

Deklaratív programozás

Hanak Péter
hanak@inf.bme.hu

Irányítástechnika és Informatika Tanszék
OM Kutatás-Fejlesztési Helyettes Államtitkárság

Szeredi Péter, Benkő Tamás
{szeredi,benko}@iqsoft.hu

Számítástudományi és Információelméleti Tanszék
IQSOFT Intelligens Software Rt.

Deklaratív programozás: tudnivalók

Honlap, levelezési lista

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

- Szeredi Péter, Benkő Tamás: Deklaratív programozás. Bevezetés a logikai programozásba.
- Hanak D. Péter: Deklaratív programozás. Bevezetés a funkcionális programozásba.
- Új, bővített kiadások 2001-ben, kétetenként 650 Ft; pdf változatban a honlapon
- Jegyzetrendelés: a honlapon megadandó módon
- A 2001. tavaszi félév előadásainak főlái a honlapon (pdf)

KÖVETELMÉNYEK — TUDNIVALÓK

Deklaratív programozás: tudnivalók (folyt.)

Fordító- és értelmezőprogramok

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

Deklaratív programozás: félévközi követelmények

Nagy házi feladat (NHF)

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

Deklaratív programozás, BMEF, 2001 tavaszi félév

0. előadás (követelmények)

Követelmények 7

Deklaratív programozás: félévközi követelmények (folyt.)

Kis házi feladatok (KHF)

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

Cykorló feladatok

- Nem kötelező, de nagyon ajánlott!
- Cykorlás a honlapon keresztül
- A gyakorlatok megoldását nyilvántartjuk
- Pontot nem adunk

Deklaratív programozás, BMEF, 2001 tavaszi félév

0. előadás (követelmények)

Deklaratív programozás: félévközi követelmények (folyt.)

Nagy házi feladat (folyt.)

- Nem kötelező, de *nagyon* ajánlott!
- 40%-os szabály (nyelvenként a max. részpontszám 40%-a kell az eredményességhez)
- Csak Prolog- vagy csak SML-változat is beadható
- Súlya az osztályzatban: 15%

Deklaratív programozás, BMEF, 2001 tavaszi félév

0. előadás (követelmények)

Követelmények 8

Deklaratív programozás: félévközi követelmények (folyt.)

Nagyzárthelyi, pótzárthelyi (NZH, PZH)

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

Deklaratív programozás, BMEF, 2001 tavaszi félév

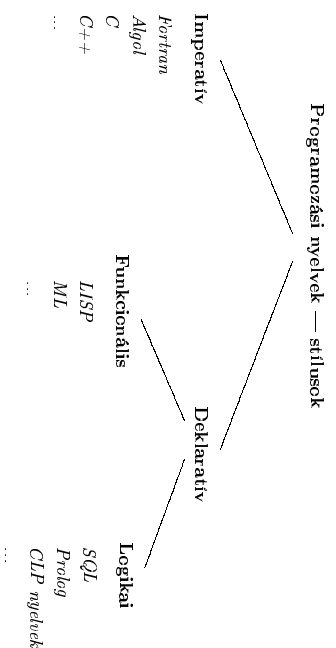
0. előadás (követelmények)

Deklaratív programozás: vizsga

Vizsga

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

Programozási nyelvek osztályozása



DEKLARATÍV ÉS IMPERATÍV PROGRAMOZÁS

Imperatív és deklaratív programozási nyelvek

Imperatív program

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

Deklaratív program

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

Deklaratív nyelvek jelszavai

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

Példa — családi kapcsolatok

Adatok

Egy gyerek–szülő kapcsolat, pl.

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

A feladat:

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

Deklaratív programozás, BMIF, 2001 tavaszi félév

1. előadás

A nagyszülő feladat — C nyelvű megoldás

```

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

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

```

Deklaratív programozás, BMIF, 2001 tavaszi félév

1. előadás

A nagyszülő feladat — SML megoldás

```

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

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

```

A függvény futtatása

```

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

```

Deklaratív programozás, BMIF, 2001 tavaszi félév

1. előadás

A nagyszülő feladat — SQL megoldás

```

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

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

```

```

SQL> select * from nagyszulok;
GYEREK          SZULO
-----
Imre             Civakodó Henrik
Imre             Burgundi Gizella
Imre             Géza
Imre             Sarolta
SQL>

```

Deklaratív programozás, BMIF, 2001 tavaszi félév

1. előadás

A nagyszülő feladat — Prolog megoldás

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

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

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

Deklaratív programozás, BMEF, 2001 tavaszi félév

1. előadás

Egy „számolás” program: faktoriális

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

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

Tanulságok, érdekességek

- SML skatulyázott függvények ⇔ Prolog egymás melletti relációk
- Negatív argumentumra a Prolog kód meghússul, az SML kód végtelen ciklusba esik

Deklaratív programozás, BMEF, 2001 tavaszi félév

1. előadás

A deklaratív és imperatív megoldások összehasonlítása

A keresési feladat megoldása

- C nyelven: ciklussal
- SQL-ben: beépített adatbázis-kereséssel
- SML-ben: magasabrendű függvénybe rejtett rekurzíóval
- Prologban: beépített mintaillesztéses eljárás hívással

Az összetett feltételek kezelése

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

A funkcionális és logikai megoldásokról

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

Deklaratív programozás, BMEF, 2001 tavaszi félév

1. előadás

Egy összetettebb példa: N-edik generációs ősök

SML megoldás

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

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

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

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

Deklaratív programozás, BMEF, 2001 tavaszi félév

1. előadás

N-edik generációs ősök — Prolog megoldás

```
% ose(N, E0, E): E0-nak N-edik generációs őse az E.
% **** N adott szám. ****
ose(0, E, E).
ose(N, E0, E) :-
    N > 0, M1 is N-1,
    szuloje(E0, Sz),
    ose(M1, Sz, E).
```

Futása

```
| ?- ose(2, 'Imre', Os).
Os = 'Géza' ? ;
Os = 'Sarelt' ? ;
Os = 'Civakodó Henrik' ? ;
Os = 'Burgundi Gizella' ? ;
no

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

N-edik generációs ősök — általánosabb Prolog megoldás

```
% osel(N, E0, E): E0-nak N-edik generációs őse az E.
osel(0, E, E).
osel(N, E0, E) :-
    szuloje(E0, Sz), osel(M1, Sz, E), N is M1+1.
```

Futása

```
| ?- osel(N, 'Imre', Os), N < 2.
N = 0, Os = 'Imre' ? ;
N = 1, Os = 'István' ? ;
N = 1, Os = 'Gizella' ? ;
no
| ?- osel(1, Utod, 'Burgundi Gizella').
U = 0, Utod = 'Burgundi Gizella' ? ;
U = 2, Utod = 'Imre' ? ;
U = 1, Utod = 'Gizella' ? ;
no
```

A funkcionális programozásról dióhéjban

Alapeszme

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

A funkcionális programozás első megvalósítása: LISP

- alapfőlet: listák könnyű/hatékony feldolgozása

A funkcionális programozás egy modern megvalósítása: SML

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

SML — előnyök és hátrányok

Miért jó?

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

Mik a hátrányai?

- megszokottól eltérő programozói stílus

Hogyan tovább?

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

A logikai programozásról dihéjban

Alapszigne

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

A logikai programozás első megvalósítása: a Prolog nyelv

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

Prolog — előnyök és hátrányok

Miért jó?

- tönör kód, többirányú eljárások
 - „automatikus” visszalépéses keresés, ciklusok kiváltása
 - „logikai” változó — meghatározatlan adatok kezelése
- Mik a hátrányai?

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

Hogyan tovább?

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

Deklaratív programozás — miért tanfjünk?

Új, magasszintű programozási elemek

- rekurzió
- mintaillesztés
- visszalépéses keresés

Új gondolkodási stílus

- a programrészek (relációk, függvények) önálló jelentéssel bírnak
- a kód és a jelentés összevethető: program-verifikáció

Új alkalmazási területek

- szimbolikus alkalmazások
- következtetési módszerekre épülő megoldások
- nagyfokú megbízhatóságot igénylő rendszerek

Egy példa: párbeszéd egy 50 soros Prolog programmal

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

BEVEZETÉS A LOGIKAI PROGRAMOZÁSBA

Bevetés 31

A Prolog/LP rövid történeti áttekintése

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

Bevetés a Logikai Programozásba

Az előadássorozat áttekintése

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

Bevetés 32

Információk a logikai programozásról

Prolog megvalósítások:

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

Hálózati információforrások:

The WWW Virtual Library: Logic Programming:
<http://www.comlab.ox.ac.uk/archive/Logic-prog.html>

CMU Prolog Repository:

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

Prolog FAQ:

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

Prolog Resource Guide:

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

Magyar nyelvű Prolog irodalom

Farkas Zsuzsa, Futó Iván, Langer Tamás, Szeredi Péter:
Az MProlog programozási nyelv.

Műszaki Könyvkiadó, 1989

Márkus Zsuzsa: Prologban programozni könnyű.

Novotrade, 1988

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

Aula Kiadó, 1999

Deklaratív programozás, BNMF, 2001 tavaszi félév 2. előadás (logikai programozás)

Prolog szintaxis 35

```
% két klózból álló predikátum definíciója, funktora: ose/3
ose(O, E, E),
ose(W, E0, E) :-
    szuloje(E0, Sz),
    ose(M1, Sz, E), N is M1+1.

% 1. klóz, tényállítas
% feje | törzs | 2. klóz, szabály
% cél | cél, cél |

<Prolog program > ::= <predikátum > ...
<predikátum > ::= <klóz > ... {azonos funktor}
<klóz > ::= <tényállítás > | <szabály > |
<tényállítás > ::= <fej >
<szabály > ::= <fej > :- <törzs >
<törzs > ::= <cél >, ...
<cél > ::= <kifejezés >
<fej > ::= <kifejezés >

• Alternatív szöhasználat:
• predikátum — eljárás
• cél — hívás
```

Deklaratív programozás, BNMF, 2001 tavaszi félév 2. előadás (logikai programozás)

A PROLOG NYELV KÖZELÍTŐ SZINTAXISA

Deklaratív programozás, BNMF, 2001 tavaszi félév 2. előadás (logikai programozás)

Prolog szintaxis 36

```
% ose(O, E, E) % összetett kifejezés, funktora ose/3
% | | |
% struktúranév | argumentum, változó
% \- argumentum, számkonstans

<kifejezés > ::= <változó > |
               <konstans > |
               <összetett kifejezés >
               <számkonstans >
               <összetett kifejezés > ::= <struktúranév > ( <argumentum >, ... )
               <struktúranév > ::= <névkonstans >
               <argumentum > ::= <kifejezés >

• Összetett kifejezés funktora = struktúranév/argumentumszám, pl. ose/3
• konstans funktora = konstans/0, pl. ?István/0
• változónak nincs funktora
```

Deklaratív programozás, BNMF, 2001 tavaszi félév 2. előadás (logikai programozás)

Lexikai elemek

```
% változó:      Fakt FAKT_fakt X2_2 _
% névkonstans: fakt ≡ 'fakt' 'István' □ ; ' ' += ** \= ≡ '\|= '
% számkonstans: 0 -123 10.0 -12.1e8
% nem (egyetlen) névkonstans: !_, István
% nem (egyetlen) számkonstans: !e8 1.e2

<változó>      <nagybetű>\<alfanum>...|
                _<alfanum>...|
<névkonstans> ::= ? <névkar>'...' |
                <kisbetű>\<alfanum>...|
<névkar>       ::= {tetszőleges nem ? és nem \ karakter} |
                \<escapé szekvencia>
<alfanum>      ::= <kisbetű> | <nagybetű> | <számjegy> | _
<tapadó_jel>   ::= + | - | * | / | \% | ^ | ~ | < | > | = | ' | ' | . | ? | @ | # | &
<számkonstans> ::= {előjelen vagy előjeltelen számjegysorozat}
                esetlegesen tízedes részsel és exponenssel}
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

2. előadás (logikai programozás)

Bépfített operátorok

```
Szabványos operátorok
1200 xfx :-, ->
1200 fx :-, ?-
1100 xfy ;
1050 xfy ->
1000 xfy ;, ;
900 fy \+
700 xfx < = \= =. . . =: = < = \=
700 xfx =\= > >= is @< @=< @> @>=
500 yfx + - \ \
400 yfx * // rem mod! << >>
200 xfx **
200 xfy ^
200 fy _?, \
```

¹!astatus nohban 200 xfx operátor
²!astatus nohban 500 fx operátor
³!astatus nohban 200 fy operátor

Egyéb bépfített operátorok

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

2. előadás (logikai programozás)

Szintaktikus édesítőszerek: operátorok

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

Operátor-deklaráció

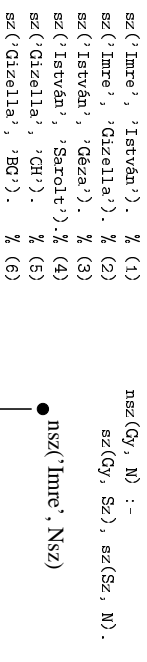
- :- op((prioritás), <fajta>, <operátornév>).
- <operátornév> tetszőleges névkonstans
- <prioritás> 0–1200 közötti egész
- <fajta>
 - infix: yfx, xfy, xfx; A op B ≡ op(A, B)
 - prefix: fx, fy; op A ≡ op(A)
 - postfix: xf, yf; A op ≡ op(A)
- a <fajta>-ban x és y az asszociativitást határozzák meg:
- x: az adott oldalon nem állhat azonos prioritású operátor zárójelezetlenül
- y: az adott oldalon állhat azonos prioritású operátor zárójelezetlenül

Deklaratív Programozás, BMEF, 2001 tavaszi félév

2. előadás (logikai programozás)

A PROLOG VÉGREHAJTÁSI MECHANIZMUSA

Prolog végrehajtási példa



A végrehajtás alapelvei: redukációs lépés

Redukációs lépés

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

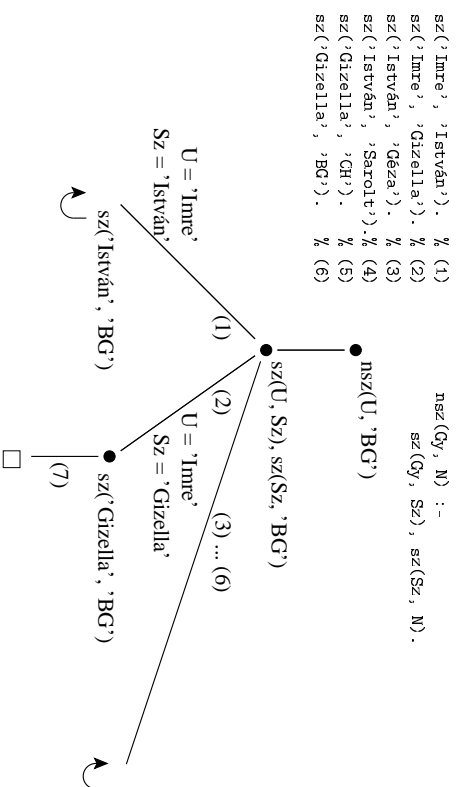
A végrehajtás alapelvei: egyesítés

Az eljáráshívás és egy klózfej azonos alakra hozása, változók behelyettesítésével

Példák

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

Újabb végrehajtási példa



A Prolog végrehajtási algoritmus

- (Kezdeti beállítások):* A verem üres, $CS :=$ célsorozat
- (Beérthet eljárások):* Ha CS első célja beérthet akkor haltsuk végre,
 - Ha sikertelen \Rightarrow 6. lépés.
 - Ha sikeres, elvégezzük a behelyettesítéseket, CS -ből elhagyjuk az első hívást, \Rightarrow 5. lépés.
- (Külsőszintű kezdőértékesítés):* $I = 1$.
- (Redukciós lépés):* CS első hívásához tartozó eljárásdefinicióban N klóz van,
 - Ha $I > N \Rightarrow$ 6. lépés.
 - Redukciós lépés az I -edik klóz és a CS célsorozat között.
 - Ha sikertelen, akkor $I := I+1 \Rightarrow$ 4. lépés.
 - Ha $I < N$ (nem utolsó), akkor veremülük $\langle CS, I \rangle$ -t.
 - $CS :=$ a redukciós lépés eredménye
- (Siker):* Ha CS üres, akkor sikeres vég, egyébként \Rightarrow 2. lépés.
- (Visszalépés):* Ha a verem nem üres, akkor leemeljük a veremből $\langle CS, I \rangle$ -t, $I := I+1$, és \Rightarrow 4. lépés.

Deklaratív programozás, BMEF, 2001 tavaszi félév

2. eloadás (logikai programozás)

Prolog végrehajtás 47

Előzetes — programfejlesztési beépített eljárások

- `consult(File)` vagy `[File]: A File` állományban levő programot beolvassa és értelmezendő alakban eltárolja. (`File = user` \Rightarrow terminálról olvas.)
 - `listing` vagy `listing(Predikátum)`: Az értelmezendő alakban eltárolt összes ill. adott nevű predikátumokat kiíratja.
 - `compile(File)`: A `File` állományban levő programot beolvassa, lefordítja.
 - A beolvasott program elemei Prolog kifejezések (`'-'`, `''`, operátorok!)
 - `halt`: A Prolog rendszer befejezi működését.
- ```
> sicstus
SICStus 3.8.5 (x86-linux-glibc2.1): Fri Oct 27 10:16:41 CEST 2000
| ?- consult(fakt).
{consulted /home/user/fakt.pl in module user, 10 msec 384 bytes}
| ?- listing(fakt).
fakt(G, 1).
fakt(A, B) :-
| A>0, C is A-1, fakt(C, D), B is D*A.
| ?- halt.
>
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

3. eloadás (logikai programozás)

## Előzetes — aritmetikai beépített eljárások

- $X$  is  $Kif1$ : A  $Kif1$  aritmetikai kifejezés értékét egyesíti  $X$ -szel
  - $Kif1 < Kif2$ ,  $Kif1 = Kif2$ ,  $Kif1 > Kif2$ ,  $Kif1 >= Kif2$ ,  $Kif1 := Kif2$ ,  $Kif1 = \setminus Kif2$ : A  $Kif1$  és  $Kif2$  aritmetikai kifejezések értéke a megadott relációban van egymással ( $= := \Rightarrow$  egyenlő,  $= \setminus \Rightarrow$  nem-egyenlő).
  - Ha  $Kif1$ ,  $Kif2$  valamelyike nem aritmetikai kifejezés  $\Rightarrow$  hiba.
  - Legfontosabb aritmetikai operátorok: `+`, `-`, `*`, `/`, `mod`, `//` (egész-osztás)
- ```
| ?- X is 1*2+3.
X = 5 ?
| ?- X is alma.
{DOMAIN ERROR: _78 is alma - arg 2: expected expression, found alma}
| ?- X =:= 1*2+3.
{INSTANTIATION ERROR: 84:=:1*2+3 - arg 1}
| ?- 1+2*3 > 2*3+1.
no
| ?-
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

3. eloadás (logikai programozás)

Prolog végrehajtás 48

Előzetes — kinyit és egyéb eljárások

- `write(X)`: Az X Prolog kifejezést kiírja (ha kell, operátorokkal).
 - `display(X)`: Az X Prolog kifejezést struktúra-alakban kiírja.
 - `nl`: Kinyit egy új sort.
 - `true`, `fail`: Mindig sikerül ill. mindig meghiúsul.
 - `trace`, `notrace`: A (teljes) nyomonkövetést be- ill. kikapcsolja.
 - `spy` Predikátum: Töréspontot helyez a Predikátum-ra.
- ```
| ?- write(+1,*2,3)), write('
1+2*3
+(1,*2,3))
yes
| ?- szuloje('István', X), write(X), nl, fail.
Géza
Sarolt
no
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

3. eloadás (logikai programozás)

### Prolog végrehajtási példa — rekurzív vevem nélkül

```

/* (1) */ fakt(0, 1).
/* (2) */ fakt(N, F) :-
 N>0, NN is N-1, fakt(NN, FF), F is FF*N.

