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}]
6> [{X,Y} || X <- lists:seq(1,4), Y <- lists:seq(1,3), X > Y].
[{2,1},
 {3,1},{3,2},
 {4,1},{4,2},{4,3}]
```

Listanézet: további példák

- Pitagoraszai számhármások

lcomp.erl

```
-spec pitag(N)(N::integer()) -> Ps::[integer()]
% Ps olyan pitagoraszai számhármások listája, melyek összege legfeljebb N
pitag(N) ->
    L = lists:seq(1,N),
    [{A,B,C} || A <- L, B <- L, C <- L,
            A+B+C =< N,
            A*A+B*B == C*C].
```

- Hányszor fordul elő egy elem egy listában?

lcomp.erl - folytatás

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

- A khf követelményeket teljesítő hallgatók

```
L=[{'Diák Detti', [{khf, [erlang, prolog, prolog]}]}, {'Laza Lali', []}],
[Nev || {Nev, M} <- L, khf:megfelelt(khf, {Nev, M})].
```

Listanézet: gyorsrendezés (Quicksort)

`lcomp.erl` - folytatás

```
-spec qsort(Us::[term()]) -> Ss::[term()]  
% Az Us lista elemeinek monoton növekedő listája Ss  
qsort([]) ->  
    [];  
qsort([Pivot|Tail]) ->  
    qsort([X || X <- Tail, X < Pivot])  
    ++ [Pivot] ++  
    qsort([X || X <- Tail, X >= Pivot]).
```

```
7> lcomp:qsort([34,1,55,78,43.2,math:pi(),math:exp(1),31.7]).  
[1,2.718281828459045,3.141592653589793,31.7,34,43.2,55,78]  
8> lcomp:qsort([ab,acb,aca,bca,bbca,cab,bca,bac,abc,a,b,c]).  
[a,ab,abc,aca,acb,b,bac,bbca,bca,bca,c,cab]  
9> lcomp:qsort("the quick brown fox jumps over the lazy dog").  
"    abcdeefghhijklmnooopqrrsttuuvwxyz"  
10> lcomp:qsort(["baba",baba,9.3,6,fun math:exp/1,[4,5,3],[4,5],2,3]).  
[6,9.3,baba,#Fun<math.exp.1>,2,3,[4,5],[4,5,3],"baba"]
```

Listanézet: permutáció

```
lcomp.erl -folytatás
```

```
-spec perms(Xs::[term()]) -> Zss::[[term()]]
% Az Xs lista elemeinek összes permutációját tartalmazó lista Zss
perms([]) ->
    [[]];
perms(L) ->
    [[H|T] || H <- L, T <- perms(L--[H])].
```

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

- Példák:

```
11> [a,b,c,a,b,c,a,b,c]--[a,b,c].
[a,b,c,a,b,c]
12> [a,b,c,a,b,c,a,b,c]--[a,b,c,c,b,a].
[a,b,c]
12> lcomp:perms([a,b,c]).
[[a,b,c],[a,c,b],[b,a,c],[b,c,a],[c,a,b],[c,b,a]]
```

Tartalom

4

Erlang alapok

- Bevezetés
- Típusok
- Az Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- **Magasabb rendű függvények, függvényérték**
- Műveletek, beépített függvények
- Örök
- Válogatás a könyvtári függvényekből (OTP 18 V7.3 STDLIB)
- Kivételkezelés
- Típusspecifikáció
- Keresési feladat pontos megoldása funkcionális megközelítésben
- Listák használata: futamok
- Prolog és Erlang összehasonlítása
- Rekurzív adatstruktúrák

Függvényérték

- A funkcionális nyelvekben a függvény is *érték* (már láttunk rá példákat):
 - leírható (jelölhető),
 - van típusa,
 - névhez (változóhoz) köthető,
 - adatszerkezet eleme lehet,
 - **paraméterként átadható,**
 - **függvényalkalmazás eredménye lehet** (zárójelezni kell!).

- Névtelen függvény (függvényjelölés, lambdajelölés) mint érték

```
fun (A11, ..., A1m) [when ŐrSzekv1] -> SzekvenciálisKif1;
...;
(An1, ..., Anm) [when ŐrSzekvn] -> SzekvenciálisKifn
end.
```

- Másutt már deklarált függvény mint érték

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


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> Fs = [Area1, Osszeg, fun bevezeto:sum/1, 12, area].
[#Fun<erl_eval.6.13229925>,#Fun<bevezeto.sum.1>,...]
8> (hd(Fs))({circle, 2}). % zárójelezni kell a függvényértéket!
12.56636
% hd/1 itt magasabb rendű függvény, zárójelezni kell az értékét
```

Magasabb rendű függvények és alkalmazásuk (1)

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

`mrend.erl` – `map/2` egy megvalósítása

```
map2(_F, []) -> [];
map2(F, [X|Xs]) -> [F(X) | map2(F, Xs)].
```

```
9> lists:map(fun erlang:length/1, ["alma", "korte"]).
[4,5]           % erlang:length/1: Built-In Function, lista hossza
10> mrend:map2(Osszeg, [[10,20], [10,20,30]]).
[30,60]
11> L=[{'Diák Detti',[{khf,[erlang:prolog,prolog]}]},{'Laza Lali',[]}].
[{'Diák Detti',[{khf,[erlang:prolog,prolog]}]},{'Laza Lali',[]}].
12> lists:map(fun(Hallg) -> khf:megfelelt(khf, Hallg) end, L).
[true,false]
13> [length(S) || S <- ["alma", "korte"]]. % leképzés listanézetel
[4,5]
```

Magasabb rendű függvények és alkalmazásuk (2)

- **A filter függvény**

- Szűrés: `lists:filter(Pred, List)`
- Eredménye a List lista Pred-et kielégítő elemeinek listája

`mrend.erl` – `filter/2` egy megvalósítása

```
filter2(_, [])          -> [];
filter2(P, [Fej|Farok]) -> case P(Fej) of
                            true  -> [Fej|filter2(P, Farok)];
                            false -> filter2(P, Farok)
                        end.
```

```
14> lists:filter(fun erlang:is_number/1, [x, 10, L, 20, {}]).
```

```
[10,20]
```

```
15> mrend:filter2(fun erlang:is_tuple/1, [x, {7,3}, 10, L, 20, {}]).
```

```
[{7,3},{}]
```

```
16> lists:filter(fun(Hallg) -> khf:megfelelt(khf, Hallg) end, L).
```

```
[{'Diák Detti', [{khf, [erlang,prolog,prolog]}]}]
```

- *Fejtörő*: miért érdemes leírni kétszer a `filter(P, Farok)` hívást?
- *Fejtörő*: hogyan lehet megvalósítani a `filter/2`-t `case` nélkül?

Magasabb rendű függvények és alkalmazásuk (3)

- Hányszor szerepel egy elem egy listában? Új megoldásunk:

mrend.erl – folytatás

```
-spec elofordul3(E::term(), L::[term()]) -> N::integer().
```

```
% Az E elem az L listában N-szer fordul elő.
```

```
elofordul3(Elem, L) ->
```

```
length(lists:filter(fun(X) -> X == Elem end, L)).
```

```
17> mrend:elofordul3(prolog, [cekla,prolog,prolog]).
```

```
2
```

- A névtelen függvényben – az elofordul3/2 törzsében – természetesen felhasználhatjuk az elofordul3/2 fejében lekötött Elem változót.

Redukálás a fold függvényekkel

- Jobbról balra haladva: `lists:foldr(Fun, Acc, List)`
- Balról jobbra haladva: `lists:foldl(Fun, Acc, List)`
- Eredménye 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])` \equiv -2

`lists:foldl(fun(X, Acc) -> X - Acc end, 0, [1,2,3,4])` \equiv 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	<pre>sum(Acc, []) -> Acc; sum(Acc, [H T]) -> plus(H, sum(Acc, T)).</pre>	<pre>foldr(Fun, Acc, []) -> Acc; foldr(Fun, Acc, [H T]) -> Fun(H, foldr(Fun, Acc, T)).</pre>
---	--	--

L	<pre>sum(Acc, []) -> Acc; sum(Acc, [H T]) -> sum(plus(H, Acc), T)).</pre>	<pre>foldl(Fun, Acc, []) -> Acc; foldl(Fun, Acc, [H T]) -> foldl(Fun, Fun(H, Acc), T)).</pre>
---	---	---

Tartalom

4

Erlang alapok

- Bevezetés
- Típusok
- Az Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabb rendű függvények, függvényérték
- **Műveletek, beépített függvények**
- Örök
- Válogatás a könyvtári függvényekből (OTP 18 V7.3 STDLIB)
- Kivételkezelés
- Típusspecifikáció
- Keresési feladat pontos megoldása funkcionális megközelítésben
- Listák használata: futamok
- Prolog és Erlang összehasonlítása
- Rekurzív adatstruktúrák

Listaműveletek

- Alpmműveletek: $hd(L)$, $tl(L)$, $length(L)$ (utóbbi lassú: $O(n)$!)
- Listák összefűzése ($A_s \oplus B_s$): $A_s ++ B_s$ vagy `lists:append(A_s, B_s)`
 $C_s = A_s ++ B_s \rightsquigarrow C_s \mapsto$ az A_s összes eleme a B_s elé fűzve az eredeti sorrendben
- Példa

```
1> [a, 'A', [65]] ++ [1+2, 2/1, 'A'] .
[a, 'A', "A", 3, 2.0, 'A']
```
- Listák különbsége: $A_s -- B_s$ vagy `lists:subtract(A_s, B_s)`
 $C_s = A_s -- B_s \rightsquigarrow C_s \mapsto$ az A_s olyan másolata, amelyből ki van hagyva a B_s -ben előforduló összes elem balról számított első előfordulása, feltéve, hogy volt ilyen elem A_s -ben
- Példák

```
1> [a, 'A', [65], 'A'] -- ["A", 2/1, 'A'] . % [65]=="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 ≠ 1.0
[1, 3]
```

Aritmetikai műveletek

- Matematikai műveletek
 - Előjel: +, - (precedencia: 1)
 - Multiplikatív műveletek: *, /, div, rem (precedencia: 2)
 - Additív műveletek: +, - (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

Összehasonlító műveletek (relációk)

- Egy reláció (összehasonlítás) eredménye a true vagy a false atom
- Termék összehasonlítási sorrendje (vö. típusok):
`number < atom < reference < function < port < pid < tuple < list < binary`
- Kisebb, egyenlő-kisebb, nagyobb-egyenlő, nagyobb reláció:
`<, =<, >=, >`
- Egyenlőségi reláció (aritmetikai egyenlőségre is):
`==, /=` **Ajánlás: helyette azonosan egyenlőt használjunk.**
- Azonosan egyenlő (különbséget tesz integer és float közt):
`:=, :=/=` Példa: `5.0 := 5` \rightsquigarrow false



Ezek lebegőpontos
értékre kerülendők:
`==, =<, >=, :=`

Elrettentő példák:

`10.1 - 9.9 == 0.2` \rightsquigarrow false

`(10.1 - 9.9) * 10` \rightsquigarrow 1.9999999999999993

`0.0000000000000001 + 1 == 1` \rightsquigarrow false

`0.0000000000000001 + 1 == 1` \rightsquigarrow true

integer \leftrightarrow float típuskonverzió

- Kerekítés (float \mapsto integer):
 - `erlang:trunc/1: trunc(5.8) ::= 5.`
 - `erlang:round/1: round(5.8) ::= 6.`
- Explicit típuskonverzió (integer \mapsto float):
 - `erlang:float/1: float(5) ::= 5.0.`

Logikai műveletek

- Mohó (strict) kiértékelésű logikai műveletek:
not, and, or, xor
- Lusta (lazy) kiértékelésű („short-circuit”) logikai műveletek:
andalso, orelse
- Csak a true és false atomokra, illetve ilyen eredményt adó kifejezésekre alkalmazhatóak
- Példák:

```
1> false and (3 div 0 == 2).
```

```
** exception error: an error occurred when evaluating  
    an arithmetic expression  
    in operator div/2  
    called as 3 div 0
```

```
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
 - többnyire 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

4

Erlang alapok

- Bevezetés
- Típusok
- Az Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabb rendű függvények, függvényérték
- Műveletek, beépített függvények
- **Őrök**
- Válogatás a könyvtári függvényekből (OTP 18 V7.3 STDLIB)
- Kivételkezelés
- Típuspecifikáció
- Keresési feladat pontos megoldása funkcionális megközelítésben
- Listák használata: futamok
- Prolog és Erlang összehasonlítása
- Rekurzív adatstruktúrák

A strukturális mintaillesztés finomítása őrrrel

- Nézzük újra a következő definíciót:

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

- Mi történik, ha megváltoztatjuk a klózok 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 őrr alkalmazásával!

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

Ismétlés: függvénydeklaráció és case

- Függvénydeklaráció:

```
fnév(A11, ..., A1m) [when ŐrSzekv1] -> SzekvenciálisKif1;  
...  
fnév(An1, ..., Anm) [when ŐrSzekvn] -> SzekvenciálisKifn.
```

- Feltételes mintaillesztés (case):

```
case Kif of  
  Minta1 [when ŐrSzekv1] -> SzekvenciálisKif1;  
  ...  
  Mintan [when ŐrSzekv1n] -> SzekvenciálisKifn  
end.
```


Örkifejezés

Az örkifejezésnek logikai értéket adó kifejezésnek kell lennie. Lehet:

- Term (vagyis tömör – tovább nem egyszerűsíthető – kifejezés)
- Örkifejezésekből aritmetikai, összehasonlító és logikai műveletekkel felépített kifejezés
- 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()`

Örkifejezésben **nem** lehet:

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

Őrkifejezés, őr, őrszekvencia

- **Őrkifejezés (guard expression)**

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

- **Őr (guard):** egyetlen őrkifejezés vagy őrkifejezések vesszővel (,) elválasztott, *konjunktív* sorozata

- Értéke `true`, ha az összes őrkifejezés `true` értékű (ÉS-kapcsolat)
- Ha az értéke `true` \leadsto *sikerül*, bármely más esetben \leadsto *megghiúsul*

- **Őrszekvencia (guard sequence):** egyetlen őr vagy örök pontosvesszővel (;) elválasztott, *diszjunktív* sorozata

- Értéke `true` (azaz *sikerül*), ha legalább egy őr `true` értékű (VAGY-kapcsolat)

Őrszekvencia, mintaillesztés

- Az őrszekvenciával olyan tulajdonságot írunk elő, amit *strukturális mintaillesztéssel* nem tudunk leírni
- Az őrszekvenciát a `when` kulcsszó vezeti be
- Az őrszekvenciában előforduló összes változónak *kötöttnek* kell lennie
- A mintaillesztés lépései klózválasztásnál, ill. `case`-nél:
 - Strukturális mintaillesztés (hasonló a Prolog illesztéséhez)
 - Őrszekvencia kiértékelése

Példa őr szekvencia használatára: 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}]
```

Példa őr szekvencia használatára: orok.erl (folytatás)

```

...
{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 egész, vagy ha N:=0, hiba miatt megghiúsul
    racionalis;
_ -> egyeb
end.

```

```

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

```

Feltételes kifejezés őrsekvenciával (if)

```
if
    ŐrSzekv1 -> SzekvenciálisKif1;
    ...
    ŐrSzekvn -> SzekvenciálisKifn
end.
```

- Kiértékelés: balról jobbra, fölülről lefelé.
- Értéke: az első true értékű őrsekvencia utáni szekvenciális kifejezés
- Ha nincs true értékű 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).
```

Az if a case speciális esete (1)

- A kiértékelendő kifejezés kiválasztására
 - a case kifejezést illeszt mintákra, majd őr szekvenciákat használ,
 - az if csak őr szekvenciákat használ.
- Az if helyettesítése case-zel (az Alapértelmezés sora opcionális):

```

case 1 of % _ = 1 mindig sikeres
  _ when ŐrSzekv1 -> Kif1;
  ...
  _ when ŐrSzekvn -> Kifn;
  _           -> Alapértelmezés
end

```

≡

```

if
  ŐrSzekv1 -> Kif1;
  ...
  ŐrSzekvn -> Kifn;
  true -> Alapért
end

```

- Fordítva: használhatunk-e case helyett if-et?

```

filter1(_P, []) -> [];
filter1(P, [Fej|Farok]) -> case P(Fej) of
  true -> [Fej|filter1(P,Farok)];
  false -> filter1(P,Farok)
end.

```

Az if a case speciális esete (2)

- Jó-e az alábbi függvénydeklaráció?

```
filter2e(_P, []) ->
  [];
filter2e(P, [Fej|Farok]) ->
  if
    P(Fej) -> [Fej|filter2e(P, Farok)];
    true  -> filtere2(P, Farok)
  end.
```

- Nem! Az if `P(Fej) -> [Fej|...]` hibás, mert örben nem lehet függvény. A hibaüzenet: `illegal guard expression`
- De egy új változó bevezetése megoldja a dolgot:

```
filter2a(_P, []) ->
  [];
filter2a(P, [Fej|Farok]) ->
  Cond = P(Fej),
  if
    Cond -> [Fej|filter2a(P, Farok)];
    true -> filter2a(P, Farok)
  end.
```


Tartalom

4

Erlang alapok

- Bevezetés
- Típusok
- Az Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabb rendű függvények, függvényérték
- Műveletek, beépített függvények
- Örök
- **Válogatás a könyvtári függvényekből (OTP 18 V7.3 STDLIB)**
- Kivételkezelés
- Típus-specifikáció
- Keresési feladat pontos megoldása funkcionális megközelítésben
- Listák használata: futamok
- Prolog és Erlang összehasonlítása
- Rekurzív adatstruktúrák

Füzérkezelő függvények – OTP 18 V7.3 string modul (1)

- `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,Chars)`, `cspan(Str,Chars)`
Az `Str` ama prefixumának hossza, amelyben kizárólag a `Chars`-beli karakterek fordulnak / nem fordulnak elő.
- `substr(Str,Start,Length)`, `substr(Str,Start)`
Az `Str` specifikált részfüzére.
- `to_lower(Str)`, `to_upper(Str)`, `to_lower(Char)`, `to_upper(Char)`
A füzér / karakter kisbetűs / nagybetűs változata.
- `to_integer(Str)`, `to_float(Str)`
Eredménye egy pár, melynek első tagja a füzér elejéről beolvasott egész / lebegőpontos szám, második tagja a füzér maradéka; hiba esetén a pár első tagja az `error` atom, második tagja a hiba oka.

Füzérkezelő függvények – OTP 18 V7.3 string modul (2)

- `tokens(Str, SepList)`
A `SepList` karakterei mentén füzérek listájára bontja az `Str`-t.
- `join(StrList, Sep)`
Az `StrList` – egymástól a `Sep` füzérel elválasztott – elemeiből álló füzér.
- `strip(Str)`, `strip(Str, Dir)`, `strip(Str, Dir, Char)`
Az `Str` füzér másolata az elejéről / végéről levágott formázó karakterek / `Char` karakterek nélkül.
`Dir = left | right | both`
- `left(Str, Num)`, `left(Str, Num, Chr)`, `right(Str, Num)`, `right(Str, Num, Chr)`, `centre(Str, Num)`, `centre(Str, Num, Chr)`
A `Str` füzér `Num` hosszú másolata balra / jobbra / középre igazítva, jobbról / balról / mindkét oldalról szóközökkel / `Chr` karakterekkel kiegészítve.

Részletek és továbbiak: Erlang OTP 18 V7.3 / Basic / stdlib / string.

<http://erlang.org/documentation/doc-7.3/doc/>

Listakezelő függvények – OTP 18 V7.3 lists modul (1)

- `nth(N,Lst)`, `nthtail(N,Lst)`, `last(Lst)`
Az `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 füzérré alakítva és egybefűzve.
Az `Lst` elemeinek típusa `atom`, `integer`, `float` és `string` lehet.
- `reverse(Lst)`, `reverse(Lst,T1)`
Az `Lst` megfordítva / megfordítva a `T1` 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.

Listakezelő függvények – OTP 18 V7.3 lists modul (2)

- `filter(Pred,Lst)`, `delete(Elem,Lst)`
Az `Lst` `Pred`-et kielégítő elemeiből álló / első `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)`
Az `Lst` elemei `Pred` / `N` darabszám 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.

Listakezelő függvények – OTP 18 V7.3 lists modul (3)

- `sublist(Lst, Len)`, `sublist(Lst, Start, Len)`
Az `Lst` 1-től / `Start`-tól kezdődő, `Len` hosszú része.
- `subtract(Lst1, Lst2) (--)`
Az `Lst1` lista `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 szerint / `Fun` szerint rendezett másolata.
- `merge(LstOfLsts)`
Az `LstOfLsts` listában lévő rendezett listák alapértelmezés szerinti összefuttatása.

Listakezelő függvények – OTP 18 V7.3 lists modul (4)

- `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 `Lst` elemeinek `Fun` szerinti redukálása balról jobbra, illetve jobbról balra haladva, az `Acc` akkumulátor használatával.

Részletek és továbbiak: Erlang OTP 18 V7.3 / Basic / stdlib / lists.

<http://erlang.org/documentation/doc-7.3/doc/>

Néhány további könyvtári modul és függvény (OTP 18 V7.3)

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

- **Formázójelek (io modul)**

<code>~a</code>	<code>~jel</code>	<code>~c</code>	az adott kódú karakter	<code>~n</code>	újsor
<code>~s</code>	fűzér	<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		

```
1> io:format("~s ~b ~c ~f~n", [[a,b,c],a,b,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
```

Részletek és továbbiak: Erlang OTP 18 V7.3 / Basic / stdlib.

<http://erlang.org/documentation/doc-7.3/doc/>

Tartalom

4

Erlang alapok

- Bevezetés
- Típusok
- Az Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabb rendű függvények, függvényérték
- Műveletek, beépített függvények
- Örök
- Válogatás a könyvtári függvényekből (OTP 18 V7.3 STDLIB)
- **Kivételkezelés**
- Típusspecifikáció
- Keresési feladat pontos megoldása funkcionális megközelítésben
- Listák használata: futamok
- Prolog és Erlang összehasonlítása
- Rekurzív adatstruktúrák

Kivételkezelés

- Az Erlangban háromféle *kivétel* van: `throw`, `exit` és `error`.
- Kivétel jelzése szintén háromféle módon lehetséges
 - `throw(Why)`
Olyan hiba jelzésére szolgál, amelynek a kezelése elvárható az alkalmazástól.
De használják egy mély hívásból való egyszerű visszatérésre is.
 - `exit(Why)`
A futó processz befejezésére.
 - `error(Why)`
Rendszerhiba jelzésére, amelynek komplex a kezelése.
- Kivétel elkapása kétféleképpen lehetséges
 - `try ... catch` kifejezéssel
 - `catch` kifejezéssel
 - visszaadja a keletkező kivétel termjét, vagy ha nincs hiba, a kiértékelt kifejezést
 - hibalokalizáláshoz hasznos, segít felderíteni a kivétel okát

Kivételkezelés: `try ... catch`

```
try Kifejezés [of
  Minta1 [when ŐrSz1] -> Kif1;
  ...
  Mintan [when ŐrSzn] -> Kifn]
catch
  ExFajta1: ExMinta1 [when ExŐrSz1] -> ExKif1;
  ...
  ExFajtan: 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, ha van
- A `try` szerkezet speciális esete a `case`, amelyben nincs kivételkezelés

Példák try ... catch és catch használatára (1)

kiv.erl

```
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}]].
[{1,'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]]]
```

Példák try ... catch és catch használatára (2)

kiv.erl – folytatás

```
% 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
```

```
9> lists:last([a,b,c]).
c
10> lists:last([]).
** exception error: no function clause matching lists:last([])
11> catch lists:last([]).
{'EXIT',{function_clause,[...% stack trace]}}
12> kiv:safe_apply(fun lists:last/1, [a,b,c]).
{ok,c}
13> kiv:safe_apply(fun lists:last/1, []).
error
```

Tartalom

4

Erlang alapok

- Bevezetés
- Típusok
- Az Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabb rendű függvények, függvényérték
- Műveletek, beépített függvények
- Örök
- Válogatás a könyvtári függvényekből (OTP 18 V7.3 STDLIB)
- Kivételkezelés
- **Típusspecifikáció**
- Keresési feladat pontos megoldása funkcionális megközelítésben
- Listák használata: futamok
- Prolog és Erlang összehasonlítása
- Rekurzív adatstruktúrák

Típuspecifikáció

- Régebben: *dokumentációs konvenció*, nem nyelvi elem az Erlangban
 - Az EDoc értelmezte, ennek alapján generált dokumentációt
 - 2017-ig ezt tanítottuk
- Újabban: a nyelv része
 - Kicsit más a szintaxisa, mint amit korábban tanítottunk, ezért a dokumentumokban előfordulhat EDoc szintaxisú típusspecifikáció is
 - Mi most az új típusspecifikációt tanuljuk (egyszerűsítve)
 - Ehhez van program a típusspecifikáció ellenőrzésére és a programkóddal való összevetésére: `dialyzer`. Nagyon hasznos!
 - Ehhez van program a típusspecifikáció automatikus előállítására: `typer`. Komoly segítség!
- A `typeName` típust így jelöljük: `typeName()`.
- Típusok: előre definiált és felhasználó által definiált

Lásd: http://erlang.org/doc/reference_manual/typespec.html
<https://learnyousomeerlang.com/dialyzer>

Előre definiált típusok

- `any()`, `term()`: bármely Erlang-típus
- `atom()`, `binary()`, `float()`, `fun()`, `function()`, `integer()`, `pid()`, `port()`, `reference()`: Erlang-alaptípusok
- `number()`: `integer()` | `float()`⁷
- `boolean()`: a `false` és a `true` atomok típusa
- `char()`: az `integer()` típus karaktereket ábrázoló része
- `iolist()` = `[char() | binary() | iolist()]`: karakter-io
- `tuple()`: ennestípus
- `list(Type)`: az `[Type]` listatípus szinonimája
- `nil()`: az `[]` üreslista-típus szinonimája
- `string()`: a `list(char())` szinonimája
- `deep_string()` = `[char() | deep_string()]`
- `none()`: a „nincs típusa” típus; nem befejeződő függvény „eredményének” megjelölésére

⁷... | ... választási lehetőség a szintaktikai leírásokban.