● Kezdeti célsorozat: fakt(2,X), write(X)
● Redukció (2)-vel: 2>0, NN is 2-1, fakt(NN, FF), X is FF*2, write(X)
● Beépítettek végrehajtása: fakt(1, FF), X is FF*2, write(X)
● Redukció (2), béép.: fakt(0, FF1), FF is FF1*1, X is FF*2, write(X) (*)
● Redukció (1)-gyel (választási pont!): FF is 1*1, X is FF*2, write(X)
● Béép.: write(2), mellékhatás: =>2 kitérása, sikeres vég
● További megoldás kérése esetén visszalépés (*)-hoz, redukció (2)-vel:
0>0, NN2 is 0-1, fakt(NN2, FF2), ..., béép. végrehajt., meghívásulás.

```

### Prolog végrehajtás — egy aritmetikai példa

#### Példa: „jó” számok

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

```

% decl(J): J egy pozitív decimális számjegy.
decl(1). decl(2). decl(3). decl(4).
decl(5). decl(6). decl(7). decl(8). decl(9).

% decl(J): J egy decimális számjegy.
decl(0).
decl(J) :- decl(J).

% Szam négyzete háromjegyű és a Szam fordítottjával kezdődik.
josszam(Szam) :-
 decl(A), decl(B),
 Szam is A * 10 + B, Szam * Szam // 10 == B * 10 + A.

```

### Prolog végrehajtási példa — nyomkövetés

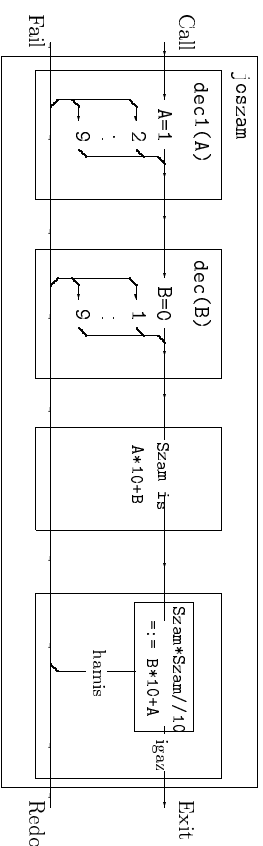
|                                           |                         |
|-------------------------------------------|-------------------------|
| ?- trace, fakt(2, X), write(X), nl, fail. |                         |
| 1 1 Call: fakt(2,X) ?                     | 1 1 Redo: fakt(2,2) ?   |
| 2 2 Call: 2>0 ?                           | 4 2 Redo: fakt(1,1) ?   |
| 3 2 Exit: 2>0 ?                           | 7 3 Redo: fakt(0,1) ?   |
| 4 2 Call: NN is 2-1 ?                     | 10 4 Call: 0>0 ?        |
| 5 2 Exit: 1 is 2-1 ?                      | 10 4 Fail: 0>0 ?        |
| 6 2 Call: fakt(1,FF) ?                    | 7 3 Fail: fakt(0,FF1) ? |
| 7 3 Exit: 1>0 ?                           | 4 2 Fail: fakt(1,FF) ?  |
| 8 3 Call: NN1 is 1-1 ?                    | 1 1 Fail: fakt(2,X) ?   |
| 9 3 Exit: 0 is 1-1 ?                      |                         |
| 10 3 Call: fakt(0,FF1) ?                  |                         |
| 11 3 Exit: fakt(0,1) ?                    |                         |
| 12 3 Call: FF is 1*1 ?                    |                         |
| 13 3 Exit: 1 is 1*1 ?                     |                         |
| 14 2 Exit: fakt(1,1) ?                    |                         |
| 15 2 Call: X is 1*2 ?                     |                         |
| 16 2 Exit: 2 is 1*2 ?                     |                         |
| 17 1 Exit: fakt(2,2) ?                    |                         |
| 18 ?                                      |                         |
| 19 2                                      |                         |

### Prolog végrehajtás — a 4-kapus doboz modell

```

josszam(Szam) :-
 decl(A), decl(B),
 Szam is A * 10 + B, Szam * Szam // 10 == B * 10 + A.

```



## Prolog végrehajtás — számintervallum felsorolása

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

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

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

3. előadás (logikai programozás)

## Prolog végrehajtás — diszjunktív kiváltása

```
% Utodnak valódi öse Os.
valodi_ose(Utod, Os) :-
 szuloje(Utod, Szulo),
 (Os = Szulo
 ; valodi_ose(Szulo, Os)
).
```

- Meghatározzuk a diszjunktívban szereplő változókat: Os, Szulo.
- Meghatározzuk a diszjunktívon kívüli szereplő változókat: Utod, Os, Szulo.
- A segéd eljárás argumentumai a közös változók: seged(Szulo, Os).
- Minden alternatívájából egy külön klóz lesz a segéd eljárásban.
- A diszjunktívt a segéd eljárás hívásával helyettesítjük.
 

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

seged(Szulo, Os) :- Os = Szulo.
seged(Szulo, Os) :- valodi_ose(Szulo, Os).
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

4. előadás (logikai programozás)

## Prolog végrehajtás — diszjunktív mint szemantikus édesítőszert

### Példa:

```
% Sz-nek gyermeke Gy.
gyermeke(Sz, Gy) :- fia(Sz, Gy).
gyermeke(Sz, Gy) :- lany(Sz, Gy).
```

Azonos fejű szabályok összevonhatók egy diszjunktív bevezetésével:

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

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

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

Az X = Y beépített eljárás definíciója:

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

4. előadás (logikai programozás)

## A PROLOG ADATFOGALMA

## Összetett adatstruktúrák — példa

- egy ember egy két mezőből álló rekord: vezetéknev, keresztnév
  - egy kétargumentumú összetett kifejezéssel ábrázoljuk
  - a struktúrájánév legyen pl. a mínusz\_jel (-): -(VMév, KNév)  $\equiv$  VMév-KNév
- ```

% szuloje(Gy, Sz): Gy szülője Sz.
szuloje(szabo-laszlo, szabo-gyorgy).
szuloje(szabo-laszlo, laszlo-amalia).
szuloje(szabo-amalia, szabo-gyorgy).
szuloje(szabo-amalia, laszlo-amalia).

% vezetekneve(E, VN): E vezetékeve VN.
vezetekneve(Nev-, Nev).

| ?- szuloje(E1, E2), keresztneve(E1, K), vezetekneve(E2, K).
K = laszlo, E1 = szabo-laszlo, E2 = laszlo-amalia ? ;

| ?- szuloje(V1-K1, K1-K2).
K1 = laszlo, K2 = amalia, V1 = szabo ?

yes
| ?- szuloje(E1, E2), E1 = _-Nev, E2 = Nev-_.
E1 = szabo-laszlo, E2 = laszlo-amalia, Nev = laszlo ?

```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

4. előadás (logikai programozás)

Többszörösen összetett adatok — többtagú nevek

- többtagú kereszl- és vezetéknevek kezelése, pl. Kovacs Eva Maria
 - a tagokat egy másik struktúrájával kapcsoljuk össze, pl. szabo-eva/maria
 - ugyanazt a - struktúránevét használjuk: kis-szabo-laszlo, szabo-(eva-maria)
- ```

% reszneve(Osszetett, Resz): Osszetett név része a Resz név.
reszneve(Nev, Nev).
reszneve(EIso-, Nev) :-
 reszneve(EIso, Nev).
reszneve(_-Masodik, Nev) :-
 reszneve(Masodik, Nev).

| ?- reszneve(kis-kovacs-bela, Resz).
Resz = kis-kovacs-bela ? ;

Resz = kis-kovacs ? ;

Resz = kovacs ? ;

Resz = bela ? ;

no

| ?- reszneve(szabo-(eva-maria), Resz).
Resz = szabo-(eva-maria) ? ;

Resz = szabo ? ;

Resz = eva-maria ? ;

Resz = eva ? ;

Resz = maria ? ;

no

```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

4. előadás (logikai programozás)

## Variációk egy témára — a logikai változó fogalma

- ```

% E keresztneve megegyezik egyik szülőjének vezetékevével
erdemes(E) :-
    szuloje(E, Sz),
    keresztneve(E, KN),
    vezetekneve(Sz, KN).

erdemes2(V-K) :-
    szuloje(V-K, K-).

erdemes3(E) :-
    keresztneve(E, KN),
    vezetekneve(Sz, KN),
    szuloje(E, Sz).

erdemes4(E) :-
    E = _-K,
    szuloje(E, K-).

| ?- spy szuloje, erdemes3(E).
+ 1 Call: szuloje(.965-.951-.951-.978) ?

```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

4. előadás (logikai programozás)

Típusok Prologban

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

```

% :- type név1 == {vnév - knév}.
% egy név1 típusú kifejezés az egy - struktúra
% vnév és knév típusú argumentumokból.
% név1 = {v-k | v ∈ vnév, k ∈ knév }
% :- type vnév == atom.
% egy vnév típusú kifejezés az egy atom
% :- pred szuloje(név1, név1).
% szuloje mindkét argumentuma név1 típusú.
szuloje(szabo-laszlo, szabo-gyorgy). %...

```

Rekurzívan definiált típusok

```

% :- type név2 == atom \/
% {név2 - név2}.
% két név2-ből álló - /2 struktúra
% :- pred reszneve(név2, név2).
% reszneve argumentumai név2 típusúak.
% reszneve(Osszetett, Resz): Osszetett név része a Resz név.
reszneve(Nev, Nev).
reszneve(EIso-, Nev) :-
    reszneve(EIso, Nev).
reszneve(_-Masodik, Nev) :-
    reszneve(Masodik, Nev).

```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

5. előadás (logikai programozás)

A Prolog adatfoglalma — az egyesítési algoritmus

A Prolog adatfoglalma: a Prolog kifejezés

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

Adatstruktúrák szétszedése, összerakása: egyesítési algoritmus

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

Az egyesítési algoritmus

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

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

Egyesítés: a behelyettesítés fogalma

A behelyettesítés

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

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

Legáltalánosabb egyesítő (*mgu* — most general unifier)

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

Egyesítési példák

$A = ose(0, E, E), B = ose(N, ?Géza?, Os)$

- (4.) A és B neve és argumentumszáma megegyezik
 - (a.) $mgu(0, N) = \{N \leftarrow 0\} = \sigma_1$
 - (b.) $mgu(E\sigma_1, ?Géza?) = mgu(E, ?Géza?) = \{E \leftarrow ?Géza?\} = \sigma_2$
 - (c.) $mgu(E\sigma_1\sigma_2, Os) = mgu(?Géza?, Os) = \{Os \leftarrow ?Géza?\} = \sigma_3$
 tehát $mgu(A, B) = \sigma_1 \otimes \sigma_2 \otimes \sigma_3 = \{N \leftarrow 0, E \leftarrow ?Géza?, Os \leftarrow ?Géza?\}$

$A = keresztnéve(V-K, K), B = keresztnéve(E, j\text{ozsi})$

- (4.) A és B neve és argumentumszáma megegyezik
 - (a.) $mgu(V-K, E) = \{E \leftarrow V-K\} = \sigma_1$
 - (b.) $mgu(K\sigma_1, j\text{ozsi}) = mgu(K, j\text{ozsi}) = \{K \leftarrow j\text{ozsi}\} = \sigma_2$
 tehát $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{E \leftarrow V-j\text{ozsi}, K \leftarrow j\text{ozsi}\}$

Egyesítési példák a gyakorlatban

```

| ?- kis-kovacs-bela = X-Y.
   X = kis-kovacs, Y = bela ? ;
   no
| ?- kis-kovacs-bela = kis-X.
   no
| ?- f(X, 3/Y-X, Y) = f(U, B-a, 3).
   B = 3/3, U = a, X = a, Y = 3 ?
| ?- f(f(X), U+2*2) = f(U, f(3)+Z).
   U = f(3), X = 3, Z = 2*2 ?
| ?- ose(0, V-jozsi, szabo-K). % = ose(0, E, E).
   K = jozsi, V = szabo ?
| ?- keresztneve(szabo-(eva-maria), N-maria). % = keresztneve(_K, K).
   N = eva ?

```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

E. előadás (logikai programozás)

A Prolog adatfoglalma 67

Változót tartalmazó kifejezések — végtelen választás veszélye

```

| ?- keresztneve(E, K).
   E = _A-K ? ; no
| ?- keresztneve(E, K), szuloje(E, K-K2).
   E = szabo-laszlo, K = laszlo, K2 = amalia ? ; no
| ?- szuloje(E, K-K2), keresztneve(E, K).
   E = szabo-laszlo, K = laszlo, K2 = amalia ? ; no
| ?- szuloje(V-K, E), reszneve(E, K).
   V = szabo, K = laszlo, E = laszlo-amalia ? ;
   V = szabo, K = amalia, E = laszlo-amalia ? ; no
| ?- reszneve(E, K), szuloje(V-K, E).
   V = szabo, K = laszlo, E = laszlo-amalia ? ;
~C
Prolog interruption (h for help)? a
{Execution aborted}
| ?- reszneve(E, K).
   K = E ? ;
   E = K-_A ? ;
   E = K-_A-_B ?

```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

E. előadás (logikai programozás)

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 költéses, ezért alaphelyzetben nem alkalmazzák.
- Szabványos eljárásként rendelkezésre áll: `unify_with_occurs_check/2`
- Kiterjesztés (pl. SICStus): az előfordulás-ellenőrzés elhagyása miatt keletkező ciklikus kifejezések tisztességes kezelése.

```

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

```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

E. előadás (logikai programozás)

PÉLDA — ÚTVONALKERESÉS

Az útvonalkeresési feladat

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

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

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

Deklaratív programezés, BME, 2001 tavaszi félév

E. eloadás (logikai programozás)

Példa — útvonalkeresés 71

Körmentes út keresése

A köök kizárására gyűjtenünk kell a már érintett városokat. A gyűjtő adastruktúra legyen pl. Honnan-Közben1-Közben2-....

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

% útvonal_2(N, A, B, K, H): A és B között van pontosan
% N szakaszból álló körmentes, K elemeln át nem menő H hosszú út.
útvonal_2(0, Hová, Hová, -, 0).
útvonal_2(N, Honnan, Hová, Kázártak, H) :-
    N > 0, M1 is N-1,
    útszakasz(Honnan, Közben, H1),
    \+ reszemeve(Kázártak, Közben),
    útvonal_2(M1, Közben, Hová, Kázártak-Közben, H2),
    H is H1+H2.
```

Deklaratív programezés, BME, 2001 tavaszi félév

E. eloadás (logikai programozás)

Az útvonalkeresési feladat — folytatás

% útvonal(N, A, B, H): A és B között van (pontosan)
% N szakaszból álló útvonal, amelynek összhossza H.
útvonal(0, Kezdet, Kezdet, 0).
útvonal(N, Kezdet, Cél, H) :-

```
    N > 0,
    M1 is N-1,
    útszakasz(Kezdet, Közben, H1),
    útvonal(M1, Közben, Cél, H2),
    H is H1+H2.

| ?- útvonal(2, 'Párizs', Hová, H).
    H = 1500, Hová = 'Berlin' ? ;
    H = 2530, Hová = 'Párizs' ? ;
    H = 1510, Hová = 'Budapest' ? ;
no
| ?-
```

Deklaratív programezés, BME, 2001 tavaszi félév

E. eloadás (logikai programozás)

Példa — útvonalkeresés 72

A meghívásúlásos negáció (NF — Negation by Failure)

A $\backslash +$ Hívás beépített meta-eljárás (vö. $\backslash -$ — nem bizonyítható)

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

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

Deklaratív programezés, BME, 2001 tavaszi félév

E. eloadás (logikai programozás)

A példa nyomonkövetése

```
| ?- spy_reszneve, átvonal_2(2, 'Párizs', Hová, H).
{Time debugger will first zip -- showing spypoints (zip)}
{Plain spypoint for user:reszneve/2 added, BID=1}
+ 1 1 Call: rezszneve('Párizs', 'Bécs') ? 1
+ 1 1 Fail: rezszneve('Párizs', 'Bécs') ? 1
+ 7 2 Call: rezszneve('Párizs', 'Bécs', 'Berlin') ? 1
+ 8 3 Call: rezszneve('Párizs', 'Berlin') ? 1
+ 8 3 Fail: rezszneve('Párizs', 'Berlin') ? 1
+ 9 3 Call: rezszneve('Bécs', 'Berlin') ? 1
+ 9 3 Fail: rezszneve('Bécs', 'Berlin') ? 1
+ 7 2 Fail: rezszneve('Párizs', 'Bécs', 'Berlin') ? 1
H = 1900, Hová = 'Berlin' ? ;
+ 13 2 Call: rezszneve('Párizs', 'Bécs', 'Párizs') ? 1
+ 14 3 Call: rezszneve('Párizs', 'Párizs') ? 1
+ 14 3 Exit: rezszneve('Párizs', 'Párizs') ? 1
?-
13 2 Exit: rezszneve('Párizs', 'Bécs', 'Párizs') ? 1
+ 16 2 Call: rezszneve('Párizs', 'Bécs', 'Budapest') ? n
H = 1510, Hová = 'Budapest' ?
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

6. előadás (logikai programozás)

Példa — átvonalkeresés

75

Probléma a gyűjtővel — általános gyűjtő-fogalom

```
% bfa_resze(Bfa, Resz): Bfa_részre Resz.
bfa_resze(Bfa, Bfa).
bfa_resze(Balfa-, Resz) :- bfa_resze(Balfa, Resz).
bfa_resze(_Jobbfa, Resz) :- bfa_resze(Jobbfa, Resz).
```

Milyen típusú adatokat kezel a fenti `bfa_resze`?

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

Deklaratív programozás, BMEF, 2001 tavaszi félév

6. előadás (logikai programozás)

Körmentes út keresése — probléma a gyűjtő adatstruktúrával

```
% rezszneve(Usszetett, Resz): Usszetett név része a Resz név.
rezszneve(Nev, Nev).
rezszneve(Eliso-, Nev) :- rezszneve(Eliso, Nev).
rezszneve(_Masodik, Nev) :- rezszneve(Masodik, Nev).

jÁrat(Kál, kál-kápolna, 20).
jÁrat(hatvan, kál, 30).

| ?- átvonal_2(2, hatvan, kál-kápolna, H).
H = 50 ? ; no

| ?- átvonal_2(2, kál-kápolna, hatvan, H).
no

| ?- spy_reszneve, átvonal_2(2, kál-kápolna, hatvan, H).
+ 1 1 Call: rezszneve(kál-kápolna, kál) ?
+ 2 2 Call: rezszneve(kál, kál) ?
+ 2 2 Exit: rezszneve(kál, kál) ?
?-
1 1 Exit: rezszneve(kál-kápolna, kál) ?
no

| ?-
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

6. előadás (logikai programozás)

Példa — átvonalkeresés

76

Probléma a gyűjtő adatstruktúrával — megkülönböztetett ünők

Bináris fa megkülönböztetett ünővel

```
% :- type bfa(T) == {bfa(T) - bfa(T)} \ve {level(T)}.
% egy szintaktikus egyszerűsítést bevezetve:
% :- type bfa(T) --> bfa(T) - bfa(T) ; level(T).
% Egy T-kbóli álló bfa az vagy két ilyenből álló - /2, struktúra
% vagy egy level/1 struktúrába csomagolt T típusú adat.
```

„Ünőris” vagyis lineáris fa (nekünk ez is elég)

```
% :- type gyuitemeny(T) --> gyuitemeny(T)-T ; semmi.
eleme(Gy-E, E).
eleme(Gy-, E) :- eleme(Gy, E).

| ?- eleme(semmi-1-2, E).
E = 2 ? ; E = 1 ? ; no

| ?- eleme(semmi-(kál-kápolna)-hatvan, kál).
no

| ?- eleme(semmi-(kál-kápolna)-hatvan, X).
X = hatvan ? ;
X = kál-kápolna ? ; no
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

6. előadás (logikai programozás)

Körmentes út keresése — javított megoldás

```
% útvonala_3(N, A, B, H): A és B között van (pontosan)
% N szakaszból álló körmentes útvonala, amelynek összhossza H.
útvonala_3(N, Honnan, Hová, H) :-
    útvonala_3(N, Honnan, Hová, semmi-Honnan, H).