Új (felhasználó által definiált) típusok

- Szintaxis: `-type newType() :: Típuskifejezés.`
- Típuskifejezés:
term, előre definiált típus, felhasználó által definiált típus, típusváltozó
- Uniótípus
`T1|T2` típuskifejezés, ha `T1` és `T2` típuskifejezések
`-type nyelv() :: cekla | prolog | erlang.`
- Listatípus
`[T]` típuskifejezés, ha `T` típuskifejezés
`-type nyelvlista() :: [nyelv()].`
Alternatív jelölés: `-type nyelvlista() :: list(nyelv()).`
- Ennestípus
`{T1,...,Tn}` típuskifejezés, ha `T1, ..., Tn` típuskifejezések
`-type diak() :: atom().`
`-type munka() :: atom().`
`-type teljesites() :: {diak(), [{munka(), nyelvlista()}]}.`
- Függvénytípus
`fun(T1,...,Tn) -> T` típuskifejezés, ha `T1, ..., Tn` és `T` típuskifejezések

Függvény-típus 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
- Paraméter- vagy argumentumváltozó: a típus-specifikáció elemeinek nevet adhatunk, pl.

```
-spec safe_last(Xs :: [any()]) -> {ok, X :: any()} | error.
```

```
% X az Xs lista utolsó eleme.
```

```
-spec split(N :: integer(), List :: [any()]) ->  
    {Prefix :: [any()], Suffix :: [any()]}
```

Példák függvény-típus specifikálására (1)

```
-spec id_1(X) -> X. % az X itt: típusváltozó
% Az argumentum azonos az eredménnyel, típusuk tetszőleges.
-spec id_2(tuple()) -> tuple().
% Az argumentum és az eredmény azonos típusú, értékük különbözhet.
-spec id_3(X::tuple()) -> X::tuple(). % X: argumentumváltozó
% Az argumentum és az eredmény azonos típusú, értékük azonos.

-spec file:open(FileName, _Mode) -> {ok, _Handle} | {error, Why}.
% A szingli típusváltozót az Erlang jelzi, '_'-sal elkerülhető.
-spec file:read_line(Handle) -> {ok, _Line} | eof.
% A típus-specifikációban term is megadható, pl. atom.

-spec lists:map_1(fun((A) -> B), [A]) -> [B].
% A típus-specifikációban lehet függvényérték is, pl. fun((A) -> B).
-spec lists:filter(fun((X) -> boolean()), [X]) -> [X].
% Típusváltozó és típuskifejezés változtatva is használható.
-spec map_2(fun((A::any()) -> B::any()), As::[any()]) ->
    Bs::[any()].
% Legkifejezőbb az argumentumváltozó és a típuskifejezés egyidejű használata.
```

Példák függvény-típus specifikálására (2)

% A 2018. évi nagy házi feladatból

```
-type code() :: [integer()]. % [integer()] ≡ list(integer())
```

% code() egy integer() típusú értékekből álló lista típusa.

```
-type blacks() :: integer().
```

```
-type whites() :: integer().
```

% Mindkettő az integer() beépített típus szinonimája.

% A szinonima bevezetésének célja az érthetőség növelése.

```
-type answer() :: {blacks(),whites()}.
```

% answer() a blacks()és whites() típusú értékekből álló pár típusa.

```
-type hint()    :: {code(),answer()}.
```

% hints() a codes() és answer() típusú értékekből álló pár típusa.

```
-spec mmind:mmind(Max::integer(), Hints::[hint()]) -> Codes::[code()].
```

% mmind/2 első argumentuma egy egész, második argumentuma egy hint()

% típusú értékekből álló lista, eredménye code() típusú értékek listája.

A paraméterváltozók arra valók, hogy a fejkommentben – lehetőleg deklaratív módon – leírjuk az argumentum(ok) és eredmény közötti összefüggést.

Tartalom

4

Erlang alapok

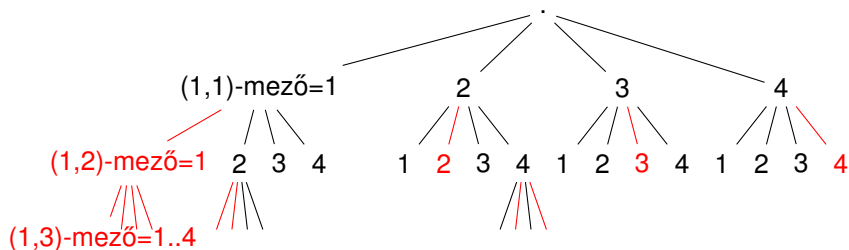
- Bevezetés
- Típusok
- Az Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabb rendű függvények, függvényérték
- Műveletek, beépített függvények
- Örök
- Válogatás a könyvtári függvényekből (OTP 18 V7.3 STDLIB)
- Kivételkezelés
- Típus-specifikáció
- **Keresési feladat pontos megoldása funkcionális megközelítésben**
- Listák használata: futamok
- Prolog és Erlang összehasonlítása
- Rekurzív adatstruktúrák

Pontos megoldás (Exact solution)

- Kombinatorikában sokszor *optimális megoldás* (optimal solution) a neve
 - nem közelítő (approximáció)
 - nem szuboptimális (bizonyos heurisztikák alkalmazásával előállított)
- Keresési feladat: valamilyen *értelmezési tartomány* azon elemeit keressük, melyek megfelelnek az előírt *feltételeknek*
 - lehetséges megoldás = *jelölt* (candidate)
 - értelmezési tartomány = *keresési tér* (search space), jelöltek halmaza
 - feltételek = *korlátok* vagy *kényszerek* (constraints)
- Példák: 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. egy 16 mezős Sudoku (1. sor, 1. oszlop) mezejének értéke lehet 1,2,3,4;
az (1. sor, 2. oszlop) mezejének értéke szintén lehet 1,2,3,4 stb.



- Bizonyos esetekben (**pirossal** jelöljük) tudjuk, hogy nem lehet megoldás (**egy sorban ugyanaz az érték több mezőben nem fordulhat elő**)
- Hatékony megoldás: a keresési fa egyes 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

$$\begin{array}{r} S E N D \\ + M O R E \\ \hline M O N E Y \end{array}$$

a papíron történő összeadás szabályai szerint, vagyis

$$\begin{aligned} (1000S + 100E + 10N + D) + (1000M + 1000 + 10R + E) = \\ = 10000M + 10000 + 100N + 10E + Y. \end{aligned}$$

- Naív megoldásunk: járjuk be a teljes keresési teret, és szűrjük azokra a nyolcasokra, amelyekre teljesülnek a feltételek
- Keresési tér $\subseteq \{0, 1, \dots, 9\}^8$, azaz egy 8-elemű Descartes-szorzat, mérete 10^8 (tizedrendű nyolcadosztályú ismétléses variáció)
- 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

`sendmory.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()) -> boolean().
```

```
% 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

sendmory.erl – folytatás

```
-spec all_different(Xs::[any()]) -> B::boolean()
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()].`

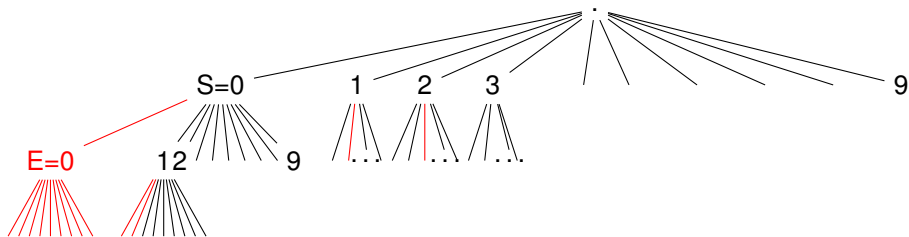
`smm1() ->`

```

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 azokat *a 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 a jelöltek **tesztelését előrébb hoztuk**
- Vegyük észre, hogy a **keresési tér csökkentésével** is ide juthatunk: új keresési tér $\subseteq \{10 \text{ elem nyolcadadosztályú ismétlés nélküli variációi}\}$
- 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()].
```

```
% Ellenőrzés csak a generálás után!
```

```
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

- Érdeemes-e a jelöltek generálását elválasztani az ellenőrzéstől? **Nem!**

`sendmory.erl` – folytatás

```
-spec invars() -> [octet()].
```

```
% Számjegyek ismétlés nélküli nyolcadosztályú 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)].
```

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], % 0 kizárva
```

```
    E <- Ds -- [S],
```

```
    N <- Ds -- [S,E],
```

```
    D <- Ds -- [S,E,N],
```

```
    M <- Ds -- [0,S,E,N,D], % 0 kizárva
```

```
    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

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

sendmory.erl – folytatás

```

smm5() ->                                %%      S E N D
Ds = lists:seq(0, 9),                    %%      + M O R E
[ {S,E,N,D,M,O,R,Y} ||                  %%      = M O N E 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})].

```

Futási eredmények (Intel x86/64 i5-3210M CPU @ 2.50GHz)

Az eddig kidolgozott megoldások jellemzése és a futási eredmények

- `smm0` – kimerítő keresés
- `smm0e` – mint `smm0`, de kis trükkal (`S <- [9,8,7,...]`)
- `smm1` – keresési fa redukálása egyezések szűrésével generálás közben
- `smm2` – keresési tér redukálása a változók tartományának szűkítésével
- `smm3` – mint `smm2`, de ellenőrzéssel generálás közben
- `smm4` – mint `smm2`, a változók tartományának további szűkítésével
- `smm5` – számok építése hátulról, részösszegek ellenőrzése generálás közben

%			Number of candidates with approx. length				
% Id	SEND+MORE=MONEY	Time	1	2	3	4	5
% 0 :	9567+1085=10652	82660 ms	0	0	0	0	100000000 candS
% 0e:	9567+1085=10652	5320 ms	0	0	0	0	5671083 candS
% 1 :	9567+1085=10652	2060 ms	0	0	0	0	1814400 candS
% 2 :	9567+1085=10652	1660 ms	0	0	0	0	1814400 candS
% 3 :	9567+1085=10652	3310 ms	0	0	0	0	1814400 candS
% 4 :	9567+1085=10652	1510 ms	0	0	0	0	1451520 candS
% 5 :	9567+1085=10652	20 ms	720	3024	1450	1536	1536 candS

Vágások a keresési fában generálás közben (2)

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

`sendmory.erl` – folytatás

```
-type partial_solution() ::  
    {SendList::[d()], MoreList::[d()], MoneyList::[d()]}.  
  
-spec smm6() -> [octet()].  
smm6() ->  
    smm6({[], [], []}, 5, lists:seq(0,9)).
```

- `{[], [], []}` a kiindulási részmegoldásunk (`PartialSolution`)
- Ötjegyű számokat kell építeni, ezért 5 a második argumentum (`Num`)
- `lists:seq(0,9)` a változók tartománya (`Domain`)

Vágások a keresési fában generálás közben (3)

- Egy `PartialSolution = {SendList, MoreList, MoneyList}` részmegoldás csak akkor bővíthető megoldássá, ha
 - a listákban a számjegyeket jelentő változók jó pozícióban vannak;
 - a részösszeg is helyes, csak az átvitelben lehet eltérés.