% útvonala_3(N, A, B, K, H): A és B között van pontosan
% N szakaszból álló körmentes, K elemein át nem menő H hosszú út.
útvonala_3(0, Hová, Hová, _, 0).
útvonala_3(N, Honnan, Hová, Kizártak, H) :-
    N > 0, N1 is N-1,
    útszakasz(Honnan, Közben, H1),
    \+ elme(Kizártak, Közben),
    útvonala_3(N1, Közben, Hová, Kizártak-Közben, H2), H is H1+H2.

| ?- útvonala_3(2, kál-kápolna, hatvan, H).
H = 50 ? ; no
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

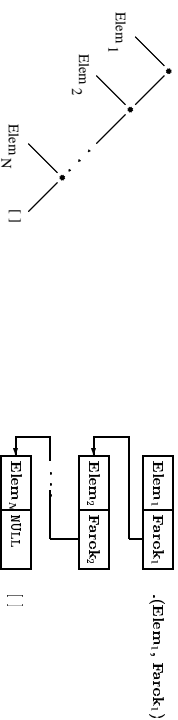
6. előadás (logikai programozás)

A Prolog lista 78

A Prolog lista-fogalma

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

A listák faststruktúra alakja és megvalósítása



Deklaratív programozás, BMEF, 2001 tavaszi félév

6. előadás (logikai programozás)

LISTÁK PROLOGBAN

A Prolog lista 80

Listák jelölése — szintaktikus édesítőszerek

- `[Fej|Farok] ≡ .(Fej, Farok)`
 - `[Elem1, Elem2, ..., ElemN|Farok] ≡ [Elem1|[Elem2|[... [ElemN|Farok] ...]]]`
 - `[Elem1, Elem2, ..., ElemN] ≡ [Elem1, Elem2, ..., ElemN|[]]`
- ```
| ?- [1,2] = [X|Y]. => X = 1, Y = [2] ?
| ?- [1,2] = [X,Y]. => X = 1, Y = 2 ?
| ?- [1,2,3] = [X|Y]. => X = 1, Y = [2,3] ?
| ?- [1,2,3] = [X,Y]. => no
| ?- [1,2,3,4] = [X,Y|Z]. => X = 1, Y = 2, Z = [3,4] ?
| ?- L = [1|_], L = [_|_]. => L = [1,2|_A] ? % nyitl végű
| ?- L = .(1,[2,3|[]]). => L = [1,2,3] ?
| ?- L = [1,2].[3,[]]. => L = [1,2,3] ?
| ?- [X|[3-Y|X|Y]] = .(A, [A-B,6]). => A=3, B=[6]/3, X=3, Y=[6] ?
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

6. előadás (logikai programozás)

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

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

Eldöntendő kérdés

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

```
⇒ yes
```

Megválaszolandó kérdések

```
| ?- member(X, [1,2,3]).
```

```
⇒ X = 1 ? ; X = 2 ? ; X = 3 ? ; no
```

```
| ?- member(X, [1,2,1]).
```

```
⇒ X = 1 ? ; X = 2 ? ; X = 1 ? ; no
```

Vegyes használat, listák metszete

```
| ?- member(X, [1,2,3]),
 member(Y, [5,4,3,2,3]).
```

```
⇒ X = 2 ? ; X = 3 ? ; X = 3 ? ; no
```

Lista elemévé tesz, végtelen választás!

```
| ?- member(1, L).
```

```
⇒ L = [1|_A] ? ; L = [_A,1|_B] ? ;
L = [_A,_B,1|_C] ? ; ...
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

6. előadás (logikai programozás)

**Tönmör és minta-kifejezések, lista-minták, nyílt végű listák**

- Tönmör (ground) kifejezés: változót nem tartalmazó kifejezés

- Minta: egy általában nem nem tönmör kifejezés, mindazon kifejezéseket „képviselet”, amelyek belőle változó-behelyettesítéssel előállnak.

- Lista-minta: listát (is) képviseelő minta.

- Nyílt végű lista: olyan lista-minta, amely bármilyen hosszú listát is képviselet.

- Zárt végű lista: olyan lista(-minta), amely egyféle hosszú listát képviselet.

| Zárt v. | Milyen listákat képviselet | Nyílt v. | Milyen listákat képviselet          |
|---------|----------------------------|----------|-------------------------------------|
| [X]     | egyelemű                   | X        | tetszőleges                         |
| [X,Y]   | kételemű                   | [X Y]    | nem üres (legalább 1 elemű)         |
| [X..X]  | két egyforma elemből álló  | [X,Y Z]  | legalább 2 elemű                    |
| [X,1,Y] | 3 elemből áll, 2. eleme 1  | [a,b Z]  | legalább 2 elemű, elemei: a, b, ... |

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

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

Deklaratív programozás, BMEF, 2001 tavaszi félév

6. előadás (logikai programozás)

**member/2 áthalanosítása: select/3**

```
% select(Elem, Lista, Marad): Elemet a Lista-ból elhagyva marad Marad.
select0(Elem, [Elem|Marad], Marad).
select0(Elem, [_|Farok], Marad) :-
 select0(Elem, Farok, Marad).
```

```
% A második klóz tömörbben (logikai stílusban) --- jobbrekurzív!
select(Elem, [Elem|Marad], Marad).
select(Elem, [_|Farok], [X|Marad0]) :-
 select(Elem, Farok, Marad0).
```

```
% a maradék elé tesszük a fejet.
```

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

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

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

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

```
| ?- select(3, [2|L], [1,2,7,3,2,1,8,9,4]).
```

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

Deklaratív programozás, BMEF, 2001 tavaszi félév

6. előadás (logikai programozás)

**Listák összerűzése: az append/3 eljárás**

```
% append(L1, L2, L3): Az L3 lista az L1 és L2 listák elemeinek
% egymás után fűzésével áll elő (jelöljük: L3 = L1@L2).
```

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

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

```
append(L1, L2, L3).
```

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

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

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

```
Ancestors:
```

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

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

```
3 Call: append([], [3,4], L31) ? g
```

```
Ancestors:
```

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

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

```
3 Call: append([], [3,4], L31) ?
```

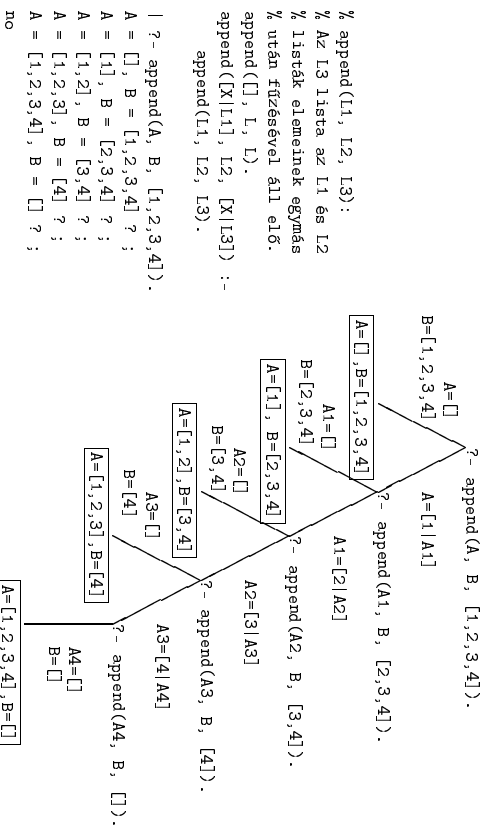
```
3 Exit: append([], [3,4], [3,4]) ?
```

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

Deklaratív programozás, BMEF, 2001 tavaszi félév

6. előadás (logikai programozás)

## Listák szétbontása az append/3 segítségével



Deklaratív Programozás, BMEF, 2001 tavaszi félév

6. előadás (logikai programozás)

## Egy érdekes feladvány

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

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

Kérdés: mennyi idő alatt tudnak leggyorsabban átkelni?

## Prolog megoldás: a következő előadáson!

Deklaratív Programozás, BMEF, 2001 tavaszi félév

6. előadás (logikai programozás)

## Variációk appendre 1. — Három lista összefűzése

```

% L1 @ L2 @ L3 = L123, ahol L1 és L2 adott.
append(L1, L2, L3, L123) :-
 append(L1, L2, L12), append(L12, L3, L123).

```

- Nem hatékony, pl.: `append([1,...,100],[1,2,3],[1], L)` 103 helyett 203 lépés!
- Szétszedésre nem alkalmas — végtelen választási pontot hoz létre (véges a keresési tér, ha az 1. és 3. argumentum legalább egyike zárt végű lista.)

Szétszedésre is alkalmas, hatékony változat

```

% L1 @ L2 @ L3 = L123, ahol vagy L1 és L2 vagy L123 adott(zárt végű).
append(L1, L2, L3, L123) :-
 append(L1, L23, L123), append(L2, L3, L23).

```

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

```

| ?- append([1,2], L23, L). => L = [1,2|L23] ?

```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

6. előadás (logikai programozás)

## Variációk appendre 2. — lista folytonos része

```

% L123 folytonos részlistája L2 (L123 = _ @ L2 @ _).
% L123 adott, L2 ismeretlen.
csublist(L2, L123) :-
 append(_L1, L23, L123),
 append(L2, _L3, L23).

```

% Adott L123-nak folytonos része egy adott L2.

```

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

```

A két változat hatékonyságának összehasonlítása

- $L123 = [0, \underbrace{1, 2, 3, 4}_{x100000}, \dots, 10]$ ,  $L2 = [0, 1, 2, 3, 4, 10]$
- `csublist(L2, L123)`: 570 msec
- `check_csublist(L2, L123)`: 430 msec

Deklaratív Programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

## Mintakeresés append/3-mal

### Párban előforduló elemek

```
% párban(Lista, Elem): A Lista számlistának Elem olyan
% eleme, amely egy ugyanilyen értékű elemmel szomszédos.
párban(L, E) :-
 append(_, [E, E], L).
```

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

### Dadogó részek

```
% dadogó(L, D): D olyan nem üres részlistája L-nek,
% amelyet egy vele megegyező részlista követ.
```

```
dadogó(L, D) :-
 append(_, Farok, L),
 D = [_],
 append(D, Vég, Farok),
 append(D, -, Vég).
```

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

Listák Prologban

81

## append és revapp — listák gyűjtési iránya

### • Prolog megalósítás

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

### • C++ megalósítás

|                                                                                                                                                                                                                                          |                                                                                                                                                                                                 |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>list append(list list1, list list2) { list list3, *lp = &amp;list3;   for (list p=list1; p; p=p-&gt;next)   { list newl = new link(p-&gt;elem);     *lp = newl; lp = &amp;newl-&gt;next;   }   *lp = list2;   return list3; }</pre> | <pre>list revapp(list list1, list list2) { list l = list2;   for (list p=list1; p; p=p-&gt;next)   { list newl = new link(p-&gt;elem);     newl-&gt;next = l; l = newl;   }   return l; }</pre> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

Listák Prologban

89

## Listák megfordítása

### Naïv (négyzetes lépésszámú) megoldás

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

### Lineáris lépésszámú megoldás

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

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

A Lists könyvtár tartalmazza a member/2, select/3, append/3 és reverse/2 eljárások definícióját:

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

Listák Prologban

82

## A select/3 továbbfejlesztése

```
% cserél(X, Xl, Y, Yl): egy X elemet Xl-ben Y-ra cserélve kapjuk Yl-t.
% Deklaratívabban: X ugyanannyiadik eleme Xl-nek, mint Y Yl-nek, és a két
% lista csak ebben az elemben különbözik.
% A végességhez Xl és Yl közül legalább az egyik zárt végű kell legyen.
cserél(X, [X|Xlستا], Y, [Y|Ylستا]) :-
 cserél(X, [Head|Xlستا], Y, [Head|Ylستا]) :-
 cserél(X, Xlستا, Y, Ylستا).
```

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

```
| ?- cserél(X, [1,2,3], 4, L).
L = [4,2,3], X = 1 ? ;
```

```
L = [1,4,3], X = 2 ? ;
L = [1,2,4], X = 3 ? ;
no
```

```
| ?- cserél(X, [1,2,3], Y, L).
L = [Y,2,3], X = 1 ? ;
L = [1,Y,3], X = 2 ? ;
L = [1,2,Y], X = 3 ? ;
no
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

## 2000 tavaszi kis házi feladat

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

```
% szám(Szám, Alap, Jk): A Szám szám Alap alapú számenndszerben vett
% jegyeinek (balról jobbra haladó) listája Jk. (A 0 szám egy jegyből áll.)
szám(0, _, [0]).
```

```
szám(Sz, Alap, Jk) :-
 Sz > 0, szám(Sz, Alap, [], Jk).
```

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

```
% Jeltöltés: LL = L-L0 <----> az LL listát L0 elé fűzve kapjuk L-t.
```

```
szám(0, _, Jk, Jk).
```

```
szám(Sz, Alap, Jk0, Jk) :-
```

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

```
 szám(Sz1, Alap, [UtsoJegy|Jk0], Jk).
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

Listák Prologban

95

## 2000 tavaszi kis házi feladat — egyszerűsítés

fszám/3 egyszerűsíthető

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

- fszám(Sz, A, [], Jk)  $\Rightarrow$  fszám12(Sz, A, Jk)

```
% fszám(Szám, Alap, Jegyek): A Szám >= 0 szám Alap > 1 alapú
% számenndszerben jobbról balra vett jegyeinek listája Jegyek.
fszám1(0, _, [0]).
```

```
fszám1(Sz, Alap, Jk) :-
```

```
 Sz > 0, fszám12(Sz, Alap, Jk).
```

```
% fszám12(Szám, Alap, Jk): A Szám >= 0 szám Alap > 1 alapú
% számenndszerben jobbról balra vett jegyeinek listája Jk.
fszám12(0, _, []).
```

```
fszám12(Sz, Alap, [UtsoJegy|Jk]) :-
```

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

```
 fszám12(Sz1, Alap, Jk).
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

## 2000 tavaszi kis házi feladat — számjegyek fordított sorrendben

```
% fszám(Sz, A, Jk): Az Sz szám A alapú fordított jegylistája Jk.
fszám(0, _, [0]).
fszám(Sz, Alap, Jk) :- Sz > 0, fszám(Sz, Alap, [], Jk).
```

```
% fszám(Sz, A, Jk0, Jk): Az Sz szám A alapú fordított jegylistája Jk-Jk0.
fszám(0, _, Jk, Jk).
```

```
fszám(Sz, Alap, Jk0, [UtsoJegy|Jk]) :-
```

```
 Sz > 0, Sz1 is Sz//Alap, UtsoJegy is Sz mod Alap,
 fszám(Sz1, Alap, Jk0, Jk).
```

## A kétféle irányú gyűjtés összehasonlítása

```
fszám(0, _, Jk, Jk).
```

```
fszám(Sz, A, Jk0, [U|Jk]) :-
```

```
 Sz > 0, Sz1 is ..., Sz > 0, Sz1 is ...,
```

```
 U is ...,
```

```
 fszám(Sz1, A, Jk0, Jk).
```

```
szám(0, _, Jk, Jk).
```

```
szám(Sz, A, Jk0, Jk) :-
```

```
 Sz > 0, Sz1 is ..., U is ...,
```

```
 szám(Sz1, A, [U|Jk0], Jk).
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

Listák Prologban

96

## Körmentes út keresése — megoldás listák használatával

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

```
% útvonala_4(N, A, B, Út, H): A és B között van (pontosan)
```

```
% N szakaszból álló körmentes Út útvonala, amelynek összhossza H.
```

```
útvonala_4(N, Honnan, Hová, Út, H) :-
```

```
 útvonala_4(N, Honnan, Hová, [Honnan], Út, H).
```

```
% útvonala_4(N, A, B, K, Út, H): A és B között van pontosan
```

```
% N szakaszból álló körmentes, K elemű út nem menő H hosszú Út út.
```

```
útvonala_4(0, Hová, Hová, [Kizártak], Út, 0) :-
```

```
 reverse(Kizártak, Út).
```

```
útvonala_4(N, Honnan, Hová, Kizártak, Út, H) :-
```

```
 N > 0, N1 is N-1,
```

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

```
 \+ member(Közben, Kizártak),
```

```
 útvonala_4(N1, Közben, Hová, [Közben|Kizártak], Út, H2), H is H1+H2.
```

```
| ?- útvonala_4(2, 'Párizs', _, Út, H).
```

```
 H = 1900, Út = ['Párizs', 'Bécs', 'Berlin'] ? ;
```

```
 H = 1510, Út = ['Párizs', 'Bécs', 'Budapest'] ? ; no
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)



## Súlyozott gráf ábrázolása állítással

### A gráf ábrázolása

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

### Típus-definíció

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

### Példa

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

Deklaratív Programozás, BME, 2001 tavaszi félév

7. előadás (logikai programozás)

## Ismétlődésmentes útvonal keresése listával ábrázolt gráfban

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

% útvonal_5(N, G, A, B, L, H): A G gráfban van egy A-ból
% B-be menő N szakaszból álló L út, melynek összhossza H.
útvonal_5(0, _Gráf, Hová, [Hová], 0).
útvonal_5(N, Gráf, Honnan, Hová, [Honnan|Út], H) :-
 N > 0, N1 is N-1,
 select(Él, Gráf, Gráf1),
 Él_véggpontok_hossz(Él, Honnan, Közben, H1),
 útvonal_5(N1, Gráf1, Közben, Hová, Út, H2),
 H is H1+H2.

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

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

Deklaratív Programozás, BME, 2001 tavaszi félév

7. előadás (logikai programozás)

## A PROLOG SZINTAXIS

### A Prolog szintaxis összefoglalása

A Prolog szintaxis 100

#### A Prolog szintaxis alapelvei

- Minden programelem kifejezés!
- A szükséges összekötő jelek (' ', ;, :- ->): szabványos operátorok.
- A beolvasott kifejezést funktoara alapján osztályozzuk:
  - *kérdés:* ?- *Cél*.
  - *Cél* lefuttatja, és a változó-behelyettesítéseket kiírja.
  - *parancs:* :- *Cél*.
- *A Cél* csendben lefuttatja. Különféle deklarációkat paramcsként helyezhetünk el a programban.
  - *szabály:* *Fej* :- *Törzs*.
  - *A szabályt felveszi a programba.*
  - *nyelvtani szabály:* *Fej* -> *Törzs*.
- *Prolog szabálytá alakítja és felveszi (lásd a DCC nyelvtanokat).*
- *tényállítás:* Minden egyéb kifejezés.
  - Üres törzsű szabályként felveszi a programba.

Deklaratív Programozás, BME, 2001 tavaszi félév

7. előadás (logikai programozás)

## A Prolog nyelv-változatok

### A SICStus rendszer két üzem módja

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

Deklaratív programozás, BNF, 2001 tavaszi félév 7. előadás (logikai programozás)

## Kifejezések szintaxisa

```

<programelemen> ::= <kifejezés 1200> <záró-pont>
<kifejezés N> ::=
 | <op N fx> <köz> <kifejezés N-1>
 | <op N fy> <köz> <kifejezés N>
 | <kifejezés N-1> <op N xfx> <kifejezés N-1>
 | <kifejezés N-1> <op N xfy> <kifejezés N>
 | <kifejezés N> <op N yfx> <kifejezés N-1>
 | <kifejezés N-1> <op N xfx>
 | <kifejezés N> <op N yf>
 | <kifejezés N-1>
<kifejezés 1000> ::= <kifejezés 999> , <kifejezés 1000>
<kifejezés 0> ::=
 <név> <argumentumok>
 { A <név> és a <közvetlenül egymás után állj}
 <kifejezés 1200> | { <kifejezés 1200> }
 <lista> | <füzér>
 <név> | <szám> | <változó>

```

Deklaratív programozás, BNF, 2001 tavaszi félév 7. előadás (logikai programozás)

## Kifejezések szintaxisa — kétszintű nyelvtanok

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

```

<kifejezés> ::= <tag>
 | <kifejezés> <additív művelet> <tag>
<tag> ::=
 | <tényező>
 | <tag> <multiplikatív művelet> <tényező>
<tényező> ::= <szám> | <azonosító> | (<kifejezés>)

```
- Ugyanez kétszintű nyelvtannal:
 

```

<kifejezés> ::= <kif 2>
<kif N> ::= <kif N-1>
 | <kif N> <N prioritású művelet> <kif N-1>
<kif 0> ::= <szám> | <azonosító> | (<kif 2>)
{az additív ill. multiplikatív műveletek prioritása 2 ill. 1}

```

Deklaratív programozás, BNF, 2001 tavaszi félév 7. előadás (logikai programozás)

## Kifejezések szintaxisa — folytatás