`sendmory.erl` – folytatás

```
-spec check_equals(partial_solution()) -> boolean().
```

```
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 (4)

- Egy `PartialSolution = {SendList, MoreList, MoneyList}` részmegoldás csak akkor bővíthető megoldássá, ha
 - a listákban a számjegyek jó pozícióban vannak: az azonos betűk egyeznek, a többi számjegy különbözik;
 - a részösszeg is helyes, csak az átvitelben lehet eltérés.

`sendmory.erl` – folytatás

```
-spec check_partialsom(partial_solution()) -> boolean().  
% Ellenőrzi, hogy aritmetikailag helyes-e a részmegoldás.  
% Az átvittel (carry) nem foglalkozik, mert pl.  
% {[1,2],[3,4],[4,6]} és {[9],[2],[1]} egyformán helyes,  
% ui. 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.
```

Vágások a keresési fában generálás közben (5)

sendmory.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 változóhoz értéket kötni: PSol1 <- [ Érték ],
    check_equals(PSol1),
    check_partialsun(PSol1),
    Solution <- smm6(PSol1, Num, Domain) ].
```

Vágások a keresési fában generálás közben (6)

sendmory.erl – folytatás

```
-spec smm7() -> [octet()].
```

% Hátulról építi a számokat, ellenőrzi a részösszegeket.

```
smm7() ->
```

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

```
   {[O,S,E,N,D], [O,M,O,R,E], [M,O,N,E,Y]} <-
```

```
     smm7(5, lists:seq(0,9))].
```

- Az `smm7` program lényegileg megegyezik `smm6`-tal, de tisztább a szerkezete
- Az `smm7/2` függvény állítja elő a megoldások listáját, ebből szedi ki `smm7/0` függvény a megoldást jelentő nyolcasokat
- Most is ötjegyű számokat építünk, ez `smm7/2` első argumentuma
- A számjegyek tartománya: 0...9, ez `smm7/2` második argumentuma

Vágások a keresési fában generálás közben (7)

sendmory.erl – folytatás

```
-spec smm7(Num::integer(), Domain::[d()]) -> PS::[partial_solution()].
% Visszaadja a Num (= Send hossza) méretű részmegoldások listáját,
% Domain a számjegykészlet.
smm7(0, _) ->
    [{[], [], []}];
smm7(N, Domain) ->
    [ PartialSolution ||
      {Send, More, Money} <- smm7(N-1, Domain),
      Dsend <- Domain,
      Dmore <- Domain,
      Dmoney <- Domain,
      begin
          PartialSolution = {[Dsend|Send], [Dmore|More], [Dmoney|Money]},
          true
      end,
      check_equals(PartialSolution),
      check_partialsom(PartialSolution)
    ].
```

Hol és hogyan lehetne csökkenteni az `smm7/2` függvényben a keresési teret?

Vágások a keresési fában generálás közben (8)

sendmory.erl – folytatás

```
-spec smm7(Num::integer(), Domain::[d()]) -> PS::[partial_solution()].
% Visszaadja a Num (= Send hossza) méretű rész megoldások listáját,
% Domain a számjegyek készlete.
smm7(0, _) ->
    [{[], [], []}];
smm7(N, Domain) ->
    [ PartialSolution ||
      {Send, More, Money} <- smm7(N-1, Domain),
      Dsend <- Domain,
      Dmore <- Domain,
      Dmoney <- Domain, % <-- ITT
    begin
      PartialSolution = {[Dsend|Send], [Dmore|More], [Dmoney|Money]},
      true
    end,
    check_equals(PartialSolution),
    check_partialsun(PartialSolution)
  ].
```

Vágások a keresési fában generálás közben (9)

sendmory.erl – folytatás

```
-spec smm7a(Num::integer(), Domain::[d()]) -> PS::[partial_solution()].
smm7a(0, _) ->
    [{[], [], []}];
smm7a(N, Domain) ->
    [PartialSolution ||
     {Send, More, Money} <- smm7a(N-1, Domain),
     Dsend <- Domain,
     Dmore <- Domain,
     Dmoney <- [ begin Carry = if Send == [] -> 0 ;
                  true -> (hd(Send) + hd(More)) div 10
                end,
                 (Dsend + Dmore + Carry) rem 10
             end
     ],
     begin PartialSolution = {[Dsend|Send], [Dmore|More], [Dmoney|Money]},
           true end,
     check_equals(PartialSolution),
     check_partialsun(PartialSolution)
    ].
```

Vágások a keresési fában generálás közben (10)

```

smm7a(0, _)    ->
...
    % Send, More és Money az éppen bővíteni
{Send,More,Money} <- smm7a(N-1, Domain), % kívánt rész megoldás
Dsend <- Domain,
Dmore <- Domain,    % Dsend, Dmore, Dmoney az új számjegyek,
Dmoney <-           % Domain a számjegyek teljes tartománya
  [ begin
    Carry =
      if Send == [] -> % Ha Send üres, More is üres,
        0 ;           % nincs átvitel (Carry = 0)
      true ->         % Egyébként Send és More fejének összegétől
        (hd(Send) + hd(More)) div 10 % függően Carry=0 vagy =1
    end,
    (Dsend + Dmore + Carry) rem 10 % Dmore pontos értékét ez
  end ],           % a három szám határozza meg.
begin PartialSolution = {[Dsend|Send],...}, true end,
check_equals(PartialSolution) % ,
check_partialsom(PartialSolution) % feleslegessé vált, törölhető
].

```

Jelentős a nyereség: Dmoney méretét 10-ről 1-re csökkentettük!

Futási eredmények (Intel x86/64 i5-3210M CPU @ 2.50GHz)

Az újabb megoldások jellemzése és a futási eredmények

- `smm5` – számok építése hátulról, részösszegek ellenőrzése generálás közben (ez a megoldás nagyon hatékonynak bizonyult, ezért az összehasonlításhoz újra szerepeltetjük)
- `smm6` – rész megoldások építése hátulról, ellenőrzés után az ígéretes rész megoldások bővítése
- `smm7` – mint `smm6`, de átláthatóbb kóddal
- `smm7a` – mint `smm7`, de az első két tartományból (`Dsend`, `Dmore`) számítja ki a harmadik tartományt (`Dmoney`), és ezzel `Dmoney` 10-edére csökken

%			Number of candidates with approx. length					
% Id	SEND+MORE=MONEY	Time	1	2	3	4	5	
% 5	: 9567+1085=10652	20 ms	720	3024	1450	1536	1536 candS	
% 6	: 9567+1085=10652	170 ms	720	3024	1450	2196	148 candS	
% 7	: 9567+1085=10652	150 ms	720	3024	1450	2196	148 candS	
% 7a	: 9567+1085=10652	30 ms	72	290	183	196	1 candS	

Korlátkielégítési probléma (Constraint satisfaction problem, CSP)

- 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, bontuk 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:

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

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:
 $0 \in [0]$, $c_4 \in [0,1]$, $m \in [1,2,3,4,5,6,7,8,9]$
Az $m \in [2,3,4,5,6,7,8,9]$ értékek feleslegesek!
- Felesleges érték elhagyásával (szűkítéssel) ekvivalens CSP-t kapunk
- SMM kezdeti tartománya; és megszűkítve, tovább már nem szűkíthető:

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

szűkítés az összes
lehetséges
korláttal, ameddig
sikerül:

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

CSP tevékenységek – címkézés (labeling)

- Tovább már nem szűkíthető CSP esetén vizsgáljuk a többértelműséget
- Többértelműség: van olyan tartomány, amely legalább két elemet tartalmaz, és egyetlen 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	és	c4: 1
s: 89	c1>0	s: 89		s: 89
e: 0123456789	esetek:	e: 0123456789		e: 0123456789
...	

- 3 az egyes választásokat – mint új CSP-eket – mind megoldjuk.

CSP tevékenységek – visszalépés

- Ha nincs többértelműség, 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-megoldásának 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

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.

```

Futási eredmények (Intel x86/64 i5-3210M CPU @ 2.50GHz)

A hatékony megoldások jellemzése és a futási eredmények

- `smm5` – számok építése hátulról, részösszegek ellenőrzése generálás közben
- `smm7a` – részmegoldások építése hátulról, ellenőrzés után az ígéretes részmegoldások bővítése, az első két tartományból (`Dsend`, `Dmore`) a harmadik tartomány (`Dmoney`) kiszámítása, és ezzel `Dmoney` 10-edére csökkentése
- CSP-alapú megoldás (CSP = Constraint Satisfaction Problem, korlátkielégítési probléma)

%			Number of candidates with approx. length				
% Id	SEND+MORE=MONEY	Time	1	2	3	4	5
% 5 :	9567+1085=10652	20 ms	720	3024	1450	1536	1536 candS
% 7a:	9567+1085=10652	30 ms	72	290	183	196	1 candS
% 99:	9567+1085=10652	20 ms	0	0	0	0	21 candS

Tartalom

4

Erlang alapok

- Bevezetés
- Típusok
- Az Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabb rendű függvények, függvényérték
- Műveletek, beépített függvények
- Örök
- Válogatás a könyvtári függvényekből (OTP 18 V7.3 STDLIB)
- Kivételkezelés
- Típus-specifikáció
- Keresési feladat pontos megoldása funkcionális megközelítésben
- **Listák használata: futamok**
- Prolog és Erlang összehasonlítása
- Rekurzív adatstruktúrák

Futam definiálása

- *Futam*: olyan nem üres lista, amelynek szomszédos elemei valamilyen feltételnek megfelelnek
- A feltételt az előző és az aktuális elemre alkalmazandó *predikátumként* adjuk át a futamot előállító függvénynek
- *Predikátum*: logikai (igaz/hamis) értéket eredményül adó függvény.
- Példa:

```
1> P = fun erlang:'<' /2.  
#Fun<erlang.<.2>  
2> P(1, 2).  
true
```

- Feladat: írjunk olyan Erlang-függvényt, amely egy lista egymás utáni elemeiből képzett diszjunkt, tovább nem bővíthető futamok listáját adja eredményül – az elemek eredeti sorrendjének megőrzésével
- Az első, naív változatban egy segédfüggvényt írunk egy lista *első futamának* (prefixumának), valamint egy másikat a *maradéklistának* az előállítására (vö. `lists:splitwith/2`)

Futamok előállítására – naív változat

- Példa (ahol $P = \text{fun erlang:'</2):$

```
4> futam:also_futam(P, [1,3,9,5,7,2,5,9,1,6,0,0,3,5,6,2]).
[1,3,9]
```

futam.erl – Futamok felsorolása

```
-type elem() :: any().
```

```
-type pred() :: fun((elem(), elem()) -> bool()).
```

```
-spec also_futam(P::pred(), Ls::[elem()]) -> Fs::[elem()].
% Fs az Ls P-t kielégítő első, tovább nem bővíthető futama
% (prefixuma).
```

```
also_futam(_P, [X]) ->
[X];
```

```
also_futam(P, [X|Ys=[Y|_]]) -> % Ys=[Y|_]: réteges minta
  case P(X, Y) of
    false -> [X];
    true  -> [X|also_futam(P, Ys)]
  end.
```

Futamok előállítás – naív változat (2)

- Példa:

```
4> futam:elso_futam(P, [1,3,9,5,7,2,5,9,1,6,0,0,3,5,6,2]).
[1,3,9]
5> futam:maradek(P, [1,3,9,5,7,2,5,9,1,6,0,0,3,5,6,2]).
[5,7,2,5,9,1,6,0,0,3,5,6,2]
```

futam.erl – folytatás

```
-spec maradek(P::pred(), Ls::[elem()]) -> Ms::[elem()].
% Ms az Ls P-t kielégítő első futama utáni maradéka.
maradek(_P, [_X]) ->
    [];
maradek(P, [X|Ys=[Y|_]]) ->
    case P(X, Y) of
        false -> Ys;
        true  -> maradek(P, Ys)
    end.
```

Futamok előállítására – naív változat (3)

- Példa:

```
6> futam:naiv_futamok(P, [1,3,9,5,7,2,5,9,1,6,0,0,3,5,6,2]).
[[1,3,9],[5,7],[2,5,9],[1,6],[0],[0,3,5,6],[2]]
7> futam:naiv_futamok(P, []).
[]
8> futam:naiv_futamok(P, [1]).
[[1]]
```

futam.erl – folytatás

```
-spec naiv_futamok(Pred::pred(), Ls::[elem()]) -> Lss:[[elem()]].
% Lss az Ls szomszédos, Pred-et kielégítő elemeiből álló, tovább
% nem bővíthető, diszjunkt futamok listája.
naiv_futamok(_P, []) -> [];
naiv_futamok(P, Ls) -> Fs = elso_futam(P, Ls),
                       Ms = maradek(P, Ls),
                       [ Fs | naiv_futamok(P, Ms) ].
```


Futamok előállítás – hatékonyabb változat

- Pazarlás kétszer megkeresni az első futamot, lásd előző példák:

```
4> futam:elso_futam(P, [1,3,9,5,7,2,5,9,1,6,0,0,3,5,6,2]).
[1,3,9]
```

```
5> futam:maradek(P, [1,3,9,5,7,2,5,9,1,6,0,0,3,5,6,2]).
[5,7,2,5,9,1,6,0,0,3,5,6,2]
```

- Kezeljük az első futamot és a maradékot *egyetlen párként*:

```
9> futam:futam_maradek(P, [1,3,9,5,7,2,5,9,1,6,0,0,3,5,6,2]).
{[1,3,9],[5,7,2,5,9,1,6,0,0,3,5,6,2]}
```

```
-spec futam_maradek(P::pred(), L::[elem()]) ->
    {Fs::[elem()], Ms::[elem()]}.
% Fs ::= elso_futam(P, L) és Ms ::= maradek(P, L).
```

```
futam_maradek(_P, [X]) -> {[X], []};
```

```
futam_maradek(P, [X|Ys=[Y|_]]) ->
    case P(X, Y) of
        true -> {Fs, Ms} = futam_maradek(P, Ys),
                {[X|Fs], Ms};
        false -> {[X], Ys}
    end.
```

Futamok előállítás – hatékonyabb változat (2)

futam.erl – folytatás

```
-spec *futamok(Pred::pred(), Ls::[elem()]) -> Lss::[[elem()]].
```

```
naiv_futamok(_P, []) ->
  [];
naiv_futamok(P, Ls) ->
  Fs = elso_futam(P, Ls),
  Ms = maradek(P, Ls),
  [Fs|naiv_futamok(P, Ms)].
```

```
futamok(_P, []) ->
  [];
futamok(P, Ls) ->
  {Fs, Ms} = futam_maradek(P, Ls),
  [Fs|futamok(P, Ms)].
```

- Példa futam_maradek felhasználására: számtani sorozatok gyűjtése

```
10> futam:difek([1,3,5,7,7,5,3,1,1,1,1,2]).
[[1,3,5,7],[7,5,3,1],[1,1,1],[2]]
```

```
-spec difek(Xs::[number()]) -> Dss::[[number()]].
```

% Dss az Xs számtani sorozatot alkotó részlistáinak listája.

```
difek([X1,X2|_]=Xs) ->
  {Fs,Ms} = futam_maradek(fun(A, B) -> B-A == X2-X1 end, Xs),
  [Fs|difek(Ms)];
difek([_]=Xs) -> [Xs];
difek([]) -> [].
```

Tartalom

4

Erlang alapok

- Bevezetés
- Típusok
- Az Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabb rendű függvények, függvényérték
- Műveletek, beépített függvények
- Örök
- Válogatás a könyvtári függvényekből (OTP 18 V7.3 STDLIB)
- Kivételkezelés
- Típusspecifikáció
- Keresési feladat pontos megoldása funkcionális megközelítésben
- Listák használata: futamok
- **Prolog és Erlang összehasonlítása**
- Rekurzív adatstruktúrák

Prolog és Erlang: néhány eltérés

Prolog	Erlang
predikátum, kétféle érték	függvény, értéke tetszőleges típusú
siker esetén változóbehelyettesítés	csak bemenő argumentum és visszatérési érték van
választási pontok, több megoldás	determinizmus, egyetlen megoldás
C_1, \dots, C_n célsorozat, redukció + visszalépés	S_1, \dots, S_n szekvenciális kifejezés, értéke S_n
összetett kifejezés (struktúra), a lista is az	ennes és lista típus (tuple, list)
operátor definiálása	-
egyesítés szimmetrikus	jobb oldalon tömör kifejezés; bal oldalon mintakifejezés, őrkifejezéssel

Prolog és Erlang: 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 (fűzér), atom

Tartalom

4

Erlang alapok

- Bevezetés
- Típusok
- Az Erlang szintaxis alapjai
- Mintaillesztés
- Listanézet
- Magasabb rendű függvények, függvényérték
- Műveletek, beépített függvények
- Örök
- Válogatás a könyvtári függvényekből (OTP 18 V7.3 STDLIB)
- Kivételkezelés
- Típus-specifikáció
- Keresési feladat pontos megoldása funkcionális megközelítésben
- Listák használata: futamok
- Prolog és Erlang összehasonlítása
- Rekurzív adatstruktúrák

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, utoljára berakott elem

`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}) -> S.
```

```
top(empty) -> error;  
top({X,_S}) -> X.
```

Kis példák verem használatára

```
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. megfordítunk 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 – Bináris fa

- Műveletek bináris fákon: fa létrehozása, mélysége, leveleinek száma

tree.erl

```
-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.
```

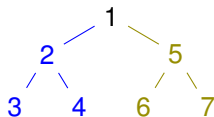
```
depth(leaf)           -> 0;           % Fa legnagyobb mélysége.  
depth({_ ,Lt,Rt}) -> 1 + erlang:max(depth(Lt), depth(Rt)).
```

```
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

Kiegészítő anyag

```

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
```

V. rész

Haladó Prolog

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

- 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 technikákbemutatása, amelyekkel
 - hatékony Prolog programok készíthetők,
 - esetleg a tiszta logikán túlmutató eszközök alkalmazásával.

Haladó Prolog – tartalomjegyzék

- Meta-logikai eljárások
- Megoldásgyűjtő eljárások
- A keresési tér szűkítése
- Determinizmus és indexelés
- Jobbrekurzió, akkumulátorok
- Imperatív programok átírása Prologba

Tartalom

- 5 Haladó Prolog
 - Meta-logikai eljárások
 - Megoldásgyűjtő beépített eljárások
 - A keresési tér szűkítése
 - Determinizmus és indexelés
 - Jobbrekurzió és akkumulátorok
 - Imperatív programok átírása Prologba
 - Vezérlési eljárások
 - Magasabbrendű eljárások

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

- A Prolog kifejezések pillanatnyi behelyettesítettségi állapotát vizsgáló eljárások (értelemszerűen logikailag nem tiszták):
 - kifejezések osztályozása (1)
 - | ?- var(X) /* X változó? */, X = 1. \implies X = 1
 - | ?- X = 1, var(X). \implies no
 - kifejezések rendezése (4)
 - | ?- X @< 3 /* X megelőzi 3-t? */, X = 4. \implies X = 4
 - % a változók megelőzik a nem változó kifejezéseket*
 - | ?- X = 4, X @< 3. \implies no
- Prolog kifejezéseket szétszedő vagy összerakó eljárások:
 - (struktúra) kifejezés \iff név és argumentumok (2)
 - | ?- X = f(alma,körte), X =.. L \implies L = [f,alma,körte]
 - névkonstansok és számok \iff karaktereik (3)
 - | ?- atom_codes(A, [0'a,0'b,0'a]) \implies A = aba

Meta-predikátumok – motiváló példa

- Formula (Form): az 'x' atom; szám; Form1 + Form2; Form1 * Form2
- Számoljuk ki egy formula értékét egy adott x behelyettesítés mellett!

% value_of(+Form, +XE, ?E): az x=XE helyettesítéssel Form értéke E.

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

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

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

```
value_of0(Frm, X, V) :-
    Frm = *(P1,P2),
    value_of0(P1, X, V1),
    value_of0(P2, X, V2),
    FrmV = *(V1,V2),
    V is FrmV.
```

```
| ?- value_of((x+1)*3-(x-1), 1, V).
```

% Nem kellene a zárójeles (kif) feldolgozásával foglalkozni?

```
| ?- value_of(exp(100,min(x,1/x)), 2, V).
```

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

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

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

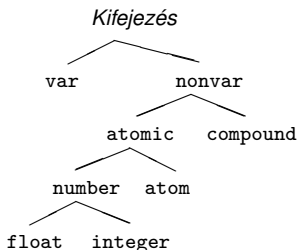
```
⇒ V = 6 ? ; no
```

```
⇒ V = 10.0 ? ; no
```

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

Kifejezések osztályozása

- Kifejezésfajták – osztályozó beépített eljárások (ismétlés)



<code>var(X)</code>	X változó
<code>nonvar(X)</code>	X nem változó
<code>atomic(X)</code>	X konstans
<code>compound(X)</code>	X struktúra
<code>atom(X)</code>	X atom
<code>number(X)</code>	X szám
<code>integer(X)</code>	X egész szám
<code>float(X)</code>	X lebegőpontos szám

- 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)

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]]

Bev. példa – Adott értékű kifejezések előállítása

- Adott számokból megadott műveletek (pl. +, -, *, /) segítségével építsünk egy megadott értékű aritmetikai kifejezést!
(Feltételezhető, hogy az adott számok mind különböznek.)
 - A számok nem „tapaszthatók” össze hosszabb számokká
 - Mindegyik adott számot pontosan egyszer kell felhasználni, sorrendjük tetszőleges lehet
 - Nem minden alpműveletet kell felhasználni, egyfajta alpművelet többször is előfordulhat
 - Zárójelek tetszőlegesen használhatók
- Példák a fenti szabályoknak megfelelő, az 1, 3, 4, 6 számokból felépített aritmetikai kifejezésekre: $1 + 6 * (3 + 4)$, $(1 + 3)/4 + 6$
- Viszonylag nehéz megtalálni egy olyan aritmetikai kifejezést, amely az 1, 3, 4, 6 számokból áll, a négy alpműveletet használja és értéke 24

Adott értékű kifejezések előállítás – megoldás univ-val

Írjunk egy eljárást az alábbi fejkommentnek megfelelően:

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

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

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

Struktúrák 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]

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)
```


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 -adik argumentumát választja ki.
(Az [] kiválasztó Kif -ből Kif -et választja ki.)
- Pl. $a*b+f(5,8,7)/c$ -ben b kiválasztója [1,2], 7 kiválasztója [2,1,3].

% kif_szám(?Kif, ?N, ?Kiv): Kif Kiv kiválasztójú része az N szám.

```
kif_szám(X, X, []) :-
```

```
    number(X).
```

```
kif_szám(X, N, [I|Kiv]) :-
```

```
    compound(X), % 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).  $\implies$  K = [1], N = 1 ? ;
```

```
      K = [2,2,1], N = 2 ? ; no
```

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), % append/2, lásd library(lists)
    atom_codes(Rész, Ds).
```

| ?- dadogó_rész(babaruhaha, R). \implies 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). \implies A = abrakadabra ?

Számok szétszedése és összerakása

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

Kifejezések összehasonlítása – beépített eljárások

- Két tetszőleges kifejezés összehasonlítását végző eljárások:

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

- Az összehasonlítás mindig a belső ábrázolás (kanonikus alak) szerint történik:
 - $| \text{?- } [1, 2, 3, 4] @< \text{struktúra}(1, 2, 3). \implies$ **sikerül (6.1 szabály)**
- Lista rendezése: `sort(+L, ?S)`
 - Jelentése: az L lista @< szerinti rendezése S, ==/2 szerint azonos elemek ismétlődését kiszűrve.
 - $| \text{?- } \text{sort}([a,c,a,b,b,c,c,e,b,d], S).$
 - $S = [a,b,c,d,e] ? ;$
 - no

Összefoglalás: a Prolog egyenlőség-szerű beépített eljárásai

- | | |
|--|---|
| <ul style="list-style-type: none"> • $U = V$: U egyesítendő V-vel.
Soha sem jelez hibát. | <ul style="list-style-type: none"> ?- $X = 1+2.$ \implies $X = 1+2$?- $3 = 1+2.$ \implies no |
| <ul style="list-style-type: none"> • $U == V$: U azonos V-vel.
Soha sem jelez hibát és soha sem helyettesít be. | <ul style="list-style-type: none"> ?- $X == 1+2.$ \implies no ?- $3 == 1+2.$ \implies no ?- $+(1,2) == 1+2 \implies$ yes |
| <ul style="list-style-type: none"> • $U ::= V$: Az U és V aritmetikai kifejezések értéke megegyezik.
Hibát jelez, ha U vagy V nem (tömör) aritmetikai kifejezés. | <ul style="list-style-type: none"> ?- $X ::= 1+2.$ \implies hiba ?- $1+2 ::= X.$ \implies hiba ?- $2+1 ::= 1+2.$ \implies yes ?- $2.0 ::= 1+1.$ \implies yes |
| <ul style="list-style-type: none"> • $U \text{ is } V$: U egyesítendő a V aritmetikai kifejezés értékével.
Hiba, ha V nem (tömör) aritmetikai kifejezés. | <ul style="list-style-type: none"> ?- $2.0 \text{ is } 1+1.$ \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 |
| <ul style="list-style-type: none"> • $(U =.. V$: U „szétszedettje” a V lista) | <ul style="list-style-type: none"> ?- $1+2 =.. X.$ \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.