```

<op N 'I'> ::= <név> {feltéve, hogy <név> N prioritású és
 'I' típusú operátornak lett deklarálva}
<argumentumok> ::= <kifejezés 999>
 | <kifejezés 999> , <argumentumok>
<lista> ::=
 | [<listakif>]
 | <listakif> ::=
 | <kifejezés 999> , <listakif>
 | <kifejezés 999> | <kifejezés 999>
<szám> ::= <előjeltelen szám>
 | + <előjelteles szám>
 | - <előjelteles szám>
<előjelteles szám> ::= <természetes szám>
 | <lebegőpontos szám>

```

Deklaratív programozás, BNF, 2001 tavaszi félév 7. előadás (logikai programozás)

## Kifejezések szintaxisa — megjegyzések

- A `{ kifejezés N }`-ben `{köz}` csak akkor kell ha az öt követő kifejezés nyitó-záró\_jellel kezdődik.
- A `{ kifejezés }` azonos a `{(kifejezés)}` struktúrával, ez pl a DCG nyelvtanoknál hasznos.
- Egy `{füzér}` "jelek közé zárt karakter sorozat, általában a karakterek kódjainak listájával azonos.
- ! ? - op(500, fx, succ).
- yes
- | ? - display(succ(1,2)), nl, display(succ(1,2)).
- succ(, (1,2))
- succ(1,2)
- yes
- | ? - write('baba').
- [98,97,98,97]

Deklaratív programozás, BMEF, 2001 tavaszi félév 7. előadás (logikai programozás)

## A Prolog lexikai elemei 2.

### (természetes szám)

- (decimális) számjegysorozat;
- 2, 8 ill. 16 alapú számrendszerben felírt szám, ilyenkor a számjegyeket rendre a 0b, 0o, 0x karakterekkel kell prefixálni (csak iso módban)
- karakterkód-konstans `0'c` alakban, ahol `c` egyetlen karakter

### (lebegőpontos szám)

- mindenképpen tartalmaz tízedespontot
- mindkét oldalon legalább egy (decimális) számjeggyel
- `e` vagy `E` betűvel jelzett esetleges exponens

Deklaratív programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

## A Prolog lexikai elemei 1. (ismétlés)

### (név)

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

Deklaratív programozás, BMEF, 2001 tavaszi félév 7. előadás (logikai programozás)

## Megjegyzések és formázó-karakterek

### Megjegyzések (comment)

- A `%` szövegtől a sor végéig
- A `/*` jelpártól a legközelebbi `*/` jelpárig.

### Formázó elemek

- szököz, új sor, tabulátor, stb. (nem látható karakterek)
- megjegyzés

### A programszöveg formázása

- formázó elemek (szököz, új sor, stb.) szabadon elhelyezhetők;
- kivétel: struktúrakifejezés neve után nem szabad;
- prefix operátor és `(közé kötelező)`;
- (záró-pont): egy `.` karakter amit egy formázó elem követ.

Deklaratív programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

## Típusok leírása Prologban

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

```
{ str(T_1 , ..., T_n) } \equiv { str(e_1 , ..., e_n) | $e_1 \in T_1$, ..., $e_n \in T_n$ }, $n \geq 0$
{ személy(atom, atom, int) } az olyan személy/3 funktorú struktúrák halmaza,
amelyben az első két argumentum atom, a harmadik egész.
```
- **Típusok, mint halmazok ünije:**a képezhető a  $\vee$  operátorral.

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

```
% :- type t1 == {atom-atom} \vee atom, % :- type ember == {ember-atom} \vee {semmi}.
```
- **Megkülönböztetett ünio:** csupa különböző funktorú összetett típus ünije.a.
Egyszerűsített jelölés:

```
:- type T == { S_1 } \vee ... \vee { S_n }. \Rightarrow :- type T ----> S_1 ; ... ; S_n .
% :- type ember ----> ember-atom; semmi.
% :- type egészlista ----> [] ; [int|egészlista].
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

Típusok Prologban 111

## Típusok leírása Prologban — folytatás

- **Paraméteres típusok — példák**

```
% :- type list(T) ----> [] ; [T|list(T)]. % T típusú elemekből álló lista.
% :- type pair(T_1 , T_2) ----> T1 - T2. % egy '-', nevű kétargumentumú struktúra,
% % első argumentuma T_1 , a második T_2 típusú.
% :- type assoc_list($KeyT$, $ValueT$)
% == list(pair($KeyT$, $ValueT$)). % KeyT és ValueT típusú párokból álló lista.
% :- type szótár == assoc_list(szó, szó).
% :- type szó == atom.
```
- **Típusdeklarációk szintaxisa**

```
<típusdeklaráció> ::= <típuselnevezés> | <típuskonstrukció>
<típuselnevezés> ::= :- type <típusazonosító> == <típusleírás>.
<típuskonstrukció> ::= :- type <típusazonosító> ----> <megkülönb. ünio>.
<megkülönb. ünio> ::= <konstruktor> ; ...
<konstruktor> ::= <névkonstans> | <struktúránév> (<típusleírás>, ...).
<típusleírás> ::= <típusnév> | <típusváltozó> |
<típusleírás> \vee <típusleírás> |
{ <típusnév> (<típusleírás>, ...)}
<típusazonosító> ::= <típusnév> | <típusnév> (<típusváltozó>, ...)
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

## Típusok leírása Prologban

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

```
{ str(T_1 , ..., T_n) } \equiv { str(e_1 , ..., e_n) | $e_1 \in T_1$, ..., $e_n \in T_n$ }, $n \geq 0$
{ személy(atom, atom, int) } az olyan személy/3 funktorú struktúrák halmaza,
amelyben az első két argumentum atom, a harmadik egész.
```
- **Típusok, mint halmazok ünije:**a képezhető a  $\vee$  operátorral.

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

```
% :- type t1 == {atom-atom} \vee atom, % :- type ember == {ember-atom} \vee {semmi}.
```
- **Megkülönböztetett ünio:** csupa különböző funktorú összetett típus ünije.a.
Egyszerűsített jelölés:

```
:- type T == { S_1 } \vee ... \vee { S_n }. \Rightarrow :- type T ----> S_1 ; ... ; S_n .
% :- type ember ----> ember-atom; semmi.
% :- type egészlista ----> [] ; [int|egészlista].
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

Típusok Prologban 112

## Predikátum-deklarációk

- **Predikátumtípus-deklaráció**

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

```
:- pred member(T , list(T)).
:- pred append(list(T), list(T), list(T)).
```
- **Predikátummód-deklaráció (Nem kötelező, több is megadható.)**

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

```
:- mode append(in, in, in). % ellenőrzésére
:- mode append(in, in, out). % két lista összeállítására
:- mode append(out, out, in). % egy lista szétszerelésére
```
- **Vegyes típus- és móddeklaráció**

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

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

## A HÍD FELADVÁNY

A híd feladvány 115

```

Az állapotátmenet
% útszakasz(áll0, áll1, idő): áll0-ból egy lépésben áll1-be lehet jutni idő alatt.
% :- pred útszakasz(áll::in, áll::out, int::out).
útszakasz(Innen-Helyzet0, Ide-Helyzet, Idő) :-
 másik(Innen, Ide), % A lámpa az Innen oldalon van, Ide a túloldal.
 cserél(E1-Innen, Helyzet0, E1-Ide, Helyzet), % E1 megy át
 Idő = E1.
útszakasz(Innen-Helyzet0, Ide-Helyzet, Idő) :-
 másik(Innen, Ide),
 cserél(E1-Innen, Helyzet0, E1-Ide, Helyzet1), % E1 átmegey
 cserél(E2-Innen, Helyzet1, E2-Ide, Helyzet), % E2 átmegey
 E1 > E2, Idő = E1.

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

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

```

Deklaratív programozás, BMEF, 2001 tavaszi félév 7. előadás (logikai programozás)

### A feladat

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

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

Kérdés: mennyi idő alatt tudnak leggyorsabban átkelni?

### Az adatstruktúrák

```

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

```

A híd feladvány 116

```

Pufftatás
% Át lehet menni a bal véghelyzetről a jobb véghelyzetre
% Hossz Idő alatt az út állapotlistán keresztül.
% :- pred átmegey(list(áll)::out, int::out).
 átmegey(Út, Hossz) :-
 véghelyzet(bal, Kezd),
 véghelyzet(jobb, Vég),
 between(1, 10, N),
 útvonala_4(N, Kezd, Vég, Út, Hossz).

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

```

Deklaratív programozás, BMEF, 2001 tavaszi félév 7. előadás (logikai programozás)

## 2. megoldás — a keresési tér korlátozása, lépések gyűjtése

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

% útszakasz(áll0, áll1, Idő, Lépcső): Egy Lépcső lépéssel Idő alatt az áll0
% állapotról az áll1 állapotba lehet jutni.
% :- pred útszakasz(áll::in, áll::out, int::out, lépcső::out).
útszakasz(Innen-Helyzet0, Ide-Helyzet, Idő, El>Ide) :-
 másikk(Innen, Ide),
 cserél(E1-Innen, Helyzet0, E1-Ide, Helyzet),
 Idő = E1.
útszakasz(Innen-Helyzet0, Ide-Helyzet, Idő, El+El2>Ide) :-
 % E1+E2 megy át.
 másikk(Innen, Ide),
 cserél(E1-Innen, Helyzet0, E1-Ide, Helyzet1),
 cserél(E2-Innen, Helyzet1, E2-Ide, Helyzet),
 E1 > E2, Idő = E1.
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

A híd feladvány 118

## 1. kis házi feladat

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

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

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

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

Pontérték: 1 plusz pont

Beadási határidő: 2001 március 19, 24:00

Beadás módja: a honlapon meghirdetendő módon.

Deklaratív Programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

```
% Lépések-kel át lehet menni a bal véghelyzetről a jobb véghelyzetbe
% Hossz idő alatt, ahol Hossz < Max.
% :- pred átmegy(int::in, list(lépés)::out, int::out).
átmegy(Max, Lépések, Hossz) :-
 véghelyzet(bal, Kezd), véghelyzet(jobb, Vég),
 útvonala_6(Kezd, Vég, Max, Lépések, Nyereség),
 Hossz is Max - Nyereség.
```

```
% útvonala_6(A, B, Max, Lk, Myer): A és B között van egy Max-nál
% Myer-rel rövidebb út (Myer > 0), amelyek lépéssorozatá Lk.
% :- pred útvonala_6(áll::in, áll::in, int::in, list(lépés)::out, int::out).
útvonala_6(Hová, Hová, Max, [], Max) :- Max > 0.
útvonala_6(Honnan, Hová, Max0, [Lép|Lépl], Myer) :-
 Max0 > 0,
 útszakasz(Honnan, Közben, H1, Lép),
 Max1 is Max0-H1,
 útvonala_6(Közben, Hová, Max1, Lépl, Myer).
```

```
| ?- átmegy(190, Lk, H).
H = 170, Lk = [20+10>jobb, 10>bal, 100+50>jobb, 20>bal, 20+10>jobb] ? ;
H = 170, Lk = [20+10>jobb, 20>bal, 100+50>jobb, 10>bal, 20+10>jobb] ? ;
no
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

A híd feladvány 120

## 1. kis házi feladat — folytatás

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

Példák

```
| ?- zaszlo([piros(a),kek(b)], Z).
no
| ?- zaszlo([zold(c),piros(d),feher(e)], Z).
Z = [piros(d),feher(e),zold(c)] ? ;
no
| ?- zaszlo([piros(f),zold(g),piros(h),feher(i)], Z).
Z = [piros(f),piros(h),feher(i),zold(g)] ? ;
no
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

7. előadás (logikai programozás)

## Az előadássorozat áttekintése

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

## BEVEZETÉS A FUNKCIONÁLIS PROGRAMOZÁSBA

Beverdés 123

### A funkcionális programozás motivációi

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

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

- A deklaratív programozás *időtlen*, nem törődik az idővel.
- Idő  $\rightarrow$  állapot  $\rightarrow$  emlékezet.

Beverdés 124

### A funkcionális programozás rövid története

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

## Az ML (Meta Language) rövid története és jelene

---

### Az ML rövid története

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

### SML-megvalósítások

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

## SML-irodalom (csak angolul)

---

### Forrásművek az előadásokhoz

- Jeffrey D. Ullman: *Elements of ML Programming* (2nd Edition, ML97)  
MIT Press 1997  
<http://www-db.stanford.edu/~ullman/emlp.html>
- Lawrence C. Paulson: *ML for the Working Programmer* (2nd Edition, ML97)  
Cambridge University Press 1996  
<http://www.cl.cam.ac.uk/users/lcp/MLbook/>
- Richard Bosworth: *A Practical Course in Functional Programming Using Standard ML*  
McGraw-Hill 1995

## Információk a funkcionális programozásról

---

### Hálózati információforrások:

#### Comp.Lang.ML FAQ

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

#### Andrew Cumming: A Gentle Introduction to ML

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

#### Stephen Gilmore: Programming in Standard ML '97

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

#### Robert Harper: Programming in Standard ML

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

#### Fox project at CMU

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

## A FÜGGVÉNY FOGALMA ÉS TULAJDONSÁGAI



## A típus és a függvény fogalma

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

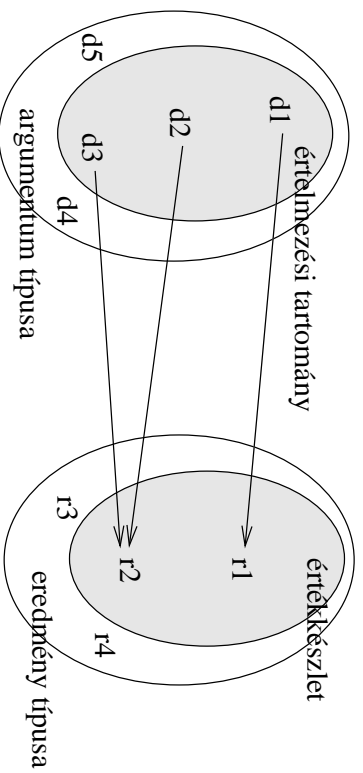
Deklaratív programozás, BMTF, 2001 tavaszi félév

8. előadás (funkcionális programozás)

A függvény fogalma és tulajdonságai

131

## A függvény mint leképezés



Deklaratív programozás, BMTF, 2001 tavaszi félév

8. előadás (funkcionális programozás)

## A függvény mint érték

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

Deklaratív programozás, BMTF, 2001 tavaszi félév

8. előadás (funkcionális programozás)

A függvény fogalma és tulajdonságai

132

## Függvények tulajdonságai és osztályozása

- Parciális függvény: értelmezési tartomány  $\subset$  argumentum típusa  
Figyelem: ez hibák forrása lehet!
- Teljes függvény: értelmezési tartomány = argumentum típusa
- Szűrjektiv függvény: értékkészlet = eredmény típusa
- Nem-szűrjektiv függvény: értékkészlet  $\subset$  eredmény típusa
- Injektiv függvény: a leképezés kölcsönösen egyértelmű
- Az  $f : \alpha \rightarrow \beta$  injektiv függvény inverze:  $f^{-1} : \beta \rightarrow \alpha$
- Bijektiv = injektiv + szűrjektiv, azaz  $f$  bijektiv, ha  $f^{-1}$  teljes függvény

Deklaratív programozás, BMTF, 2001 tavaszi félév

8. előadás (funkcionális programozás)

## Függvények alkalmazása

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

## Két- vagy többargumentumú függvények

- Függvény alkalmazása két- vagy több argumentumra
  1. Az argumentumokat *összetett adatnak* – párnak, rekordnak, listának stb. – tekintjük, pl.  $f(1,2)$ 
    - az  $f$  függvény alkalmazását jelenti az  $(1,2)$  párra.
  2. A függvényt több egymás utáni lépésben alkalmazzuk az argumentumokra, pl.  $f12 \equiv (f1)2$  azt jelenti, hogy
    - az első lépésben az  $f$  függvény alkalmazzuk az 1 értékre, ami egy függvényt ad eredményül,
    - a második lépésben az első lépésben kapott függvényt alkalmazzuk a 2 értékre, így kapjuk meg az  $f12$  függvényalkalmazás (vég)eredményét.
- Infix jelölés:  $x \oplus y \equiv$  az  $\oplus$  függvény alkalmazása az  $(x,y)$  párra mint argumentumra

## Függvények alkalmazása az SML-ben

- Az SML-ben az  $f$  és az  $e$  tetszőleges *név* lehet, amelyeket megfelelően *szeparálni* kell egymástól:  $f\ e$ , vagy  $f(e)$ , vagy  $(f)e$
- Szeparátor: nulla, egy vagy több *formázó* karakter (`␣`, `\t`, `\n` stb.). Nulla db formázó karakter elegendő pl. a (előtt és a ) után.
- A szeparátor a legerősebb balra kötő infix operátor az SML-ben.
- Példák:
 

```
Math.sin 1.00, (Math.cos)Math.pi, round(3.17), 2 + 3, (real)(3 + 2 * 5)
```
- Függvények egy csoportosítása az SML-ben
  - Beépített függvények, pl. `+`, `*` (infix), `real`, `round` (prefix)
  - Könyvtári függvények, pl. `Math.sin`, `Math.cos`, `Math.pi`
  - Felhasználó által definiálható függvények, pl. `terület`, `\`, `head`

## FÜGGVÉNYEK AZ SML-BEN

**SML-példa: Egyszeres Hamming-távolságú ciklikus kód**

- A függvényt *táblázattal* adjuk meg:
 

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

Deklaratív programozás, BMF, 2001 tavaszi félév 8. előadás (funkcionális programozás)

**Értékdeklaráció SML-ben: függvényérték deklarálása**

- Név kötése függvényértékhez
  - `val incMod = fn i => fn n => (i + 1) mod n`
  - `val kovKod = fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00`
- Szintaktikai édesítőszerez
  - `fun incMod n i = (i + 1) mod n`
  - **Figyelni:** `i` és `n` sorrendje megfordult!
    - `fun kovKod 00 = 01`
    - | `kovKod 01 = 11`
    - | `kovKod 11 = 10`
    - | `kovKod 10 = 00`
- Alkalmazásuk argumentumra
  - `incMod 128 111`
  - `kovKod 01`

Deklaratív programozás, BMF, 2001 tavaszi félév

8. előadás (funkcionális programozás)

**SML-példa: modulo  $n$  alapú inkrementálás**

- A függvényt most *algoritmussal* adjuk meg, nem táblázattal
  - `n` nem lehetne változó, túl sok változatot kellene felírni stb.
- `fn i => (i + 1) mod n`
  - az `i` ún. kötött változó, a névtelen függvény argumentuma
  - az `n` ebben a kifejezésben szabad változó, és nincs értéke (!)
  - az `n-et` is le kell kötni mint a függvény argumentumát
- `fn i => fn n => (i + 1) mod n`
- A függvény néhány alkalmazása:
  - `(fn i => (fn n => (i + 1) mod n) 4) 1`
  - `(fn i => (fn n => (i + 1) mod n) 128) 111`
  - `(fn i => (fn n => (i + 1) mod n) 4) ^7`
  - `(fn i => (fn n => (i + 1) mod n) 128) 6..0 - hibás!`

Deklaratív programozás, BMF, 2001 tavaszi félév 8. előadás (funkcionális programozás)

**Fejlesztés**

Legyen *fejlesztés* minden (függvény)érték-deklarációhoz!