?- X \= 1+2.	⇒	no
?- +(1,2) \= 1+2.	⇒	no

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

?- X \== 1+2.	⇒	yes
?- 3 \== 1+2.	⇒	yes
?- +(1,2)\==1+2	⇒	no

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

?- X =\= 1+2.	⇒	hiba
?- 1+2 =\= X.	⇒	hiba
?- 2+1 =\= 1+2.	⇒	no
?- 2.0 =\= 1+1.	⇒	no

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

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

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

Tartalom

5

Haladó Prolog

- Meta-logikai eljárások
- **Megoldásgyűjtő beépített eljárások**
- A keresési tér szűkítése
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Imperatív programok átírása Prologba
- Vezérlési eljárások
- Magasabbrendű eljárások

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):

$$\text{Lista} = \{ \text{Gyűjtő } \textit{másolat} \mid (\exists X \dots Z) \text{Cél igaz} \}$$

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

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

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

- Példa az eljárás használatára:

```
gráf([a-c,a-b,b-c,c-e,b-d]).
```

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

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

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: $Lista = \{ Gyűjtő \mid (\exists V_1, \dots, V_n) Cél \text{ igaz} \} \neq []$.
 $| \text{?- gráf}(_G), \text{bagof}(B, A \text{^member}(A-B, _G), VegP).$
 $\implies VegP = [c,b,c,e,d] ? ; no$

- Egymásba ágyazott gyűjtések

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

% A G irányított gráf fokszámlistája FL:

% FL = { A-N | N = |{ V | A-V ∈ G }|, N > 0 }

fokszámai(G, FL) :-

```

    bagof(A-N, Vk^(bagof(V, member(A-V, G), Vk),
                    length(Vk, N)
                    ), FL).

```

| ?- gráf(_G), fokszámai(_G, FL).

$\implies FL = [a-2,b-2,c-1] ? ; no$

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), Vks), length(Vks, N).
```

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

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

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

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

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

- az eljárás végrehajtása:
 - ugyanaz mint: bagof(Gyűjtő, Cél, L0), sort(L0, Lista),
 - itt sort/2 egy univerzális rendező eljárás, 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).
```

```
| ?- gráf(_G), gráf_pontjai(_G, Pk).
```

```
⇒ Pk = [a,b,c,d,e] ? ; no
```

```
| ?- gráf(_G), bagof(P, pontja(P, _G), Pk).
```

```
⇒ Pk = [a,c,a,b,b,c,c,e,b,d] ? ; no
```

Tartalom

5

Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- **A keresési tér szűkítése**
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Imperatív programok átírása Prologba
- Vezérlési eljárások
- Magasabbrendű eljárások

Nyelvi eszközök a keresési tér szűkítésére

- Az első Prolog rendszerektől kezdve: vágó, szabványos jelölése !
- Későbbi kiterjesztés: az (`if -> then ; else`) feltételes szerk.
- Feltételes szerkezet – procedurális szemantika (ismétlés)
A (`felt->akkor;egyébként`), `folyt` célsorozat végrehajtása:
 - Végrehajtjuk a `felt` hívást (egy önálló végrehajtási környezetben).
 - Ha `felt` sikeres \implies „`akkor,folyt`” célsorozattal folytatjuk, a `felt` **első** megoldása által eredményezett behelyettesítésekkel.
A `felt` cél **többi megoldását nem keressük meg!**
 - Ha `felt` meghiúsul \implies „`egyébként,folyt`” célsorozattal folytatjuk.
- A vágó beépített eljárás – procedurális szemantika (ismétlés)
 - mindig sikerül; de mellékhatásként megszünteti a választási pontokat egészen a szülő célig, azt is beleértve. (Egy `C` cél szülője az a cél, amelyet, a `C`-t tartalmazó klóz fejével illesztettünk.)
- A vágó szemléltetése a 4-kapus doboz modellben: a vágó `Fail` kapujából a körülvevő (szülő) doboz `Fail` kapujára megyünk.

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

- Rekurzív megoldás (mérnöki)

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

- Visszalépéses keresés (matematikus), nem hatékony

```
első_poz_elem2(L, EP) :-
    append(NemPozL, [EP|_], L), EP > 0,
    \+ van_poz_eleme(NemPozL).
van_poz_eleme(L) :-
    member(P, L),
    P > 0.
```

- Választás a feltételben (Prolog hekker)

```
első_poz_elem3(L, EP) :-
    ( member(EP, L), EP > 0 -> true ).
```

Az eljárás (+,+) módban hibás: első_poz_elem3([1,2], 2) \implies yes
 (+,+) módban a fenti kód jelentése: P az L lista **egyik** pozitív eleme.

- Kimenő paraméter értékadásának késleltetése (Prolog hekker)

```
első_poz_elem4(L, EP) :-
    ( member(X, L), X > 0 -> EP = X ). % (+,+) módban is jó!
```

A 3.–4. megoldás épít a member/2 felsorolási sorrendjére!

Vágót használó megoldások a példafeladatra

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


```
első_poz_elem5([X|_], X) :- X > 0, !.
```

```
első_poz_elem5([X|L], EP) :- X =< 0, első_poz_elem5(L, EP).
```
 - 2 megszünteti a választási pontokat az előtte levő eljáráshívásokban.


```
első_poz_elem6(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
- Célszerű lehet hatékonysági okból elhagyni a fenti $X \leq 0$ feltételt:


```
első_poz_elem7([X|_], X) :- X > 0, !.
```

```
első_poz_elem7([X|L], EP) :- első_poz_elem7(L, EP).
```
- Milyen színűek a fenti vágók?
Mi a válasz az `első_poz_elem...([1,2], 2)` alakú hívásokra?

További példák a vágó eljárás használatára

Milyen színűek az alábbi vágók?

```
% 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, !,
```

vörös

```
·    pozitívak(Ek, Pk).
```

```
pozitívak(_E|Ek, Pk) :-
```

```
    /* \+ _E > 0, */ pozitívak(Ek, Pk).
```

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

Figyelem: a fenti példák nem tökéletesek, hatékonyabb és/vagy általánosabban használható változatukat később ismertetjük!

A vágás alapszabálya

- Ha a vágó zöld, nincs gond a jelentéssel, de ez többnyire ismételt/felesleges vizsgálatokkal jár
- Ha a vágó vörös, attól a program működhet helyesen is
- Miért nem működik helyesen az `?- első_poz_elem7([1,2], 2)` hívás?

```
első_poz_elem7([X|_], X) :- X > 0, !. (1)
```

```
első_poz_elem7([X|L], EP) :- első_poz_elem7(L, EP).
```

A fejlesztés (1)-gyel nem sikerül! (1) ekvivalens átírása:

```
első_poz_elem7([X|_], EP) :- EP = X, X > 0, !. (2)
```

Az `EP = X` vizsgálat nem feltétele a vágásnak, a vágás utánra való!

- A megoldás a **vágás alapszabálya**:
A kimenő paraméterek értékadását mindig a vágó után végezzük!

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

```
első_poz_elem8([X|L], EP) :- első_poz_elem8(L, EP).
```

- Ez nemcsak általánosabban használható, hanem hatékonyabb kódot is ad: csak akkor helyettesíti be a kimenő paramétert, ha már tudja, hogy pozitív (nincs „előre-behelyettesítés”, mint (1)-ben és (2)-ben
- Az alapszabály betartásakor az indexelés is hatékonyabb lesz

A korábbi 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).
```

- A vágó helyett **a diszjunktív feltételes szerkezet használatát javasoljuk**. (Az első_poz_elem8 és első_poz_elem1 eljárásokból ugyanaz a kód generálódik!)
- Feltételes szerkezet használatakor is fontos, hogy a kimenő paraméterek ne szerepeljenek a feltételben (vö. első_poz_elem3 és első_poz_elem4)

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

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

$\max(X, Y, X) :- X \geq Y.$

$\max(X, Y, Y) :- Y > X.$

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

$\max(X, Y, X) :- X \geq Y, !.$

$\max(X, Y, Y) :- Y > X.$

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

$\max(X, Y, X) :- X \geq Y, !.$

$\max(X, Y, Y).$

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

$\max(X, Y, Z) :- X \geq Y, !, Z = X.$

$\max(X, Y, Y) /* :- Y > X */.$

Tartalom

5

Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- **Determinizmus és indexelés**
- Jobbrekurzió és akkumulátorok
- Imperatív programok átírása Prologba
- Vezérlési eljárások
- Magasabbrendű eljárások

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 **nem**determinisztikus lefutást:

```

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

```

A determinisztikus lefutás 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 eljárás hívások (pl. `<`, `=<`, `==`), és/vagy
 - kifejezés-típust ellenőrző eljárás hívások (pl. `atom`, `number`), és/vagy
 - általános összehasonlító eljárás hívások (pl. `@<`, `@=<`, `==`).
- Választásmentes kód keletkezik a „`fej :- felt, !, akkor.`” klózból, ha `fej` argumentumai különböző változók, és `felt` olyan mint fent.
- Például választásmentes kód keletkezik az alábbi definíciókból:

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

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

Indexelés

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

Példa indexelésre

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

- A $p(A, B)$ hívással illesztendő klózok:

- ha A változó, akkor (1) (2) (3) (4) (5)
- ha $A = 0$, akkor (1) (2)
- ha A fő funktora $s/1$, akkor (2) (3) (4)
- ha $A = 9$, akkor (2) (5)
- minden más esetben (2)

- Példák hívásokra:

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

Struktúrák, változók a fejargumentumban

- Ha a klózek szétválasztásához szükség van az első (struktúra) argumentum részeire is, akkor érdemes segédeljárást bevezetni.
- Pl. $p/2$ és $q/2$ ekvivalens, de $q(\text{Nonvar}, Y)$ determinisztikus lefutású!

$p(0, a).$	$q(0, a).$	$q_seged(0, b).$
$p(s(0), b).$	$q(s(X), Y) :-$	$q_seged(1, c).$
$p(s(1), c).$	$q_seged(X, Y).$	
$p(9, z).$	$q(9, z).$	

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

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

- 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 (funktoruk: `'[]'`/0 ill. `'.'`/2).
- A két klóz sorrendje nem érdekes (feltéve, hogy zárt listával hívjuk az első pozíción) – de azért tegyük a leálló klózt mindig előre.

Listakezelő eljárások indexelése: példák

- Az `append/3` választásmentesen fut le, ha első argumentuma zárt végű.

```
append([], L, L).
```

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

- A `last/2` közvetlen megfogalmazása nemdeterminisztikusan fut le:

```
% last(L, E): Az L lista utolsó eleme E.
```

```
last([E], E).
```

```
last([_|L], E) :- last(L, E).
```

- Érdemes segédeljárást bevezetni, `last2/2` választásmentesen fut

```
last2([X|L], E) :- last2(L, X, E).
```

```
% last2(L, X, E): Az [X|L] lista utolsó eleme E.
```

```
last2([], E, E).
```

```
last2([X|L], _, E) :- last2(L, X, E).
```

Az indexelés és a vágó kölcsönhatása

- Hogyan vehető figyelembe a vágó az indexelés fordításakor?
- Példa: a $p(1, A)$ hívás választásmentes, de a $q(1, A)$ nem!