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

Deklaratív programozás, BMF, 2001 tavaszi félév

8. előadás (funkcionális programozás)

## Típusok

- Típusok és programozási nyelvek
  - Típus nélküli nyelvek, pl. assembly, LISP, Prolog
  - Gyengén típusos nyelvek, pl. Fortran, Algol, BASIC, C, C++, Pascal
  - Erősen típusos nyelvek, pl. Ada, SML, clean
  - Erős típus: a típusok ( $\sim$  halmazok) diszjunktak (nincs közös elemük)
- Egyszerű SML-típusok
  - int – előjeles egész szám, a  $\mathbb{Z}$  egy részhalmaza
  - word, word8 – előjel nélküli pozitív egész, az  $\mathbb{N}_0$  egy részhalmaza
  - real – előjeles racionális (valós?) szám, a  $\mathbb{Q}$  egy részhalmaza
  - bool, char, order, unit
  - string
- Összetett SML-típusok (példák)
  - rekord
  - lista

Deklaratív programozás, BMF, 2001 tavaszi félév

8. előadás (funkcionális programozás)

## TÍPUSOK ÉS ÉRTÉKEK AZ SML-BEN

Típusok és értékek az SML-ben

143

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

- Függvényértéket így kötöttünk tetszőleges névhez:
 

```
val incMod = fn i => fn n => (i + 1) mod n
```
- Tetszőleges típusú érték köthető tetszőleges névhez:
 

|                              |                          |                      |
|------------------------------|--------------------------|----------------------|
| val harom = 2 + 1            | : int                    |                      |
| val MHz = 94.5               | : real                   |                      |
| val veege = true             | : bool                   | true, false          |
| val kisa = #"a"              | : char                   |                      |
| val palindrom = "ABBA"       | : string                 |                      |
| val kisebb = LESS            | : order                  | LESS, EQUAL, GREATER |
| val kiemSemi = ()            | : unit                   | Egyetlen érték a ()! |
| val rat = {num = 3, den = 4} | : {den : int, num : int} | Mezőnevek ábrécben.  |
| val blista = [2,3,4] @ [3,2] | : int list               |                      |
| val telenek = [0w123, 0wxd]  | : word list              |                      |
- Típusmegkötés:
 

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

Deklaratív programozás, BMF, 2001 tavaszi félév

8. előadás (funkcionális programozás)

## EGYSZERŰSÍTETT SML-SZINTAXIS

## SML-szintaxis: különleges állandó

- Előjeles egész állandó  
Példák: `0` `~0` `4` `~04` `99999` `0xFFFF` `~0x1ff`  
Ellenpéldák: `0.0` `~0.0` `4.0` `1E0` `-317` `0xFFFF` `-0x1ff`
- Valós állandó  
Példák: `0.7` `~0.7` `3.32E5` `3E-7` `~3E-7` `3e-7` `~3e-7`  
Ellenpéldák: `23` `.3` `4.E5` `1E2.0` `1E+7` `1E-7`
- Előjel nélküli egész állandó  
Példák: `0w0` `0w4` `0w99999` `0wFFFF` `0w1ff`  
Ellenpéldák: `0w0.0` `~0w4` `-0w4` `0w1E0` `0wFFFF` `0wFFFF`
- Füzerállandó: `"`-ek között álló nulla vagy több nyomtatható karakter, szóköz vagy `\` jellel kezdődő *escape-szekvencia* (l. a táblázatot a következő lapon).
- Karakterállandó: `#` jellel közvetlenül követő, egykarakteres fügzerállandó.  
Példák: `#"a"` `#"\n"` `#"\z"` `#"\255"` `#"\\"`  
Ellenpéldák: `#"a"` `#c` `#"!"`

Deklaratív programozás, BMTE, 2001 tavaszi félév

9. előadás (funkcionális programozás)

Egyszerűsített SML-szintaxis

147

## SML-szintaxis: név

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

Deklaratív programozás, BMTE, 2001 tavaszi félév

9. előadás (funkcionális programozás)

## SML-szintaxis: escape-szekvenciák

- Escape-szekvenciák  
`\a` Csengőjel (BEL, ASCII 7).  
`\b` Viszsalépés (BS, ASCII 8).  
`\t` Vízszintes tabulátor (HT, ASCII 9).  
`\n` Újsor, soromelés (LF, ASCII 10).  
`\v` Függőleges tabulátor (VT, ASCII 11).  
`\f` Lapdobás (FF, ASCII 12).  
`\r` Koosi-vissza (CR, ASCII 13).  
`\^c` Vezérlő karakter, ahol  $64 \leq c \leq 95$  (`@ ... _`), és `\^c` ASCII-kódja 64-gyel kevesebb `c` ASCII-kódjánál.  
`\ddd` A `ddd` kódú karakter (`d` decimális számjegy).  
`\uxxxx` A `xxxx` kódú karakter (`x` hexadecimális számjegy).  
`\"` Idézőjel (`"`).  
`\\` Hátrahört-vonal (`\`).  
`\f...f` Figyelmen kívül hagyott sorozat. `f...f` nulla vagy több formázókaraktert (szóköz, HT, LF, VT, FF, CR) jelent.

Deklaratív programozás, BMTE, 2001 tavaszi félév

9. előadás (funkcionális programozás)

Egyszerűsített SML-szintaxis

148

## SML-szintaxis: szintaktikai kategóriák (egyszerűsítve)

- A nevek és más azonosítók *szintaktikai kategóriákba* sorolhatók  

|                |                  |                      |      |
|----------------|------------------|----------------------|------|
| <i>vid</i>     | értéknév         | value identifier     | long |
| <i>tyvar</i>   | típusváltozó     | type variable        | long |
| <i>tycon</i>   | típuskonstruktor | type constructor     | long |
| <i>lab</i>     | mezőnév          | record label         |      |
| <i>strid</i>   | strukturánév     | structure identifier | long |
| <i>sigid</i>   | szignatúránév    | signature identifier |      |
| <i>unittid</i> | állománynév      | unit identifier      |      |
- Az *értéknév* tetszőleges név; jelölhet állandó értéket, függvényértéket, adatkonstruktor, kivételkonstruktor. Példák: `pi` `+ sin nil true Match`
- A *típusváltozó* perccjellel kezdődő alfanumerikus név. Példa: `'a`.
- A *típuskonstruktor* tetszőleges név; jelölhet típusállandót vagy típusfüggvényértéket. Példák: `int order $ * -> list`
- A *mezőnév* tetszőleges név vagy (nem 0-val kezdődő) pozitív egész szám.  
Példák: `num` `2`

Deklaratív programozás, BMTE, 2001 tavaszi félév

9. előadás (funkcionális programozás)

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

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

|                           |                    |                      |
|---------------------------|--------------------|----------------------|
| <i>longa</i> ::= <i>x</i> | név                | identifier           |
|                           | <i>longstrid.x</i> | qualified identifier |

Példák:

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

## Struktúra, szignatúra, tárgykódú és forráskódú állományok

Példák

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

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

- A *struktúránév* és a *szignatúránév* a *modulnagylev* fogalomkörébe tartozó tetszőleges nevek.

Példák: `Char` `Int` `List` `TextIO`

- Az *állománynév* a *modulnagylev* fogalomkörébe tartozó tetszőleges olyan név, amelyet az adott operációs rendszer is megenged; forráskódú vagy tárgykódú struktúra- vagy szignatúra-állományt azonosít.
- A *strid* struktúránév a `unitid.no` tárgykódú struktúra-állományra hivatkozik, ahol `unitid` = *strid*. A `unitid.sml` struktúra-állomány fordításakor már léteznie kell a `unitid.ui` tárgykódú szignatúra-állománynak, összeszerkesztésekor pedig már léteznie kell a `unitid.no` tárgykódú struktúra-állománynak.
- A *sigid* szignatúránév a `unitid.ui` tárgykódú szignatúra-állományra hivatkozik, ahol `unitid` = *sigid*. A `unitid.ui` tárgykódú szignatúra-állományt a `unitid.sig` forráskódú szignatúra-állomány lefordításával kell előállítani.

## Függvényjel helyzete és kötése

- Függvényjel helyzete és kötése (általában)
  - Egy függvényjel *prefix*, *infix* vagy *postfix* helyzetű lehet.
  - Az infix helyzetű függvényjelet gyakran *operátornak* nevezik.
  - Egy (infix helyzetű) operátor lehet *asszociatív* vagy *nem-asszociatív*; köthet balra vagy jobbra, vagy semerre. Asszociatív operátor esetén a kötési iránynak nincs jelentősége.
- Infix Prolog-operátor kötése
  - `xfx` = `f` mindkét oldalán `f` csak zárójelben ismétlődhet (`f, nem köt?`),
  - `yfx` = `f` bal oldalán `f` zárójelzés nélkül ismétlődhet (`f „balra köt?”`),
  - `xfy` = `f` jobb oldalán `f` zárójelzés nélkül ismétlődhet (`f „jobbra köt?”`).

## Függvényjel helyzete és kötése az SML-ben

- Kifejezések és típuskifejezések az SML-ben
  - Az SML-ben a szokásos kifejezések mellett vannak *típuskifejezések* is.
  - A függvények *értékekre*, a típusfüggvények *típusokra* alkalmazhatók.
- Függvényjel és típusfüggvényjel helyzete és kötése az SML-ben
  - Függvényjel: *prefixa* vagy *infix*.
  - Típusfüggvényjel: *infix* vagy *postfix*.
  - Az *infix* helyzetű függvényjel és típusfüggvényjel (szokásos néven operátor, ill. típusoperátor) balra vagy jobbra köt, vagy semerre nem köt.
- Típusoperátorok
  - A két infix helyzetű beépített típusoperátor közül a `->` jobbra, a `*` semerre nem köt.
  - A `*` operátornak magasabb a precedenciája, mint a `->` operátornak.
  - A típusoperátoroknak magasabb a precedenciája a többi operátorénál.

Deklaratív programozás, BMF, 2001 tavaszi félév

9. előadás (funkcionális programozás)

Egyszerűsített SML-szintaxis

155

## A beépített operátorok és precedenciájuk az SML-ben

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

`wordint` = `int`, `word`, `word8`.

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

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

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

Deklaratív programozás, BMF, 2001 tavaszi félév

9. előadás (funkcionális programozás)

## Függvényjel helyzete és kötése az SML-ben

- Tetszőleges kétargumentumú függvényjellet lehet meghatározott preferenciájú (infix helyzetű) operátorokként deklarálni az `infix` vagy az `infixr` direktívával.
- Az `infix` balra, az `infixr` jobbra kötő operátort deklarál.
- Egy minősített nevet, vagy egy olyan nevet, amelyet az `op` direktíva előz meg, csak *prefixa* helyzetben lehet alkalmazni.
- A `nonfix` direktíva az (infix helyzetű) operátort tartósan prefix helyzetűvé alakítja. (Az `op` direktíva csak átmenetileg teszi prefix helyzetűvé.)
- Az `idn` tetszőleges név ( $n \geq 1$ ). A `d 0` és `9` közötti számszám, az operátor precedenciája (opcionális, alapértelmezés szerinti értéke 0). Nagyobb szám nagyobb precedenciát jelent (éppen fordítva, mint a Prologban!).
 

|                               |                                                |  |            |  |                    |
|-------------------------------|------------------------------------------------|--|------------|--|--------------------|
| <code>infix &lt;d&gt;</code>  | <code>id<sub>1</sub> ... id<sub>n</sub></code> |  | balra köt  |  | binds to the left  |
| <code>infixr &lt;d&gt;</code> | <code>id<sub>1</sub> ... id<sub>n</sub></code> |  | jobbra köt |  | binds to the right |
| <code>nonfix</code>           | <code>id<sub>1</sub> ... id<sub>n</sub></code> |  | prefix     |  | prefix             |

Deklaratív programozás, BMF, 2001 tavaszi félév

9. előadás (funkcionális programozás)

Egyszerűsített SML-szintaxis

156

## SML-szintaxis: nemterminális szimbólumok, nyelvtani jelölések

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

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

Deklaratív programozás, BMF, 2001 tavaszi félév

9. előadás (funkcionális programozás)

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

- Kifejezés (*exp*: expression)

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

- Példák:

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

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

- Atomi kifejezés (*atexp*: atomic expression)

|                                                       |                      |                     |
|-------------------------------------------------------|----------------------|---------------------|
| (10) <i>atexp</i> ::= <i>scon</i>                     | különleges állandó   | special constant    |
| (11) <i>&lt;op&gt; longuid</i>                        | értéknév             | value identifier    |
| (12) <i>{&lt;exprow&gt;}</i>                          | rekord               | record              |
| (13) <i># lab</i>                                     | rekordszelektor      | record selector     |
| (14) <i>(exp<sub>1</sub> , exp<sub>2</sub>)</i>       | pár                  | pair                |
| (15) <i>()</i>                                        | nullas               | 0-tuple             |
| (16) <i>[exp<sub>1</sub> , ... , exp<sub>n</sub>]</i> | lista, $n \geq 0$    | list, $n \geq 0$    |
| (17) <i>(exp)</i>                                     | kifejezés zárójelben | parenthesized expr. |

- Példák:

```
1. 12, #"Z", 0w123
Math.pi, false, Math.sin, sin
#den {num=1, den=2}
(2, 3.5), (), [1, 2, 3]
```

vö. (10)

vö. (11)

vö. (12), (13), (18)

vö. (14), (15), (16)

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

- Infix kifejezés (*infexp*: infix expression)

|                                                     |                     |                     |                      |
|-----------------------------------------------------|---------------------|---------------------|----------------------|
| (6) <i>infexp</i> ::= <i>appexp</i>                 |                     |                     |                      |
| (7) <i>infexp<sub>1</sub> id infexp<sub>2</sub></i> | infix alkalmazás    | infix application   |                      |
| (8) <i>appkativ kifejezés (appexp)</i>              | appkativ kifejezés  | appkativ expression |                      |
| (9) <i>appexp</i> ::= <i>atexp</i>                  | <i>appexp atexp</i> | (prefix) alkalmazás | (prefix) application |

- Példák:

```
3 + 4;
Real.toString 3.56;
Int.toString(round 3.56);
vö. (7)
vö. (9)
vö. (9), (17)
```

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

- Kifejezősor (*exprow*: expression row)

(18) *exprow* ::= *lab = exp* < , *exprow* >

- Klózsorozat (*match*)

(19) *match* ::= *mrule* < | *match* >

- Klóz (*mrule*: match rule)

(20) *mrule* ::= *pat => exp*

- Példák:

```
num=1, den=2
00 => 01 | 01 => 11 | 11 => 10 | 10 => 00
vö. (18)
vö. (19), (20)
```



## SML-szintaxis: deklarációk és kötések

• Deklaráció (*dec*: declaration)

|                     |                                                                                             |                              |                         |
|---------------------|---------------------------------------------------------------------------------------------|------------------------------|-------------------------|
| (20) <i>dec</i> ::= | <code>val <i>tyvar</i> seq <i>valbind</i></code>                                            | értékdeklaráció              | value declaration       |
| (21)                | <code>fun <i>tyvar</i> seq <i>valbind</i></code>                                            | függvénydeklaráció           | function declaration    |
| (22)                | <code>type <i>typpbind</i></code>                                                           | üres deklaráció              | type declaration        |
| (23)                |                                                                                             | üres deklaráció              | empty declaration       |
| (24)                | <code>dec<sub>1</sub> &lt;;&gt; dec<sub>2</sub></code>                                      | deklaráció-sorozat           | sequential declaration  |
| (25)                | <code>infix &lt;<i>d</i>&gt; <i>id</i><sub>1</sub> ... <i>id</i><sub><i>n</i></sub></code>  | infix-direktíva, $n \geq 1$  | infix (left) directive  |
| (26)                | <code>infixr &lt;<i>d</i>&gt; <i>id</i><sub>1</sub> ... <i>id</i><sub><i>n</i></sub></code> | infixr-direktíva, $n \geq 1$ | infix (right) directive |
| (27)                | <code>nonfix <i>id</i><sub>1</sub> ... <i>id</i><sub><i>n</i></sub></code>                  | nonfix-direktíva, $n \geq 1$ | nonfix directive        |

## • Példák:

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

Deklaratív programozás, BMF, 2001 tavaszi félév

9. előadás (funkcionális programozás)

163

## SML-szintaxis: típuskifejezések

Egyszerűsített SML-szintaxis

163

• Típus (*ty*: type)

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

• Típuskifejezés-sor (*tyrow*: type-expression row)(37) *tyrow* ::= `lab : ty <, tyrow>`

## • Példák:

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

Deklaratív programozás, BMF, 2001 tavaszi félév

10. előadás (funkcionális programozás)

164

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

• Értékkötés (*valbind*: value binding)

|                         |                                                                 |                |                   |
|-------------------------|-----------------------------------------------------------------|----------------|-------------------|
| (28) <i>valbind</i> ::= | <code><i>pat</i> = <i>exp</i> &lt;and <i>valbind</i>&gt;</code> | értékkötés     | value binding     |
| (29)                    | <code>rec <i>valbind</i></code>                                 | rekurzív kötés | recursive binding |

• Függvényérték-kötés (*valbind*: function value binding)

|                         |                                                                                                                                                       |                 |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------|
| (30) <i>valbind</i> ::= | <code>&lt;op&gt; var <i>atpat</i><sub>1</sub> ... <i>atpat</i><sub><i>n</i></sub> &lt;; <i>ty</i>&gt; = <i>exp</i><sub>1</sub></code>                 | $m_1, n \geq 1$ |
|                         | <code>  &lt;op&gt; var <i>atpat</i><sub>2</sub> ... <i>atpat</i><sub><i>n</i></sub> &lt;; <i>ty</i>&gt; = <i>exp</i><sub>2</sub></code>               |                 |
|                         | <code>  ...</code>                                                                                                                                    |                 |
|                         | <code>  &lt;op&gt; var <i>atpat</i><sub><i>m</i></sub> ... <i>atpat</i><sub><i>m</i></sub> &lt;; <i>ty</i>&gt; = <i>exp</i><sub><i>m</i></sub></code> |                 |
|                         | <code>&lt;and <i>valbind</i>&gt;</code>                                                                                                               |                 |

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

## • Példák:

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

Deklaratív programozás, BMF, 2001 tavaszi félév

9. előadás (funkcionális programozás)

164

## SML-szintaxis: minták

Egyszerűsített SML-szintaxis

164

• Atomni minta (*atpat*: atomic pattern)

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

## • Példák:

```
fun le GREATER = false | le EQUAL = true | le LESS = true; vö. (40)
fun le GREATER = false | le _ = true; vö. (38), (40)
fun neg Bool.false = true | neg (true) = Bool.false; vö. (40), (45)
fun prod [a, b] = a*b | prod [a, b, c] = a*b*c
| prod [a] = a | prod () = 1; vö. (43), (44)
```

Deklaratív programozás, BMF, 2001 tavaszi félév

10. előadás (funkcionális programozás)

164

## SML-szintaxis: minták (folyt.)

- Mintasor (*patrow*: pattern row)

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

- Példák:

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

## SML-szintaxis: szintaktikai korlátozások

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

## SML-szintaxis: minták (folyt.)

- Minta (*pat*: pattern)

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

- Példa:

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

## RACIONÁLIS SZÁMOK

## Példa: racionális számok

- A racionális számokat *rekordként* ábrázoljuk; az új (gyenge) típus neve `rat`.  
`type rat = {num : int, den : int};`
- Nevet adunk néhány állandónak.  
`val ratZero = {num = 0, den = 1}; val ratOne = {num = 1, den = 1};`  
`val ratHalf = {num = 1, den = 2}; val ratThird = {num = 1, den = 3};`
- A `rat` típusú számokat *normalizált* alakban tároljuk, különben pl.  $\frac{1}{2}$  és  $\frac{2}{4}$  nem lenne egyenlő. A normalizáláshoz szükségünk van a számláló és a nevező legnagyobb közös osztójára (`gcd`). A közös osztó egyik fontos tulajdonsága, hogy  $d|n$  és  $d|m \Rightarrow d|n \bmod m$ .  
`(* gcd : int -> int -> int`  
`gcd n m = n és m legnagyobb közös osztója`  
`*)`  
`fun gcd n 0 = abs n`  
`| gcd n m = gcd (abs m) (abs (n mod m));`

Deklaratív Programozás, BME, 2001 tavaszi félév

10. előadás (funkcionális programozás)

## Példa: racionális számok (folyt.)

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

Deklaratív Programozás, BME, 2001 tavaszi félév

10. előadás (funkcionális programozás)

## Kitérő: pár és típusa

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

- A + kétoperandusú művelet, argumentuma egy *pár*, pl. 3 + 4.
- + : `int * int -> int` vagy + : `real * real -> real`, ahol \* egy újabb típusművelet, a *kereszt-szorzat* (*Descartes-szorzat*) jele.
- A + műveleti jel (függvényjel) *többszörös terhelésű*.
- + prefix helyzetben is használható, ha elírjuk az op kulcsszót, pl. `op+(3, 4)`. Ilyenkor az operandusait *párként, zárójelbe zárva* kell megadni.
- A beépített infix típusoperátorok precedenciája és kötése
- Két beépített infix típusoperátor van az SML-ben: `->` (leképzés) és \* (kereszt-szorzat). A \* precedenciája a nagyobb. A `->` jobbra két, a \* nem két sem balra, sem jobbra.

- Példák: `?a * ?b * ?c = (?a * ?b) * ?c`  
`?a -> ?b -> ?c = ?a -> (?b - ?c)`  
`?a * ?b -> ?c = (?a * ?b) -> ?c`

Deklaratív Programozás, BME, 2001 tavaszi félév

10. előadás (funkcionális programozás)

## PÁR ÉS TÍPUSA

**Példa: racionális számok – a négy alapművelet**

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

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

**Példa: racionális számok – relációs műveletek**

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

```
(* <<, >>, <=, >= : rat * rat -> bool
r1 << r2 = igaz, ha r1 kisebb r2-nél
r1 >> r2 = igaz, ha r1 nagyobb r2-nél
r1 <= r2 = igaz, ha r1 nem nagyobb r2-nél
r1 >= r2 = igaz, ha r2 nem nagyobb r1-nél
*)
infix 4 << >> <= >=;
```

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

**POLIMORFIZMUS**

## Polimorfizmus

- Nézzük az identitásfüggvényt:  $\text{fun id } x = x.$
- Mi az  $x$  típusa? Bármilyen típusú lehet: típusát *típusváltozó* jelöli.  
`> val 'a id = fn : 'a -> 'a`
- id *polimorf* függvényt jelöl,  $x$  és id *polítípusú* nevek.

- A *percjel*lel kezdődő típusnév (pl. 'a, olvasd *alfa*): *típusváltozó*.

Polimorfizmus többféle változatban fordul elő a programezésben.

- Egy *polimorf* név egyetlen olyan algoritmust azonosít, amely tetszőleges típusú argumentumra alkalmazható; ez a *paraméteres polimorfizmus*.
- Egy *többszörösen terhel*t név több különböző algoritmust azonosít: ahány típusú argumentumra alkalmazható, annyiféle; ez az *ad-hoc* vagy *többszörös terheléses* polimorfizmus.
- A polimorfizmus harmadik változata az *öröklődéses polimorfizmus* (vö. objektum-orientált programezés).

## Kitérő: két függvény kompozíciója

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

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

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

- Az  $o$  típusa  $? * ? -> ?$  szerkezetű. Mit írjunk a  $?$ -ek helyébe? Vezessük le!
- A függvénydefiniáció jobb oldalán álló kifejezés elenzésével kezdjük.

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

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

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

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

## KÉT FÜGGVÉNY KOMPOZÍCIÓJA

## RACIONÁLIS SZÁMOK

## Példa: racionális számok (folyt.)

- A racionális számokon értelmezett  $<=>$  és  $>>=$  másképpen:  
`val op<= = not o op>>;`    `val op>>= = not o op<=;`

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

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

```
*)
```

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

- Példák rat típusú értékek használatára

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

Deklaratív Programozás, BMF, 2001 tavaszi félév

10. előadás (funkcionális programozás)

## Példa: racionális számok (folyt.)

- Példák rat típusú értékek használatára (folyt.)

```
toRat 2 3 << toRat 5 4; toRat 2 3 >> toRat 5 3;
toRat 1 4 <<< toRat 3 10; toRat 3 10 >>> toRat 1 4;
infix 8 /-/;
fun n /- / d = toRat n d;
```

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

- Példák gcd részleges alkalmazására

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

Deklaratív Programozás, BMF, 2001 tavaszi félév

10. előadás (funkcionális programozás)

## Paraméteres polimorfizmus

- Az identitásfüggvény és típusa: `fun id x = x, id : 'a -> 'a.`

Az `mosml` válasza: `val 'a id = fn : 'a -> 'a. Az id poliktípusú név.`

- Az = és a  $<>$  műveletet *készen kapjuk* a legtöbb típusra (vö. `rat`).

A típusuk: `=, <> : 'a * 'a -> bool. A 'a egyenlőségi típust jelöl, az ilyen típusú értékeken az egyenlőségvizsgálat elvégezhető.`

- Az egyenlőségvizsgálat *kortároltlan* polimorf: nem minden értékre végezhető el. Pl. egy *f* és egy *g* függvény akkor és csak akkor egyenlő, ha  $\forall x. f x = g x$ . Ezt *általánosságban* lehetetlen eldönteni.

- Mi a `<`, `>`, `<=`, `>=` típusa?

Pl. az `op<=`-re az `mosml` válasza: `val it = fn : int * int -> bool.`

E négy művelet *ad-hoc* módon polimorf, a nevek *többszörösen terhelhetők*, alapértelmezés szerint int típusú értékekre alkalmazhatók.

- Az = részlegesen alkalmazható változata legyen: `fun eq x y = x = y.`  
Típusa: `eq : 'a -> 'a -> bool.`

Deklaratív Programozás, BMF, 2001 tavaszi félév

11. előadás (funkcionális programozás)

## POLIMORFIZMUS

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

### A kifejezés Az mosml válasza

- ```

eq 3 3;
eq "id" "idn";
eq id id;
  | Toplevel input:
  | eq id id;
  | ^^
  | Type clash: expression of type
  | 'e -> 'e
  | cannot have equality type 'f
eq 3;
eq "id";
val eqStr_id = eq "id";
  > val it = fn : string -> bool
  > val eqStr_id = fn : string -> bool

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

  ● Az eq 'id' függvényértéket ad eredményül, ezért az eqStr_id függvényt
jelöl. Olyan függvényt, amely az 'id' füzetre alkalmazva true, minden más
esetben false értéket ad eredményül.

```

Deklaratív programozás, BMF, 2001 tavaszi félév

11. előadás (funkcionális programozás)

Érték-polimorfizmus

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

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

A kifejezés Az mosml válasza

- ```

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

 ● Az SML ún. érték-polimorfizmust használ.

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

 ● Az mosml a nem általánosítható típusváltozókat meghagyja szabad
típusváltozónak (pl. id id).

```

Deklaratív programozás, BMF, 2001 tavaszi félév

11. előadás (funkcionális programozás)

## Nem-expanzív kifejezés (egyszerűsítve)

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

|                        |                                             |                                                |
|------------------------|---------------------------------------------|------------------------------------------------|
| <i>scop</i>            | különlleges állandó<br>(esetleg minősített) | special constant<br>(possibly qualified) value |
| <i>longvid</i>         | értéknév                                    | identifier                                     |
| { < <i>nemrow</i> > }  | nem-expanzív<br>elemekből álló rekord       | record of non-expansive<br>expressions         |
| ( <i>nem</i> )         | nem-expanzív kifejezés                      | parenthesized<br>non-expansive expression      |
| <i>nem</i> : <i>ty</i> | nem-expanzív kifejezés<br>típusmegkötéssel  | typed non-expansive<br>expression              |
| fn <i>match</i>        | függvénykifejezés                           | function expression                            |
- Nem-expanzív kifejezőssor (*nemrow*: non-expansive expression row)

*nemrow* ::= *lab* = *nem* < , *nemrow* >

Deklaratív programozás, BMF, 2001 tavaszi félév

11. előadás (funkcionális programozás)

Deklaratív programozás, BMF, 2001 tavaszi félév

11. előadás (funkcionális programozás)

## Példák nem-expanzív és expanzív kifejezésekre

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

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

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

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

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

Deklaratív Programozás, BMF, 2001 tavaszi félév

1.1. előadás (funkcionális programozás)

Polimorfizmus 191

## $\eta$ -expanzió

- A típusváltozó általánosítása mindig kikényszeríthető a deklaráció jobb oldalának  *$\eta$ -expanziójával*.

Az  $\eta$ -expanzió az e kifejezést a nem-expanzív fn y => e y kifejezéssel helyettesíti.

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

A fenti deklarációban a külső zárójelpár el is hagyható:

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

- Az x1 politípusú név.

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

Deklaratív Programozás, BMF, 2001 tavaszi félév

1.1. előadás (funkcionális programozás)

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

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

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

```
x;
> val it = fn : string list -> int
```

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

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

Deklaratív Programozás, BMF, 2001 tavaszi félév

1.1. előadás (funkcionális programozás)

## LISTÁK



## Listák: definíciók, konstruktorok

- Definíciók
  - A lista azonos típusú elemek véges (de nem korlátos!) sorozata.
  - A lista olyan *rekurzív* lineáris adatszerkezet, amely azonos típusú elemekből áll, és
    - vagy üres,
    - vagy egy elemből és az elemet követő listából áll.
- Konstruktorok
  - Az üres lista jele a `nil` *konstruktorállandó*. `nil` típusa `'a list`.
  - A `::` *konstruktoroperátor* új listát hoz létre egy elemből és egy (esetleg üres) listából (infix, 5-ös precedenciájú, jobbra köt, típusa `'a * 'a list -> 'a list`).
  - A `nil` helyett általában a `[]` jelet használjuk (szintaktikai édesítőszert).
  - A `::`-et négyesponthnak vagy *cons*-nak olvassuk (vö. *constructor*, ami a függvény hagyományos neve a  $\lambda$ -kalkulusban és egyes funkcionális nyelvekben).

Deklaratív programozás, BMF, 2001 tavaszi félév

11. előadás (funkcionális programozás)

## Listák: fej (hd), farkok (tl)

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

```
(* hd : 'a list -> 'a *)
fun hd (x :: _) = x;
```
- A nem-üres lista első utáni elemeiből áll a lista *farka*.
 

```
(* tl : 'a list -> 'a list *)
fun tl (_ :: xs) = xs;
```
- `hd` és `tl` *parciális* függvények. Ha könyvtárbeli megfelelőiket (`List.hd`, `List.tl`) üres listára alkalmazzuk, `Empty` néven *kiértékel* jeleznek.
 

Pontos: a parciális függvények nem tévesztendőek össze a parciálisán (azaz részlegesen) alkalmazható függvényekkel!

Deklaratív programozás, BMF, 2001 tavaszi félév

11. előadás (funkcionális programozás)

## Listák: jelölések, minták

- Példák
  - Listák létrehozása konstruktorokkal
 

```
[] nil #" " :: nil
3 :: 5 :: 9 :: nil = 3 :: (5 :: (9 :: nil))
```
  - Szintaktikus édesítőszert lista jelölésére
 

```
[3, 5, 9] = 3 :: 5 :: 9 :: nil
```
  - Vigyázat! A Prolog lista\_jelölése hasonló, de vannak lényeges különbségek:
 

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

A `[]` és a `nil` állandók, a `::` operátor, valamint a `[x1, x2, ..., xn]` lista\_jelölés mintában is alkalmazhatók.

Deklaratív programozás, BMF, 2001 tavaszi félév

11. előadás (funkcionális programozás)

## Listák: hossz (length), elemek összege (isum), szorzata (rprod)

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

```
(* length : 'a list -> int *)
fun length (_ :: xs) = 1 + length xs
 | length [] = 0;
```
- Egy egész számokból álló lista elemeinek összegét adja eredményül `isum`.
 

```
(* isum : int list -> int *)
fun isum (x :: xs) = x + isum xs
 | isum [] = 0;
```
- Egy valós számokból álló lista elemeinek szorzatát adja eredményül `rprod`.
 

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

Deklaratív programozás, BMF, 2001 tavaszi félév

11. előadás (funkcionális programozás)

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

| <code>hd</code> , <code>tl</code>                              | Az <code>msoml</code> válasza                              |
|----------------------------------------------------------------|------------------------------------------------------------|
| A kifejezés                                                    |                                                            |
| <code>List.hd [1, 2, 3];</code>                                | <code>&gt; val it = 1 : int</code>                         |
| <code>List.hd [];</code>                                       | <code>  Uncaught exception:</code><br><code>  Empty</code> |
| <code>List.tl [1, 2, 3];</code>                                | <code>&gt; val it = [2, 3] : int list</code>               |
| <code>List.tl [];</code>                                       | <code>  Uncaught exception:</code><br><code>  Empty</code> |
| ● <code>length</code> , <code>isum</code> , <code>rprod</code> | Az <code>msoml</code> válasza                              |
| A kifejezés                                                    |                                                            |
| <code>length [1, 2, 3, 4];</code>                              | <code>&gt; val it = 4 : int</code>                         |
| <code>length [];</code>                                        | <code>&gt; val it = 0 : int</code>                         |
| <code>isum [1, 2, 3, 4];</code>                                | <code>&gt; val it = 10 : int</code>                        |
| <code>isum [];</code>                                          | <code>&gt; val it = 0 : int</code>                         |
| <code>rprod [1.0, 2.0, 3.0, 4.0];</code>                       | <code>&gt; val it = 24.0 : real</code>                     |
| <code>rprod [];</code>                                         | <code>&gt; val it = 1.0 : real</code>                      |

Deklaratív Programozás, BMF, 2001 tavaszi félév

11. előadás (funkcionális programozás)

**Lista: adott transzformáció alkalmazása minden elemre (map)**

- Példa: vonjunk négyzetgyököket egy valós számköböl álló lista minden eleméből!
 

```
map Math.sqrt [1.0, 4.0, 9.0, 16.0] = [1.0, 2.0, 3.0, 4.0]
```
- Általában: `map f [x1, x2, ..., xn] = [f x1, f x2, ..., f xn]`
- A függvény típusa: `map : ('a -> 'b) -> 'a list -> 'b list`
- Egy-egy klézti írunk a triviális és a nem-triviális eset lefedésére
  - `map f [] = []`
  - `map f (x :: xs) = f x :: map f xs`
- `fun map f (x :: xs) = f x :: map f xs | map f [] = [];`
- map típusa, ha egyargumentumú függvénynek tekintjük (vi. -> jobbra kőt):
 

```
map : ('a -> 'b) -> ('a list -> 'b list).
```

Azaz ha `map`-et egy `'a -> 'b` típusú függvényre alkalmazunk, akkor olyan függvényt ad eredményül, amelyet egy `'a list` típusú listára alkalmazva egy `'b list` típusú listát kapunk.

Deklaratív Programozás, BMF, 2001 tavaszi félév

11. előadás (funkcionális programozás)

Programhelyesség 200

**A program helyességének igazolása a map példáján**

- A rekurzív programról be kell látnunk, hogy
  - funkcionálisan helyes (azt kapjuk eredményül, amit várunk),
  - a kiértékelése biztosan befejeződik (nem esik „végtelen ciklusba”).
- Bizonyítása *hossz szerinti strukturális indukcióval* (amely visszavezethető a teljes indukcióra) lehetséges.
  - `fun map f (x :: xs) = f x :: map f xs | map f [] = [];`
  - Fellesszük, hogy a `map` jó eredményt ad az eggyel rövidebb listára (azaz a lista farkára). Alkalmazzuk az `f`-et a lista első elemére (a fejére). A fej transzformálásával kapott eredményt a fark transzformálásával kapott lista elé fűzve valóban a várt eredményt kapjuk.
  - A kiértékelés véges számú lépésben befejeződik, mert a lista véges, a `map` függvényt a *rekurzív ágban* minden lépésben egyre rövidebb listára alkalmazzuk, és gondoskodtunk a rekurzív leállításáról (a *triviális eset* kezeléséről, vi. van nem rekurzív ág).

## PROGRAMHELYESSÉG

Deklaratív Programozás, BMF, 2001 tavaszi félév

11. előadás (funkcionális programozás)

## Lista: adott predikátumot kielégítő elemek kiválogatása (filter)

- Kitérő: explode, implode

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

- Példa: gyűjtsük ki a kisbetűket egy karakterlistából!

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

- Általában: ha  $p \ x_1 = \text{true}$ ,  $p \ x_2 = \text{false}$ ,  $p \ x_3 = \text{true}$ , ...,  $p \ x_n = \text{true}$ , akkor filter  $p \ [x_1, x_2, x_3, \dots, x_n] = [x_1, x_3, \dots, x_n]$ .
- A függvény típusa: filter : ('a -> bool) -> 'a list -> 'a list
- Egy-egy klózt írunk a triviális és a nem-triviális eset lefedésére
  - filter p [] = []
  - filter p (x :: xs) = if p x then x :: filter p xs else filter p xs

Deklaratív Programozás, BMF, 2001 tavaszi félév

11. előadás (funkcionális programozás)

Listák 203

## Lista: filter (folyt.)

- Ezzel filter definíciója

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

- filter típusa, ha egyargumentumú függvénynek tekintjük (-> jobbra költ!):  
 filter : ('a -> bool) -> ('a list -> 'a list).  
 Azaz ha filter-t egy 'a -> bool típusú függvényre (predikátumra) alkalmazzuk, akkor olyan függvényt ad eredményül, amelyet egy 'a list típusú listára alkalmazva egy 'a list típusú listát kapunk.

## Lista redukciója kétoperandusú művelettel (foldr, foldl)

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

```
foldr op* 1.0 [] = 1.0; foldl op+ 0 [] = 0;
foldr op* 1.0 [4,0] = 4.0; foldl op+ 0 [4] = 4;
foldr op* 1.0 [1,0,2,0,3,0,4,0] = 24.0; foldl op+ 0 [1,2,3,4] = 10;
```

- Jelöljön  $\oplus$  tetszőleges kétoperandusú infix operátort. Akkor  
 foldr  $\oplus$  e [x<sub>1</sub>, x<sub>2</sub>, ..., x<sub>n</sub>] = (x<sub>1</sub>  $\oplus$  (x<sub>2</sub>  $\oplus$  ...  $\oplus$  (x<sub>n</sub>  $\oplus$  e) ...))  
 foldl  $\oplus$  e [] = e  
 foldl  $\oplus$  e [x<sub>1</sub>, x<sub>2</sub>, ..., x<sub>n</sub>] = (x<sub>n</sub>  $\oplus$  ...  $\oplus$  (x<sub>2</sub>  $\oplus$  (x<sub>1</sub>  $\oplus$  e)) ...)  
 foldl  $\oplus$  e [] = e
- Asszociatív műveleteknél foldr és foldl eredménye azonos.

Deklaratív Programozás, BMF, 2001 tavaszi félév

11. előadás (funkcionális programozás)

Deklaratív Programozás, BMF, 2001 tavaszi félév

12. előadás (funkcionális programozás)

## Példák foldr és foldl alkalmazására

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

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

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

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

Deklaratív Programozás, BMF, 2001 tavaszi félév

12. előadás (funkcionális programozás)

## Lista: foldr és foldl definíciója

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

```
[* foldr f e xs = az xs elemekre jobbról balra haladva alkalmazott,
 kétooperandusú, e egységelenű f művelet eredménye
 foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun foldr f e (x::xs) = f(x, foldr f e xs)
 | foldr f e [] = e;
```
- foldl  $op \oplus e [x_1, x_2, \dots, x_n] = (x_n \oplus \dots \oplus (x_2 \oplus (x_1 \oplus e)) \dots)$   
 foldl  $op \oplus e [] = e$ 

```
[* foldl f e xs = az xs elemekre balról jobbra haladva alkalmazott,
 kétooperandusú, e egységelenű f művelet eredménye
 foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun foldl f e (x::xs) = foldl f (f(x, e)) xs
 | foldl f e [] = e;
```

Deklaratív Programozás, BMF, 2001 tavaszi félév

12. előadás (funkcionális programozás)

## Példák foldr és foldl alkalmazására (folyt.)

- Egy lista elemeit egy másik lista elé fűzi foldr és foldl, ha kétooperandusú műveletként a *cons* konstruktorfüggvényt – azaz az  $op:::ot$  – alkalmazzuk.
 

```
foldr op::: ys [x1, x2, x3] = (x1 :: (x2 :: (x3 :: ys)))
foldl op::: ys [x1, x2, x3] = (x3 :: (x2 :: (x1 :: ys)))
```
- A :: nem asszociatív, ezért foldl és foldr eredménye különböző!
 

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

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

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

Deklaratív Programozás, BMF, 2001 tavaszi félév

12. előadás (funkcionális programozás)

## Lista redukciója bal oldali egységelenű függvényvel (foldl)

- A kivonás művelete balra köti:  $x_1 - x_2 - x_3 - x_4 = ((x_1 - x_2) - x_3) - x_4$ .
- Nem feleltethető meg sem foldr-nek, sem foldl-nek.
 

```
foldr op⊖ e [x1, x2, ..., xn] = (x1 ⊖ (x2 ⊖ ... ⊖ (xn ⊖ e) ...))
foldl op⊖ e [x1, x2, ..., xn] = (xn ⊖ ... ⊖ (x2 ⊖ (x1 ⊖ e)) ...)
```
- Nevezzük foldl-nek a listában *balról jobbra* haladó, alábbi specifikációjú függvényt. Vegyük észre, hogy  $\oplus$  bal oldali egységelenet vár.
 

```
foldl op⊖ e [x1, x2, ..., xn] = (... ((e ⊖ x1) ⊖ x2) ⊖ ... ⊖ xn)
```
- foldl olyan kétargumentumú függvényt vár, amelynek az „egységelen” (valójában: a részeredmény) az *első* argumentuma:  $f : 'a * 'b -> 'a$ .
 

```
(* foldl : ('a * 'b -> 'a) -> 'a -> 'b list -> 'a
 foldl f e xs = az xs elemekre balról jobbra haladva alkalmazott,
 kétooperandusú, e egységelenű f művelet eredménye *)
fun foldl f e (x::xs) = foldl f (f(e, x)) xs
 | foldl f e [] = e;
```

Deklaratív Programozás, BMF, 2001 tavaszi félév

12. előadás (funkcionális programozás)

## Példák listaelemek különbségének és hányadosának képzésére

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

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

- Ha többször használjuk a műveletet, érdemes nekik nevet adni. A *kisebbitendő*, ill. az *osztandó* speciális kezelését elrejtjük.