$p(1, Y) :- !, Y = 2. \quad \% (1)$	$q(1, 2) :- !. \quad \% (1)$
$p(X, X). \quad \% (2)$	$q(X, X). \quad \% (2)$
$\text{Arg1}=1 \rightarrow (1), \text{Arg1} \neq 1 \rightarrow (2)$	$\text{Arg1}=1 \rightarrow \{(1), (2)\}, \text{Arg1} \neq 1 \rightarrow (2)$

- 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 klózokat.
- Példa: $p(X, D, E) :- X = s(A, B, C), !, \dots \quad p(X, Y, Z) :- \dots$
- 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: $f_1 = 1$; $f_2 = 2$; $f_n = f_{\lfloor 3n/4 \rfloor} + f_{\lfloor 2n/3 \rfloor}$, $n > 2$

```
% determ. xx='' | % determ. lefut. xx='c' | % választásmentes, xx='ci'
fib(1, 1).      | fibc(1, 1) :- !.      | fibci(1, F) :- !, F = 1.
fib(2, 2).      | fibc(2, 2) :- !.      | fibci(2, F) :- !, F = 2.
fib(N, F) :-    | fibc(N, F) :-         | fibci(N, F) :-
```

$N > 2$, N_2 is $N*3//4$, N_3 is $N*2//3$,
 $\text{fibxx}(N_2, F_2)$, $\text{fibxx}(N_3, F_3)$,
 F is F_2+F_3 .

- 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

- 5 Haladó Prolog
 - Meta-logikai eljárások
 - Megoldásgyűjtő beépített eljárások
 - A keresési tér szűkítése
 - Determinizmus és indexelés
 - **Jobbrekurzió és akkumulátorok**
 - Imperatív programok átírása Prologba
 - Vezérlési eljárások
 - Magasabbrendű eljárások

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. szemégyűjtésre alkalmassá válik.
- Ez az optimalizálás nemcsak rekurzív hívás esetén, hanem minden **utolsó** hívás esetén megvalósul – a pontos név: utolsó hívás optimalizálás (last call optimisation).
- A jobbrekurzió így tehát nem növeli a memória-igényt, korlátlan mélységig futhat – mint a ciklusok az imperatív nyelvekben. Példa:
`ciklus(Állapot) :- lépés(Állapot, Állapot1), !, ciklus(Állapot1).
ciklus(_Állapot).`

Predikátumok jobbrekurzív alakra hozása – listaösszeg

- A listaösszegzés „természetes”, nem jobbrekurzív definíciója:

`% sum0(+L, ?S): L elemeinek összege S (S = 0+Ln+Ln-1+...+L1).`

`sum0([], 0).`

`sum0([X|L], S):- sum0(L,S0), S is S0+X.`

- Jobbrekurzív lista-összegző:

`% sum(+L, ?S): L elemeinek összege S (S = 0+L1+L2+...+Ln).`

`sum(L, S):- sum(L, 0, S).`

`% sum(+L, +S0, ?S): L elemeit S0-hoz adva kapjuk S-t. ($\equiv \sum L = S-S0$)`

`sum([], S, S).`

`sum([X|L], S0, S):- S1 is S0+X, sum(L, S1, S).`

- A jobbrekurzív `sum` eljárás több mint **3-szor gyorsabb** mint a `sum0`!
- Az **akkumulátor** az imperatív (azaz megváltoztatható értékű) változó fogalmának deklaratív megfelelője:
 - A `sum/3`-ban az `S0` és `S` argumentumok akkumulátorpárt alkotnak.
 - Az akkumulátorpár két része az adott változó mennyiség (a példában az összeg) különböző időpontokban vett értékeit mutatja:
 - `S0` az összeg a `sum/3` **meghívásakor**: a változó kezdőértéke;
 - `S` az összeg a `sum/3` **lefutása után**: a változó végértéke.

Az akkumulátorok használata

- Az akkumulátorokkal általánosan több egymás utáni változtatást is leírhatunk:

```
p(..., A0, A):-
    q0(..., A0, A1), ...,
    q1(..., A1, A2), ...,
    qn(..., An, A).
```

- A `sum/3` második klóza ilyen alakra hozva:

```
sum([X|L], S0, S):- plus(X, S0, S1), sum(L, S1, S).
plus(X, S0, S) :- S is S0+X.
```

- Akkumulátorváltozók elnevezési konvenciója: kezdőérték: *Vál t0*; közbülső értékek: *Vál t1, ..., Vál tn*; végérték: *Vál t*.
- A Prolog akkumulátorpár nem más mint a funkcionális programozásból ismert gyűjtőargumentum és a függvény eredményének együttese.

Akkumulátorok használata – folytatás

- Többszörös akkumulálás – lista összege és négyzetösszege

```
% sum2(+L, +S0, ?S, +Q0, ?Q): S-S0 =  $\sum L_i$ , Q-Q0 =  $\sum L_i^2$ 
```

```
sum2([], S, S, Q, Q).
```

```
sum2([X|L], S0, S, Q0, Q):-
```

```
    S1 is S0+X, Q1 is Q0+X*X, sum2(L, S1, S, Q1, Q).
```

- Többszörös akkumulátorok összevonása egyetlen **állapotváltozóvá**

```
% sum3(+L, +S0/Q0, ?S/Q): S-S0 =  $\sum L_i$ , Q-Q0 =  $\sum L_i^2$ 
```

```
sum3([], SQ, SQ).
```

```
sum3([X|L], SQ0, SQ) :-
```

```
    plus3(X, SQ0, SQ1), sum3(L, SQ1, SQ).
```

```
    % teljesen analóg a "sima" összegzővel
```

```
plus3(X, S0/Q0, S/Q) :- S is S0+X, Q is Q0+X*X.
```

Tartalom

- 5 Haladó Prolog
 - Meta-logikai eljárások
 - Megoldásgyűjtő beépített eljárások
 - A keresési tér szűkítése
 - Determinizmus és indexelés
 - Jobbrekurzió és akkumulátorok
 - **Imperatív programok átírása Prologba**
 - Vezérlési eljárások
 - Magasabbrendű eljárások

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 \implies 2+1-argumentumú Prolog eljárás.
- A függvény eredménye \implies utolsó arg.: $\text{hatv}(+A, +H, ?E): A^H = E$.
- Ciklus \implies segédeljárás: $\text{hatv}(+A0, +H0, +E0, ?E): A0^{H0} * E0 = E$.
- »a« és »h« C változók \implies »+A0« és »+H0« bemenő *paraméterek* (nem kell végérték),
»e« C változó \implies »+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 megfeleltetethető egy Prolog változó (pl. h -nak $H0$, $H1$, ...):
 - A ciklusmag elején a C változók a megfelelő Prolog argumentumban levő változónak felelnek meg.
 - Egy C értékadásnak egy új Prolog változó bevezetése felel meg, az ez után következő kódban az új változó felel meg a C változónak.
 - Ha a diszjunkció, vagy if-then-else egyik ága megváltoztat egy változót, akkor a többi ágon is be kell vezetni az új Prolog változót, a régivel azonos értékkel (ld. `if (h & 1) ...`).
- A C ciklusmag végén a Prolog eljárást vissza kell hívni, argumentumaiban az egyes C változóknak pillanatnyilag megfeleltetett Prolog változóval.
- A C ciklus **ciklus-invariánsa** nem más mint a Prolog eljárás fejkommentje, a példában:

`% hatv(+A0, +H0, +E0, ?E): $A0^{H0} * E0 = E$.`

Programhelyesség-bizonyítás (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: $x = \text{mfoku_gyok}(a, b, c)$
 - előfeltételek: $b*b-4*a*c \geq 0$, $a \neq 0$
 - utófeltétel: $a*x*x+b*x+c = 0$
 - a program:

```
double mfoku_gyok(a, b, c)
double a, b, c;
{ double d = sqrt(b*b-4*a*c);
  return (-b+d)/2/a;
}
```
- A program helyességének bizonyítása lineáris kódra viszonylag egyszerű.

Ciklikus programok helyességének bizonyítása (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

5

Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Imperatív programok átírása Prologba
- **Vezérlési eljárások**
- Magasabbrendű eljárások

Vezérlési eljárások, a `call/1` beépített eljárás

- Vezérlési eljárás: A Prolog végrehajtáshoz kapcsolódó beépített eljárás.
- A vezérlési eljárások többsége **magasabbrendű** eljárás, azaz olyan eljárás, amely egy vagy több argumentumát eljáráshívásként értelmezi.
- A meta-eljárások fő képviselője a `call(+Cél)`:
 - Cél egy struktúra vagy névkonstans (vö. `callable/1`).
 - Jelentése (deklaratív szemantika): Cél igaz.
 - Hatása (procedurális szemantika): a Cél kifejezést **hívássá alakítja** és végrehajtja.
- A klóztörzsben célként megengedett egy `X` változó használata, ezt a rendszer egy `call(X)` hívássá alakítja át.

```
| kétszer(X) :- call(X), X.
```

```
| ?- kétszer(write(ba)), nl.    =>    baba
```

```
| ?- listing(kétszer).        =>    kétszer(X) :-
                                     call(X), call(X).
```

Vezérlési szerkezetek mint eljárások

- A `call/1` argumentumában szerepelhetnek vezérlési szerkezetek is, mert ezek beépített eljárásként is jelen vannak a Prolog rendszerben:
 - `(',')`/2: konjunkció.
 - `(';')`/2: diszjunkció.
 - `(->)`/2: if-then; `(';')`/2: if-then-else.
 - `(\+)`/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).
- 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 megghiúsláshoz
% szükséges időt, a Txt szöveg kisérétében.
time(Txt, Goal) :-
    statistics(runtime, [T0,_]), % T0 az indítás óta eltelt CPU idő,
                                % msec-ban (szemétgyűjtés nélkül).
    (
        call(Goal) -> Res = true
    ;   Res = false
    ),
    statistics(runtime, [T1,_]), T is T1-T0,
    format('~w futási idő = ~3d sec\n', [Txt,T]),
        % ~w formázó: kiírás a write/1 segítségével
        % ~3d formázó: I egész kiírása I/1000-ként, 3 tizedesre
    Res = true. % megghiúsul, ha Goal megghiúsult
```

További beépített vezérlési eljárások

- `once`(Cél): Cél igaz, és csak az első megoldását kérjük. Definíciója:
`once(X) :- call(X), !.`
 vagy, feltételes szerkezettel
`once(X) :- (call(X) -> true).`
- `true`: azonosan igaz, `fail`: azonosan hamis (mindig meghiúsul).
- `repeat`: végtelen sokszor igaz (végtelen választási pont). Definíciója:
`repeat.`
`repeat :- repeat.`
- A `repeat` eljárást egy mellékhatásos eljárás ismétlésére használhatjuk.
- Példa (egyszerű kalkulátor):

```
bc :- repeat, read(Expr),
    ( Expr = end_of_file -> true
    ; Res is Expr, write(Expr = Res), nl, fail
    ),
    !.
```
- A végtelen választási pontot kötelező egy vágóval semlegesíteni!

Példa: magasabbrendű reláció definiálása – 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 növvő:
    forall(append(_, [A,B|_], _L), A < B).
true ?
```

- forall/2 csak eldöntendő kérdés esetén használható.

Tartalom

5

Haladó Prolog

- Meta-logikai eljárások
- Megoldásgyűjtő beépített eljárások
- A keresési tér szűkítése
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Imperatív programok átírása Prologba
- Vezérlési eljárások
- **Magasabbrendű eljárások**

Magasabbrendű eljárások – listakezelés

- Magasabbrendű (vagy meta-eljárás) egy eljárás,
 - ha eljárásként értelmezi egy vagy több argumentumát
 - pl. `call/1`, `findall/3`, `\+ /1` stb.

- Listafeldolgozás `findall` segítségével – példák

- 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). \implies 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). \implies 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:

% Pred az A utolsó argumentummal meghívva igaz.

```
call(Pred, A) :-
```

```
    Pred =.. FAs0, append(FAs0, [A], FAs1),
```

```
    Pred1 =.. FAs1, call(Pred1).
```

% Pred az A és B utolsó argumentumokkal meghívva igaz.

```
call(Pred, A, B) :-
```

```
    Pred =.. FAs0, append(FAs0, [A,B], FAs2),
```

```
    Pred2 =.. FAs2, call(Pred2).
```

...

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.