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

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

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

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

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

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

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

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

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

## KIFEJJEZÉSEK KIÉRTÉKELÉSE

### Mohó kiértékelés: faktoriális kiszámítása naïv rekurzívval

- A faktoriális matematikai definíciója és megvalósítása SML-ben

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

(* fac : int -> int (--) fontos a klózok sorrendje! --)
 fac n = n!
 PRE n >= 0 *)
fun fac 0 = 1 | fac n = n * fac (n-1);
```

- **fac** mohó kiértékelése  $n = 4$  esetén (egyres triviális lépéseket elhagynk).  
`fac 4`  $\rightarrow$  `4 * fac (4-1)`  $\rightarrow$  `4 * fac 3`  $\rightarrow$  `4 * (3 * fac (3-1))`  $\rightarrow$   
 $\rightarrow$  `4 * (3 * fac (2))`  $\rightarrow$   $\dots$   $\rightarrow$  `4 * (3 * (2 * (1 * 1)))`  $\rightarrow$   $\dots$   $\rightarrow$  24
- A rekurzív kiértékelés követi a matematikai definíciót.
- Rontja a hatékonyságot, hogy a rekurzív végrehajtás során minden részeredményt a veremben tárolni kell.
- Ha a szorzás asszociativitását kihasználjuk, nem kell tárolni az összes tényezőt, csak az aktuális részeredményt.

## Faktorális kiszámítása jobbrekurzíval

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

```
(* faci : int -> int -> int (--- fontos a klézok sorrendje! ---)
 faci n p = p * n! (--- p az akkumulátor ---)
*)
fun faci 0 p = p
 | faci n p = faci (n-1) (n*p);
```
- faci*-t felhasználjuk az egyparaméteres *fac* függvény definiálására. Az akkumulátornak alkalmas *kezdőértéket* adunk.
 

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

Deklaratív programozás, BMEF, 2001 tavaszi félév

12. előadás (funkcionális programozás)

## Lokális kifejezés

- Lokális kifejezést* használunk, ha ismétlődő részkifejezéseket *csak egyszer* akarunk kiszámítani, vagy akkor, ha bizonyos értékeket a program többi része elől *el akarunk rejtetni*.
  - Szintaxisa: `let d in e end, ahol`
  - d* nemüres deklaráciosorozat,
  - e* nemüres kifejezés.
- Példa:
- ```
fun fac n =
  let
    fun faci 0 p = p
      | faci n p = faci (n-1) (n*p)
    in
      faci n 1
    end
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

12. előadás (funkcionális programozás)

Faktorális kiszámítása jobbrekurzíval (folyt.)

- fac* nem rekurzív, ezért csak *faci* kiterjedését vizsgáljuk (egyes triviális lépéseket összevonnunk).


```
A függvény: fun faci 0 p = p | faci n p = faci (n-1) (n*p)
   faci 4 1 → faci (4-1) (4*1) → faci 3 4 → faci (3-1) (3*4) →
   → faci 2 12 → ... → faci 0 24 → 24
```
- Kiterjeelés közben a *p* *akkumulátor* gyűjti a részeredményt, ezért *faci* tárgyánál állandó.
- A kiterjeelés *iteratív*.
- A jó fordítóprogram felismeri a jobbrekurzívt, és hatékony tárgykódot állít elő: az argumentumokat frissíthető lokális változókban tárolja, a rekurzíót iterációval helyettesíti.
- A jobbrekurzívt *terminális rekurzió*nak is nevezik (angolul: *tail* vagy *terminal recursion*).
- földl *jobbrekurzív*, e argumentuma akkumulátorként viselkedik.

Deklaratív programozás, BMEF, 2001 tavaszi félév

12. előadás (funkcionális programozás)

Lokális deklaráció

- Lokális deklarációt* használunk olyan értékek bevezetésére, amelyeket a program többi része elől *el akarunk rejtetni*.
 - Szintaxisa: `local d1 in d2 end, ahol`
 - d1* és *d2* nemüres deklaráciosorozatok.
- Példa:
- ```
local
 fun faci 0 p = p
 | faci n p = faci (n-1) (n*p)
in
 fun fac n = faci n 1
end
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

12. előadás (funkcionális programozás)

## Logikai műveletek

- **Típusnév:** `bool`, `adatkonstruktorok`: `false`, `true`, `beépített függvény`: `not`.
- **Lusta kiértékelésű beépített operátorok**
  - Három argumentumú: `if b then e1 else e2`.  
Nem értékeli ki az `e2`-t, ha a `b igaz`, ill. az `e1-et`, ha a `b hamis`.
  - Két argumentumúak:
    - `e1 andalso e2` : nem értékeli ki az `e2`-t, ha az `e1 hamis`.
    - `e1 orelse e2` : nem értékeli ki az `e2`-t, ha az `e1 igaz`.
- Mind a három csupán szintaktikai édesítfőszéri!
  - `if b then e1 else e2`  $\equiv$  `(fn true => e1 | false => e2) b`
  - `e1 andalso e2`  $\equiv$  `(fn true => e2 | false => false) e1`
  - `e1 orelse e2`  $\equiv$  `(fn true => true | false => e2) e1`
  - `fun iffThenElse b = (fn true => e1 | false => e2) b; iffThenElse true;`
- **Tipikus hiba: `if exp then true else false` !!!**

## Logikai műveletek (folyt.)

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

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

fun op|| (a, b) = a orelse b;
infix 1 ||;
```

## LISTÁK

## Listák összefűzése és megfordítása

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

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

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

## Lista legnagyobb elemének megkeresése

- Egy egészlista legnagyobb elemének kiválasztásához szükségünk van az Int.max függvényre.

- Üres listának nincs legnagyobb eleme,
- egyelemű listában az egyetlen elem a legnagyobb,
- legalább két elemű lista legnagyobb elemét úgy kapjuk, hogy az első elem és a maradéklista elemeinek legnagyobbika közötti kiválasztjuk a legnagyobbat.

```
(* maxl ns = az ns egészlista legnagyobb eleme
 maxl : int list -> int *)
fun maxl [n] = n
 | maxl (n::ns) = Int.max(n, maxl ns)
 | maxl [] = raise Empty;
```

- max egy változata egészszekre

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

## Listák összefűzése és megfordítása

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

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

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

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

## Lista legnagyobb elemének megkeresése (folyt.)

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

```
(* maxl max ns = az ns lista legnagyobb eleme
 maxl : ('a * 'a -> 'a) -> 'a list -> 'a *)
fun maxl max [n] = n
 | maxl max (n::ns) = max(n, maxl max ns)
 | maxl max [] = raise Empty;
```

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

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



## Lista (folyt.)

- **Változatos max-ra**

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

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

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

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

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

Deklaratív programozás, BMF, 2001 tavaszi félév

13. előadás (funkcionális programozás)

## Halmazműveletek

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

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

• newMem egy új elemet rak be egy listába, ha még nincs benne.
 (* newMem(x, xs) = [x] és xs listaként ábrázolt uniója
 newMem : 'a * 'a list -> 'a list *)
fun newMem (x, xs) = if x isMem xs then xs else x::xs;

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

Deklaratív programozás, BMF, 2001 tavaszi félév

13. előadás (funkcionális programozás)

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

- **Legyen**  $xs = [x_0, x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_{n-1}]$ , akkor
 

```
take(xs, i) = [x_0, x_1, \dots, x_{i-1}] és drop(xs, i) = [x_i, x_{i+1}, \dots, x_{n-1}].

(* take (xs, i) = xs, ha i < 0;
 az xs első i db eleméből álló lista, ha i >= 0
 take : 'a list * int -> 'a list *)
fun take (_, 0) = []
 | take ([_, _]) = []
 | take (x::xs, i) = x :: take(xs, i-1);

(* drop(xs, i) = xs, ha i < 0;
 az xs első i db elemének elhagyásával előállított lista, ha i >= 0
 drop : 'a list * int -> 'a list *)
fun drop ([_, _]) = []
 | drop (x::xs, i) = if i > 0 then drop (xs, i-1) else x::xs;

• Könyvtári változatok, List.take és List.drop i < 0 vagy i > length xs esetén
 Subscript kivételt jelez.
```

Deklaratív programozás, BMF, 2001 tavaszi félév

13. előadás (funkcionális programozás)

## Halmazműveletek (folyt.)

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

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

• Szerezhetőbb a halmazokat a megszokott halmazműveletekkel kezelni. Őt
 halmazműveletet definiálunk:

 • unió (union, $S \cup T$),
 • metszet (inter, $S \cap T$),
 • részhalmaz-e (isSubset, $T \subseteq S$),
 • egyenlők-e (isSetEq, $S = T$),
 • hatványhalmaz (powerSet, P^S).
```

Deklaratív programozás, BMF, 2001 tavaszi félév

13. előadás (funkcionális programozás)

## Halmazműveletek (folyt.)

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

- Két halmaz uniója

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

- Két halmaz metszele

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

13. előadás (funkcionális programozás)

## Halmazműveletek (folyt.)

- Halmaz hatványhalmaza

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

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

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

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

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

13. előadás (funkcionális programozás)

## Halmazműveletek (folyt.)

- Részhalmaza-e egy halmaz egy másiknak?

```
(* isSubset (xs, ys) = az xs elemeiből álló halmaz részhalmaza-e
 az ys elemeiből álló halmaznak
 isSubset : 'a list * 'a list -> bool *)
fun isSubset (x::xs, ys) = (x isMem ys) andalso isSubset(xs, ys)
 | isSubset ([], _) = true;
infix isSubset;
```

- Két halmaz egyenlősége

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

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

13. előadás (funkcionális programozás)

## Halmazműveletek (folyt.)

- Halmaz hatványhalmaza (folyt.)

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

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

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

13. előadás (funkcionális programozás)

## Sztatikus és dinamikus kötés, mohó és lusta kiértékelés

- **Sztatikus kötés:** a formális paraméter összes előfordulását *fordítási időben* helyettesítjük az argumentummal (aktuális paraméterrel) a függvény (eljárás) törzsében.
  - **Dinamikus kötés:** a formális paraméter összes előfordulását *futási időben* helyettesítjük az argumentummal (aktuális paraméterrel) a függvény (eljárás) törzsében.
- Kérdés, hogy az aktuális paraméterként átadott kifejezést az értelmező mikor értékeli ki: a behelyettesítés *előtt* vagy *után*.

- **Mohó kiértékelés:** a behelyettesítés *előtt* kiértékeljük az összes argumentumot (más megnevezések: *érték szerinti* paraméterátadás, *eager evaluation*, *call-by-value*).
- **Lusta kiértékelés:** a behelyettesítés *után* csak azt az argumentumot értékeli ki, amelyikre szükség van, és csak akkor, amikor *szükség van* rá (más megnevezések: *szükség szerinti* paraméterátadás, *lazy evaluation*, *call-by-need*.)

Deklaratív programozás, BMEF, 2001 tavaszi félév

1.4. előadás (funkcionális programozás)

Kifejezések kiértékelése 265

## A mohó és a lusta kiértékelés összevetése

- Más paraméterátadási eljárások
- *névszerinti* paraméterátadás (*call-by-name*, Algol).
- *hivatkozás szerinti* paraméterátadás (*call-by-reference*, Pascal, C stb.)
- Nézzünk két egyszerű függvényt!

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

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

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

Deklaratív programozás, BMEF, 2001 tavaszi félév

1.4. előadás (funkcionális programozás)

Kifejezések kiértékelése 266

## Mohó kiértékelés

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

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

- Nézzük `sq(sq(2))` egyszerűsítését! (Az egyszerűsítés eredménye tovább már nem egyszerűsíthető, ún. *kanonikus* kifejezés.)  
`sq három alkalmazásából csak a harmadiknak kanonikus kifejezés az argumentuma.`

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

Az utolsó lépése kivételével `zero(sq(sq 2))` egyszerűsítési lépései ugyanezek, pedig az eredmény nyilvánvalóan 0!

Mohó kiértékelés mellett a számítógépet feleslegesen dolgoztatjuk!

Deklaratív programozás, BMEF, 2001 tavaszi félév

1.4. előadás (funkcionális programozás)

## Név szerinti paraméterátadás

- Egy függvény alkalmazása előtt sokszor nemcsak főlegesen, hanem káros is előre kiszámítani az argumentumokat, mert végtelen rekurzív vagy illegális művelet (indexhatár-túllépés, 0-val való osztás stb.) lehet az „eredménye”.
- Az Algol *névszerinti* paraméterátadása a formális paraméter összes előfordulását az argumentumként átadott *teljes* (nem kanonikus) *kifejezéssel* helyettesíti a függvény törzsében.  
Ezért `zero(sq(sq(2)))` *névszerinti* paraméterátadás esetén *azonnal*, az argumentum kiértékelése nélkül 0-t ad eredményül!
- `fun sq x = x * x;` `fun zero x = 0;`
- A *névszerinti* paraméterátadás sem mindig kedvező: pl. `sq(sq(2))` esetén `sq` mindvégig alkalmazása *megkét-szerzi* az argumentumok számát. Aligha ezt akarjuk!  
`sq(sq(2)) → sq(sq(2)) * sq(sq(2)) → (sq 2 * sq 2) * sq(sq 2) → ((2*2) * sq 2) * sq(sq 2) → ... → (4*(2*2)) * sq(sq 2) → ...`

Deklaratív programozás, BMEF, 2001 tavaszi félév

14. előadás (funkcionális programozás)

## Lusta kiértékelés

- Nézzük pl. `sq(sq(2))` lusta kiértékelését!  
`fun sq x = x * x;` `fun zero x = 0;`  
( $x = [L]$  jelentése: az  $x$  összes előfordulása osztozik az  $L$  értékken.)  
`sq(sq(sq 2)) → x * x [x = sq(sq 2)] → x * x [x = y * y] [y = sq 2] → x * x [x = y * y] [y = 2 * 2] → x * x [x = y * y] [y = 4] → x * x [x = 4 * 4] → x * x [x = 16] → 16 * 16 → 256`
- Gyakran nyerünk, de néha veszünk a lusta kiértékeléssel.  
Látnuk, hogy `fun fac1(0, p) = p | fac1(n, p) = fac1(n-1, n*p)` *mozdó kiértékelés* esetén hatékonyabb `fac`-nál, mert az `n*p` szorzást azonnal végrehajtja. *Lusta kiértékelés* esetén az `n`-et azonnal kiszámítaná (szükség van `n` értékére az `implicit n = 0` vizsgálathoz), a `p` kiértékelését azonban a szorzások akkumulálásával késleltethetné:  
`fac1(4, 1) → fac1(4-1, 4*1) → fac1(3-1, 3*(4*1)) → fac1(2-1, 2*(3*(4*1))) ... → 24`

Deklaratív programozás, BMEF, 2001 tavaszi félév

14. előadás (funkcionális programozás)

## Lusta kiértékelés

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

Deklaratív programozás, BMEF, 2001 tavaszi félév

14. előadás (funkcionális programozás)

## ÖSSZETETT ADATTÍPUSOK

## Ennes és típusa

- Két különböző típusú értékből rekordot vagy párt képezhetünk. Pl.  $\{x = 2, y = 1.0\} : \{x : \text{int}, y : \text{real}\}$  és  $(2, 1.0) : (\text{int} * \text{real})$ .
- A pár is csak szintaktikai édesítőszers. Pl.  $(2, 1.0) = \{1 = 2, 2 = 1.0\} = \{2 = 1.0, 1 = 2\}$ , de  $(2, 1.0)$  és  $\{1 = 1.0, 2 = 2\}$  különböző típusúak. Az 1 és a 2 *mezőnevek* (vö. szintaxis).
- Rekordot kettőnél több értékből is összeállíthatunk. Pl. 

```
{nev = "Bea", tel = 3192144, kor = 19} : {kor : int, nev : string, tel : int}.
Egy hasonló rekord egészszám-mezőnevekkel:
{1 = "Bea", 3 = 3192144, 2 = 19} : {1 : string, 2 : int, 3 : int.
Az utóbbi azonos az alábbi emnessel (n-es, n-tuple):
("Bea", 19, 3192144) : (string * int * int),
azaz (string * int * int) \equiv {1 = string, 2 = int, 3 = int}.
```
- Egy rekordban a tagok sorrendje közömbös, az értékeket a mezőnév azonosítja.

Deklaratív Programozás, BME, 2001 tavaszi félév

14. előadás (funkcionális programozás)

## FELHASZNÁLÓI ADATTÍPUSOK

## Ennes és típusa (folyt.)

- Egy emnesben a tagok sorrendje meghatározói Pl.  $(2, 1.0) : (\text{int} * \text{real})$ , de  $(1.0, 2) : (\text{real} * \text{int})$ . A két emnes különböző!
- Ennes lehet függvény argumentuma és eredménye, összetett adat eleme stb. Példa: Fibonacci-számok iterációval.  
A definíció:  $F_0 = 0; F_1 = 1; F_n = F_{n-2} + F_{n-1}, n > 1$ .  

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

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

Deklaratív Programozás, BME, 2001 tavaszi félév

14. előadás (funkcionális programozás)

Felhasználói adattípusok 244

## A datatype deklaráció

- person néven új összetett típust hozunk létre:  

```
datatype person = King
| Peer of string * string * int
| Knight of string
| Peasant of string;
```
- Az új típusnak négy *adatkonstruktor*a (röviden: *konstruktor*) van: King, Peer, Knight és Peasant.
- King ún. *adatkonstruktorállandó*, a többi ún. *adatkonstruktorfüggvény*.
- Az adatkonstruktoroknak is van típusuk:  

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

Deklaratív Programozás, BME, 2001 tavaszi félév

14. előadás (funkcionális programozás)

## A datatype deklaráció (folyt.)

- King (király) csak egy van, ezért definiálhatuk konstruktorállandóként.
- A Peer-t (főnemesi) nemesi címe (string), birtokának neve (string) és sorszáma (int) azonosítja.
- A Knight-ot (lovagot) és a Peasant-ot (parasztot) csupán a neve (string) azonosítja.
- Példa a person adattípus alkalmazására:

```
- val persons = [King, Peasant "Jack Cade", Knight "Gawain",
 Peer("Duke", "Norfolk", 9)]
> val persons = [King, Peasant "Jack Cade", ...] : person list
```

- Az egyes esetek mintaillesztéssel választhatók szét.
- Minden esetet le kell fedni mintával; ha nem, figyelmeztetést kapunk.
- A minták tetszőlegesen összetettek lehetnek.

Deklaratív Programozás, BMEF, 2001 tavaszi félév

14. előadás (funkcionális programozás)

Felhasználói adattípusok 247

## A datatype deklaráció (folyt.)

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

```
sirs(p::ps) = sirs ps if Vs.p≠Knight s.
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

14. előadás (funkcionális programozás)

## A datatype deklaráció (folyt.)

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

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

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

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

14. előadás (funkcionális programozás)

Felhasználói adattípusok 248

## A datatype deklaráció (folyt.)

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

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

14. előadás (funkcionális programozás)

## A felsorolásos típus datatype deklarációval

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

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

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

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

14. előadás (funkcionális programozás)

Felhasználói adattípusok 251

## Polimorf adattípusok

- Látnuk, hogy a list *posztfix* pozícióú *típusoperátor*, nem típus: a datatype deklaráció az adatkonstruktorok mellett *típuskonstruktor*t is létrehoz.

- A belső 'a list típushoz hasonló 'a list listát és vele együtt a Nil és a Cons *adatkonstruktorokat* például így definiálhatjuk:

```
datatype 'a list = Nil | Cons of 'a * 'a list;
```

- A Cons *adatkonstruktorfüggvény* alkalmazásával elég körülményes a listák létrehozása. Az 1, 2, 3, 4 sorozatot például így kell megadni:

```
Cons(1, Cons(2, Cons(3, Cons(4, Nil))));
```

- Bevezethetjük az *infix* pozícióú ::: *adatkonstruktoroperátort*:

```
infix 5 ::: ; val op ::: = Cons;
```

- A *hatáspontot* közvetlenül a típusdeklarációban is definiálhatjuk:

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

14. előadás (funkcionális programozás)

## A felsorolásos típus datatype deklarációval (folyt.)

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

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

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

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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

14. előadás (funkcionális programozás)

Felhasználói adattípusok 252

## Polimorf adattípusok: megkülönböztetett egyesítés

- Következő példánk két típus *megkülönböztetett egyesítése*, más néven diszjunkt uniója:

```
datatype ('a, 'b) disun = In1 of 'a | In2 of 'b;
```

- Itt három dolgot definiáltunk:

1. a kétargumentumú disun típusoperátort,
2. az In1 : 'a -> ('a, 'b) disun és
3. az In2 : 'b -> ('a, 'b) disun adatkonstruktorfüggvényeket.