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). \implies 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). \implies E = 321

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, NL), 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

VI. rész

Haladó Erlang

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

Tartalom

- 6 Haladó Erlang
 - Rekurzió fajtái
 - Halmazműveletek (rendezetlen listával)
 - Lusta farkú lista Erlangban

Rekurzió alapesetei

- 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).
```

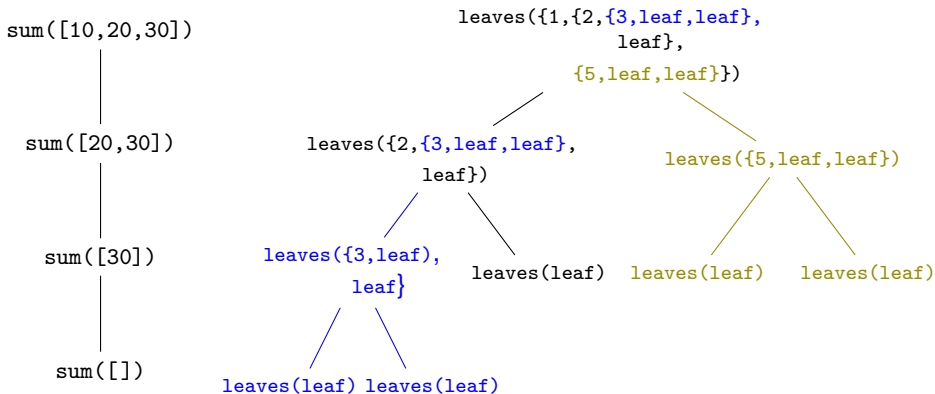
- Mindkettőből *rekurzív folyamat* jön létre, ha alkalmazzuk: minden egyes rekurzív hívás mélyíti a vermet

- 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* függvénye⁸

⁸Itt: lineáris függvénye.

Jobbrekurzió, iteráció

A rekurziót gyakran érdemes akkumulátorral 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 (azaz `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 ⁹	Lépések (CG mérete)	CG mélysége	Tárigény \approx 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 $h \leq 20$	$\Theta(m^h)$	$\Theta(h)$	

- Az |állapot| a CG egy pontjának a memóriamérete. Pl. szummázásnál az állapot a részösszeg, aminek a tárigénye logaritmikus (a számjegyek számával arányos). Az SMM-ben (CSP feladat) ez egy kitöltés memóriamérete.
- A rekurzióból fakadó tárigény lehet jelentős is (vö `sum/1`, `sumi/1`), és lehet elhanyagolható is a lépésekhez képest (SMM, Mastermind)
- 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 törekedni kell rá

⁹ $f(n) = \Theta(g(n))$ jelentése: $g(n) \cdot k_1 \leq f(n) \leq g(n) \cdot k_2$ valamilyen $k_1, k_2 > 0$ -ra

A jobbrekurzió mindig *nagyságrendekkel* előnyösebb? Nem!

- A jobbrekurzív `sum(L1)` tárigénye konstans (azaz $\Theta(1)$), a lineáris-rekurzív `sum(L1)` össz-tárigénye $\Theta(\text{length}(L1))$
- Melyiknek alacsonyabb a tárigénye?
 - `bevezeto:append(L1,L2)`
 - `R1=lists: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` tárigénye *nagyságrendileg* hasonló, mint a lineáris-rekurzív `append` függvényé!
- Ha az akkumulátor mérete nem konstans (azaz $\Theta(1)$), meggondolandó a jobbrekurzió. . .

Példa elágazó rekurzióra: Fibonacci-sorozat (1)

- 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; pl. írjuk át a Fibonacci-számok (0,0,1,1,2,3,5,8,13,...) matematikai definícióját programmá

$$F(n) = \begin{cases} 0, & \text{ha } n = 0, \\ 1, & \text{ha } n = 1, \\ F(n-2) + F(n-1) & \text{különben.} \end{cases}$$

Naív Fibonacci, előfelt.: $N \in \mathcal{N}$

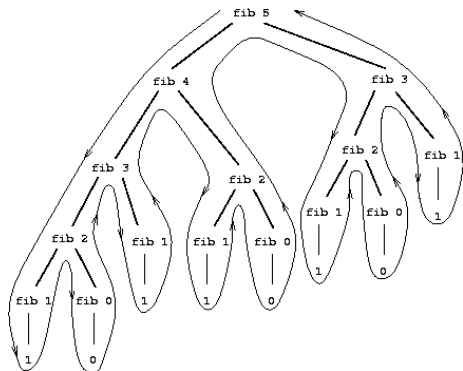
fib(0) -> 0;

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.
- Forrás:* Structure and Interpretation of Computer Programs, 2nd ed., by H. Abelson, G. J. Sussman, J. Sussman, The MIT Press, 1996

Példa elágazó rekurzióra: Fibonacci-sorozat (2)



- Elágazó-rekurzív folyamat hívási fája fib(5) kiszámításakor
- Alkalmatlan a Fibonacci-számok előállítására
- Az $F(n)$ meghatározásához pontosan $F(n+1)$ levélből álló fát kell bejárni, azaz ennyiszor 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

Példa elágazó rekurzióra: Fibonacci-sorozat (3)

- A lépések száma – $F(n)$ -hez hasonlóan – 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 $F(1) \equiv 1$, ill. $F(0) \equiv 0$, és ismétlődve 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-edik Fibonacci-szám.
fibi(N) -> fibi(N-1, 1, 0).
```

```
% fibi(N,A,B) az A←A+B, B←A transzformáció N-edik 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 rekurziónál n -nel exponenciálisan, lineáris rekurziónál n -nel arányosan nőtt!
- Pl. a `tree:leaves/1` függvény is lineáris-rekurzívvá alakítható, **de** ezzel nem javítható a hatékonysága: valamilyen LIFO tárolót kellene használni a mélységi bejáráshoz a rendszerem helyett

Programhelyesség informális igazolása (1)

- Egy rekurzív programról is be kell látnunk – az iteratív programhoz hasonlóan –, hogy
 - funkcionálisan helyes (azaz azt kapjuk eredményül, amit várunk)
 - a kiértékelése biztosan befejeződik (nem „végtelen” a rekurzió)
- Ellenpélda: a `fac(-1)` hívás végtelen ciklushoz vezet, bár az argumentum minden rekurzív híváskor csökken
- A helyesség 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 `fac/1` az $\mathbb{N} = 0$ 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 a programhelyesség informális igazolását

Programhelyesség informális igazolása (2)

-spec $\text{map}(\text{fun}(A) \rightarrow B, [A]) \rightarrow [B]$.

$\text{map}(_F, []) \rightarrow []$;

$\text{map}(F, [X|Xs]) \rightarrow [F(X)|\text{map}(F, Xs)]$.

- 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 *rekurziót tartalmazó klózban* a függvényt minden lépésben *egyre rövidülő* listára alkalmazzuk (a strukturális tulajdonság „csökken”), és
 - a rekurzió le fog állni, mert *van rekurziót nem tartalmazó klóz*, amire az *alapesetben*, a strukturális tulajdonság zérussá válásakor kerül sor.

Tartalom

- 6 Haladó Erlang
 - Rekurzió fajtái
 - Halmazműveletek (rendezetlen listával)
 - Lusta farkú lista Erlangban

Tagsági vizsgálat

- A halmazt mi most rendezetlen listával ábrázoljuk
- A műveletek sokkal hatékonyabbak volnának rendezett adatszerkezettel (pl. rendezett lista, keresőfa, hash)
- Erlang STDLIB: sets, ordsets (based on **ordered** lists), gb_sets (ordered sets based on **general balanced** trees))

set.erl – Halmazkezelő függvények

```
-type set() :: list().
-spec empty() -> E::set().    % E az üres halmaz.
empty() ->                    % Az absztrakció miatt szükséges:
    [].                        % ábrázolástól független interfész.

-isMember(X::any(), Ys::set()) -> B::boolean().
% B igaz, ha az X elem benne van az Ys halmazban.

isMember(_, []) -> isMember2(_, []) -> false;
    false;          isMember2(X, [X|_Ys]) -> true;
isMember(X, [Y|Ys]) -> isMember2(X, [_Y|Ys]) ->
    X==Y orelse isMember(X,Ys).          isMember2(X, Ys).
```

- *Megjegyzés:* orelse, mint már tudjuk, lusta kiértékelésű

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()) -> Rs::set().  
% Az Rs 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 ismétlődésmentes halmaza.  
listToSet([]) ->  
    [];  
listToSet([X|Xs]) ->  
    newMember(X, listToSet(Xs)).
```

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`
 - részhalmaza-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 zárthelyin várhatók ilyen feladatok.)

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)
    ].
```

Részhalmaz-e, egyenlők-e

set.erl – folytatás

```
-spec isSubset(Xs::set(), Ys::set()) -> B::boolean().
```

```
% 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::boolean().
```

```
% B igaz, ha Xs és Ys elemei azonosak.
```

```
isEqual(Xs, Ys) ->
```

```
    isSubset(Xs, Ys) andalso isSubset(Ys, Xs).
```

- isSubset lassú a rendezetlenség miatt
- andalso, mint már tudjuk, 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, jelölése: $P(S)$
- $P(S)$ -t *rekurzívan* például úgy állíthatjuk elő, hogy kiveszünk S -ből egy x elemet, majd előállítjuk az $S \setminus \{x\}$ hatványhalmazát
- Példa: $S = \{10, 20, 30\}$, $x \leftarrow 10$, $P(S \setminus \{x\}) = \{\{\}, \{20\}, \{30\}, \{20, 30\}\}$
- 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 $P(\{10, 20, 30\}) =$
 $\{\{\}, \{20\}, \{30\}, \{20, 30\}\} \cup \{\{\}\cup\{10\}, \{20\}\cup\{10\}, \{30\}\cup\{10\}, \{20, 30\}\cup\{10\}\}$

% 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 ]
    end,
    [[]],
    Xs).
```

Halmaz hatványhalmaza – hatékonyabb változat

- $A P ++ [[X|Ys] \ || \ Ys \leftarrow P]$ 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
```

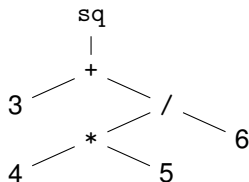

Tartalom

- 6 Haladó Erlang
 - Rekurzió fajtái
 - Halmazműveletek (rendezetlen listával)
 - **Lusta farkú lista Erlangban**

Összetett kifejezés kiértékelése

- Egy összetett kifejezést az Erlang két lépésben értékeli ki **mohó** kiértékeléssel, az alábbi rekurzív kiértékelési szabály szerint:
 - Először** kiértékeli az operátort (műveleti jelet, függvényjelet) és az argumentumait,
 - majd **ezután** alkalmazza az operátort az argumentumokra.
- A kifejezéseket *kifejezésfával* ábrázoljuk
- 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 (fenti 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:

$$\text{sq}(X) \rightarrow X * X.$$

$$\text{sumsq}(X, Y) \rightarrow \text{sq}(X) + \text{sq}(Y).$$

$$f(A) \rightarrow \text{sumsq}(A+1, A*2).$$

Mohó kiértékelés esetén minden lépésben egy részkifejezést egy vele egyenértékű kifejezéssel helyettesítünk. Pl. az $f(5)$ mohó kiértékelése:

$$f(5) \rightarrow \text{sumsq}(5+1, 5*2) \rightarrow \text{sumsq}(6, 5*2) \rightarrow \text{sumsq}(6, 10) \rightarrow \text{sq}(6) + \text{sq}(10) \rightarrow 6*6 + \text{sq}(10) \rightarrow 36 + \text{sq}(10) \rightarrow 36 + 10*10 \rightarrow 36 + 100 \rightarrow 136$$

- A függvényalkalmazás itt bemutatott *helyettesítési modellje*, az „egyenlők helyettesítése egyenlőkkel” (*equals replaced by equals*) segíti a függvényalkalmazás *jelentésének* megértését
- Olyan esetekben alkalmazható, amikor egy függvény *jelentése független* a környezetétől (pl. ha minden mellékhatás ki van zárva)
- A 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 nevezzük *mohó* (eager) vagy *applikatív sorrendű* (applicative order) kiértékelésnek
- 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
- Pl. 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
```

- Pl. a `false andalso f(5) > 100` lusta kiértékelése:

```
false andalso f(5) > 100 → false
```

Mohó és lusta kiértékelés összevetése

- 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 a lusta kiértékelést használó, jobb fordítók (pl. Alice, Haskell) úgy kerülik el, hogy
 - az azonos részkifejezéseket megjelölik,
 - 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, két elemű ennes (pár), nem csak listákra használhatjuk:
 - 1> `[1|[2]]`. *% Lista, mert a | utáni rész lista.*
[1,2]
 - 2> `[1|[2|[]]]`. *% Lista, mint az előző.*
[1,2]
 - 3> `[1|2]`. *% Pár, mert a | utáni rész nem lista.*
[1|2]
- A következő fóliákon az átláthatóság kedvéért a listaszintaxist használjuk egy két elemű ennesre (párra): a lusta listára

lazy.erl – Lusta farkú lista

```
-type lazy:list() :: [] | [any()|fun() -> lazy:list()]
```

- A fenti szerkezetben a második tag (a fark) – a függvénydefiníció miatt – *késleltett kiértékelésű* (vö. delayed evaluation)

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 vesszük 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