- ('a, 'b) disun az 'a és 'b típusok megkülönböztetett egyesítése.

*Megkülönböztetettnek* nevezzük az egyesítést, mert később is bármikor meg tudjuk mondani, hogy egy ('a, 'b) disun típusú pár egyik vagy másik eleme melyik alaptípusból származik. Az új típusba tartozó értékek In1 x alakúak, ha x 'a típusú, és In2 y alakúak, ha y 'b típusú.

- Az In1 és In2 konstruktorfüggvények olyan *címke*nek tekinthetők, amelyek az 'a típust megkülönböztetik a 'b típustól.

Deklaratív Programozás, BMEF, 2001 tavaszi félév

14. előadás (funkcionális programozás)

## Megkülönböztetett egyesítés (folyt.)

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

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

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

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

```
(* concat d = a d diszjunkt unió In1 címkejű elemeknek konkatenációja
 concat : (string, 'a) disun list -> string *)
```

```
fun concat [] = ""
 | concat (In1 s :: ls) = s ^ concat ls
 | concat (In2 _ :: ls) = concat ls;
```

## Megkülönböztetett egyesítés (folyt.)

- Egy példa `concat` alkalmazására:

```
- concat [In1 "Íi", In2 King, In1 "Skócia"];
> val it = "Íi Skócia : string"
```

- Az `In1` konstruktorfüggvény típusa `'a -> ('a, 'b) disun`, ezért a `string` típusú `"Íi"` argumentumra alkalmazva (`string, 'b`) `disun` típusú érték az eredmény.

- Az `In2` konstruktorfüggvény típusa `'b -> ('a, 'b) disun`, ezért a `person` típusú `King` kifejezésre alkalmazva (`'a, person`) `disun` típusú érték az eredmény.

- Az `[In1 "Íi", In2 King, In1 "Skócia"]` kifejezésben mindkét alaptípust lekövetjük, ezért ennek a listának a típusa: `((string, person) disun) list`.

- Az `[In2 "Íi", In2 King, In1 "Skócia"]` kifejezés kiterjedése hibajelzést eredményez, mert a `'b` típusváltozót nem lehet ugyanabban a kifejezésben egyszer íggy, mászor úgy lekötni.

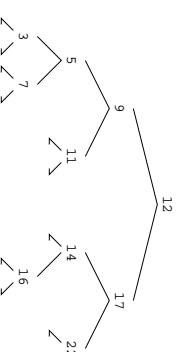
## BINÁRIS FÁK

### Bináris fák datatype deklarációjával

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

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

- Tekintsük például az alábbi fát:



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



## Bináris fák datatype deklarációjával (folyt.)

```
B(B(B(L,3,L),
```

```
5,
```

```
B(L,7,L)
```

```
),
```

```
9,
```

```
B(L,11,L)
```

```
),
```

```
12,
```

```
B(B(L,
```

```
14,
```

```
B(L,16,L)
```

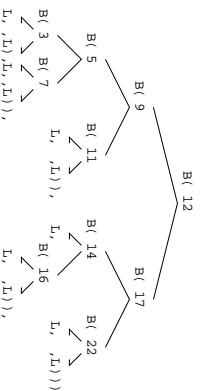
```
),
```

```
17,
```

```
B(L,22,L)
```

```
)
```

```
);
```



Deklaratív programozás, BMF, 2001 tavaszi félév

14. előadás (funkcionális programozás)

Bináris fák 268

## Bináris fák datatype deklarációjával (folyt.)

- Másféle fastruktúrákat is deklarálhatunk, pl.

- kezdhetjük az 'a típusú értékkel, majd folytathatjuk előbb a bal, aztán a jobb részfa megadásával,

- felhasználhatjuk a levelet is értékek tárolására,

- az értéket nem tároló üres csomópont pedig E-vel jelölhetjük.

- A leírtak szerinti bináris fát hoz létre a következő deklaráció:

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

- A rekurzív függvényekhez hasonlóan a rekurzív adattípusok deklarációjában is kell lennie nemrekurzív ágának (ún. triviális esetnek).

- A nemrekurzív ág hiánya miatt az alábbi szintaktikailag helyes deklarációk használhatatlanok:

```
datatype 'a badtree = B of 'a badtree * 'a * 'a badtree;
```

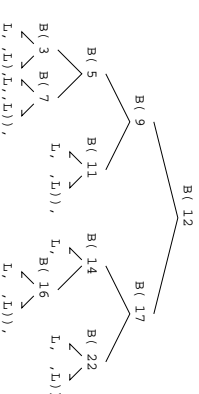
```
datatype 'a badtree = L of 'a badtree | B of 'a badtree * 'a * 'a badtree;
```

Deklaratív programozás, BMF, 2001 tavaszi félév

14. előadás (funkcionális programozás)

## Bináris fák datatype deklarációjával (folyt.)

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



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

Deklaratív programozás, BMF, 2001 tavaszi félév

14. előadás (funkcionális programozás)

Bináris fák 269

## Egyszerű műveletek bináris fákban

- nodes egy fa csomópontjait számlálja meg. Legyen

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

```
(* nodes f = az f fa csomópontjainak a száma
```

```
nodes : 'a tree -> int *)
```

```
fun nodes (N(_, t1, t2)) = 1 + nodes t2 + nodes t1
```

```
 | nodes L = 0;
```

- nodes akkumulátort használó változata (nodesa):

```
fun nodesa f =
```

```
 let (* nodes0(f, n) = n + a csomópontok száma f-ben
```

```
 nodes0 : 'a tree * int -> int *)
```

```
 fun nodes0 (N(_, t1, t2), n) = nodes0(t1, nodes0(t2, n+1))
```

```
 | nodes0 (L, n) = n
```

```
 in nodes0(f, 0)
```

```
 end;
```

Deklaratív programozás, BMF, 2001 tavaszi félév

14. előadás (funkcionális programozás)

## Egyszerű műveletek bináris fákon (folyt.)

- A fa gyökereből a leveléhez vezető úton az élék számát (az út hosszát) az adott level szintjének is nevezzük. A szintek közül a legnagyobbat a fa *mélységének* hívjuk.

- depth egy fa mélységét határozza meg.

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

- depth akkumulátort használó változata (deptha):

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

Deklaratív programozás, BMEF, 2001 tavaszi félév 14. előadás (funkcionális programozás)

SML-Prolog átvezetés 263

## SML-Prolog átvezetés: további példák

### SML

```
fun append xs ys = foldr op:: ys xs
| fakt n = n * fakt (n-1)
| fakt 0 = 1
```

típusos nyelv  
magasabbrendű függvény  
rekurzív  
kivétel  
(pl. fakt negatív számmal)

### Prolog

```
/* Prologban kevésbé használtak
 a magasabbrendű eljárások */
fakt(0, 1).
fakt(N, F) :-
 N>0, M1 is N-1,
 fakt(M1, F1), F is N*F1.
```

típusatlan nyelv  
rekurzív, ritkábban magasabbrendű pred.  
visszalépéses ciklus  
(pl. két lista közös eleme)  
meghívásulás, kivétel

Deklaratív programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

## SML-Prolog átvezetés: párhuzamos két nyelv között

### SML

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

### SML „Prologosztva”

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

### függvény

#### klóz

változó: egyetlen, ismert érték  
egyrányú mintaillesztés  
egyrésztlenű klózváltásztás  
egy eredmény  
egyrányú használat

#### adatkonstruktor-függvény

egy másikba ágyazott függvényhívások

### Prolog

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

### Prolog „SML-estve”

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

### predikátum

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

Deklaratív programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

SML-Prolog átvezetés 264

## Összefoglalás: Prolog programok szemantikája

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

Deklaratív programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

## Ismétlés: A Prolog végrehajtási mechanizmus, dióhéjban

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

Deklaratív programozás, BME, 2001 tavaszi félév

15/16. előadás (logikai programozás)

SML-Prolog átvizelés 267

## 4. fejezet: Prolog programozási módszerek

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

Deklaratív programozás, BME, 2001 tavaszi félév

15/16. előadás (logikai programozás)

## Ismétlés: A Prolog egyesítési algoritmus, dióhéjban

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

Deklaratív programozás, BME, 2001 tavaszi félév

15/16. előadás (logikai programozás)

SML-Prolog átvizelés 268

## Prolog programozási módszerek: tartalomjegyzék

- A keresési tér szűktése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzívó és akkumulátorok
- Algoritmusok Prologban
- Megoldások gyűjtése és felsorolása
- Megoldágyűjtő eljárások
- Meta-logikai eljárások
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- „Hagyományos” beépített eljárások
- Nyelvtani elemzés

Deklaratív programozás, BME, 2001 tavaszi félév

15/16. előadás (logikai programozás)

## Prolog nyelvi eszközök a keresési tér szűkítésére

### Eszközök

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

Miért vágunk le ágakat a keresési térben?

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

## Példák a vágó eljárás használatára

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

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

% pirosak(+L, -Pirosak): Pirosak az L piros/1 funktorú elemekből áll.
pirosak([], []).
pirosak([_|_], [E|Pk]) :-
 E = piros(_, _),
 pirosak([_|_], Pk).

/* \+ _E = piros(_), */ pirosak(EK, PK).
```

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

## A vágó definíciója

### Segédfogalom

- Egy cél *szülője* az a cél, amelyik az őt tartalmazó klóz fejével illesztődött.
- Pl. a last([E], E) :- !. klózbeli vágó szülője lehet a last([\_], X) hívás.
- A g (ancestors) nyomonkövetési parancs kirúgja a kurrens cél őseit (szülőjét, annak szülőjét, stb.)

A vágó végrehajtása:

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

A vágás kétféle választási pontot szüntet meg:

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

## A vágó által megszüntetett választási pontok

```
% vágó nélküli példa
q(X) :- s(X).
q(X) :- t(X).
```

```
% ugyanaz a példa vágóval
```

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

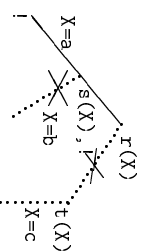
```
s(a). s(b). t(c).
```

```
% a vágó nélküli példa futása
:- q(X), write(X), fail.
----> abc
```

```
abc
```

```
% a vágót tartalmazó példa futása
```

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



Deklaratív Programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

## Példák a diszjunktív feltételes szerkezet használatára

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

```
% last(+L, ?E): az L nem üres lista utolsó eleme E.
```

```
last([E|_], Last) :-
 (L = [] -> Last = E
 ; last(L, Last)
).
```

```
% pirosak(+L, ?Pirosak): Pirosak az L piros/1 funktorú elemelből áll.
```

```
pirosak([], []).
pirosak([E|_], Pk) :-
 (E = piros(_) -> Pk = [E|Pk0]
 ; Pk = Pk0
), pirosak(Ek, Pk0).
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

## A diszjunktív feltételes szerkezet definíciója

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

```
p :-
 ...
 (felt1 -> akkor1 segéd(...)
 ; felt2 -> akkor2 ...
 ; ... =>
 ; egyébként segéd(...) :- felt1, !, akkor1.
) segéd(...) :- felt2, !, akkor2.
 ...
 segéd(...) :- egyébként.
```

- Az egyébként alternatíva elmaradhat, ilyenkor a megfelelő klóz is elmarad.
- SICStus módban a felt részekben vágó nem lehet, ISO módban lehet, de hatásköre (szülője) a felt rész.
- Az akkor részekben lehet vágó. Ennek hatásköre, a  $\rightarrow$  nyílból generált vágóval ellentétben, a teljes p predikátum (távolbatható vágó).

Deklaratív Programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

## A vágás első alapesete — klóz mellett való elkötelezés

- A klóz melletti elkötelezés általában egyszerű feltételes szerkezetet jelent.
 

```
szülő :- feltétel, !, akkor.
szülő :- egyébként.
```
- A vágó szükségtelemné teszi a feltételt negációjának végrehajtását a többi klózban. A logikailag tiszta, de nem hatékony alak:
 

```
szülő :- feltétel, akkor.
szülő :- \+ feltétel, egyébként.
```

A fenti két alak csak akkor ekvivalens, ha feltétel egyszerű, nincs benne választás.
- Analógia: ha a, b és c logikai változók (pl. Pascalban), akkor
 

```
if a then b else c ≡ a ∧ b ∨ ¬ a ∧ c
```
- A vágó által kiváltott negált feltételt célszerű kommentként jelezni:
 

```
szülő :- feltétel, !, akkor.
szülő :- /* \+ feltétel, */ egyébként.
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

## Feltételes szerkezetek

### Feltételes szerkezet — példa

```
% abs(X, A): A az X abszolút értéke.
abs(X, A) :- X < 0, !, A is -X.
abs(X, X) /* :- X >= 0 */.
```

### Általános alak

```
p :- felt1, !, akkor1.
p :- felt2, !, akkor2.
...
p :- egyébként.
```

### Diszjunktív feltételes szerkezet

```
abs(X, Y) :-
 (X < 0 -> Y is -X
 ; Y = X
).
```

### Általános alak

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

Deklaratív programozás, BMEF, 2001 tavaszi félév 15/16. előadás (logikai programozás)

## A bevezető példáknak a vágás alapszabályát betartó változata

```
% fakt(+N, ?F): Ni = 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([_], Last) :- !, Last = E.
last([_|_], E) :- !, last(L, E).
```

```
% pirosak(+L, ?Pirosak): Pirosak az L piros/1 funktorú elemelhől áll.
pirosak([], []).
pirosak([_|_], Pk) :-
```

```
 E = piros(_), !, Pk = [E|Pk0], pirosak(Ek, Pk0).
pirosak([_|_], Pk) :-
 /* \+_ _E = piros(_), */ pirosak(Ek, Pk).
```

Megjegyzés: a diszjunktív alakban a feltételek eleve explicitiek, nincs fejillesztési probléma.

Deklaratív programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

## Feltételes szerkezetek és fejillesztés

- **Vigyázat:** a tényleges feltétel részét képezik a fejbéli egyesítések!

```
% a vágót fej-egyesítés előzi meg % az egyesítés explicitté téve:
abs(X, X) :- X >= 0, !. abs(X, A) :- A = X, X >= 0, !.
abs(X, A) :- A is -X. abs(X, A) :- A is -X.
```

- A fej-egyesítés gondot okozhat, ha az eljárást ellenőrzésre használjuk:  
| ?- abs(10, -10). ---> yes

- A megoldás a **vágás alapszabályai**:

- A kimenő paraméterek értékadását mindig a vágó után végezzük!  
abs(X, A) :- X >= 0, !, A = X.  
abs(X, A) :- A is -X.

- Ez nemcsak általánosabban használható, hanem hatékonyabb kódot is ad!
- („*kimenő*” *paraméterek* — vágó alkalmazásakor általában nincs többirányú használat :-)

Deklaratív programozás, BMEF, 2001 tavaszi félév 15/16. előadás (logikai programozás)

## Példasor: max(X, Y, Z): X és Y maximuma Z.

- 1. változat, tiszta Prolog. Lassú, választási pontot hagy.

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

- 2. változat, zöld vágóval. Lassú, nem hagy választási pontot.

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

- 3. változat, vörös vágóval. Gyors, nem hagy választási pontot, de nem használható ellenőrzésre, pl. | ?- max(10, 1, 1) sikerül.

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

- 4. változat, vörös vágóval. Helyes, gyors és nem hagy választási pontot.

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

Deklaratív programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

## A vágás második alapesete — első megoldásra való megszorítás

Mikor használjuk az első megoldásra megszorító vágót?

- belhelyettesítést nem okozó, eldöntendő kérdés esetén;
- feladat-specifikus optimalizálásra;
- végtelen választási pontot létrehozó eljárások hasznosítására.

Eldöntendő kérdés: eljárás-hívás csupa bemenő paraméterrel

```
% van_elég_hosszú_út(+N, +A, +B, +Min):
% A és B között van N lépéses út, amelynek összhossza legalább Min km.
van_elég_hosszú_út(N, A, B, Min) :-
 útvonali(N, A, B, Hossz), Hossz >= Min, !.
```

Eldöntendő kérdés esetén általában nincs értelme többszörös választ adni/várni.

Deklaratív Programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

## Feladat-specifikus optimalizálás — 2. példa

A feladat: megkeresendő egy lista elején álló „plató” hossza (platónak hívjuk a csupa azonos elemből álló folytonos részhalmazt).

```
% Az L lista első eleme H-szor ismétlődik a lista kezdőszelvényként.
kezdethossz(L, H) :-
 L = [E|_], append(Ek, Farok, L),
 \+ Farok = [E|_], !, % vörös vágó
 /* egyformák(Ek, E), */
 length(Ek, H).
/*
% egyformák(Ek, E): Az Ek lista minden eleme E.
egyformák([], _).
egyformák([E|EK], E) :-
 egyformák(EK, E).
*/
| ?- kezdethossz([1,1,1,2,3,5], H).
H = 3 ? ; no
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

## Feladat-specifikus optimalizálás

- Példafeladat: egy nem-negatív számokból álló lista első pozitív eleme
  - Első megoldás, rekurzíóval (mémóriki :-)
- ```
első_poz_elem([0|L], EP) :- !, első_poz_elem(L, EP).
első_poz_elem([E|_], EP) /* :- EP > 0 */.
```
- Második megoldás, visszalépéses kereséssel (matematikusai :-)
- ```
első_poz_elem(L, EP) :-
 append(Nk, [E|_], L), EP > 0, \+ poz_elem(Nk, _).
poz_elem(L, P) :- member(P, L), P > 0.
```
- Harmadik megoldás, vágóval (Prolog hacker megoldása :-)
- ```
első_poz_elem(L, EP) :- member(EP, L), EP > 0, !.
```
- **Figyelem: a harmadik megoldás épít a member/2 felsorolási sorrendjére!**

Deklaratív Programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

Végtelen választás megszelídítése: memberchk

% memberchk(X, L): "X eleme az L listának" kérdés első megoldása.

```
% 1. változat
memberchk(X, L):-
    member(X, L), !.
% 2. ekvivalens változat
memberchk(X, [X|_]) :- !.
memberchk(X, [_|L]) :- memberchk(X, L).
```

memberchk/2 használata

- Eldöntő kérdésben (visszalépéskor nem keresi végig a lista maradvékát.)
- ```
memberchk(1, [1,2,3,4,5,6,7,8,9])
```
- Nyílt végű lista elemévé tesz, pl.:
- ```
| ?- memberchk(1,L), memberchk(2,L), memberchk(1,L).
L = [1,2|_A] ?
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

Nyílt végű listák kezelése memberchk segítségével: szótárprogram

szótárak(Sz):-

```
read(M-A), !, % kifejezést olvas be és egyesíti az argumentummal
memberchk(M-A,Sz),
write(M-A), nl,
szótárak(_).
```

Egy futása:

```
| ?- szótárak(Sz).          | : alma-X.
| : alma-apple.           | alma-apple
| : alma-apple           | : X-pear.
| : korte-pear.          | korte-pear
| : korte-pear           | : vege.
                           | : vege.
Sz = [alma-apple,korte-pear|_A] ?
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

Vezérlési eljárások 287

Vezérlési eljárások: call/1

- Vezérlési eljárás: A Prolog végrehajtási mechanizmushoz kapcsolódó beépített eljárás (pl. a vágó).
- A vezérlési eljárások többsége *magasabrendű* eljárás, azaz olyan eljárás, amely egy vagy több argumentumát eljárásnévként értelmezi. (A magasabrendű Prolog eljárásokat szokás *meta-eljárásnak* is hívni.)
- A meta-eljárások fő képviselője és alapvető építőeleme a call/1:
 - Hívási minta: call(+Cél)
 - Argumentumok: Cél egy „meghívható kifejezés” (callable), azaz struktúra, vagy atom.
 - Jelentése: Cél igaz.
 - Hatása: a Cél kifejezést eljárásnévessé alakítja és meghívja
- Ha klóntörzsben célként szerepel egy X változó, akkor azt a rendszer egy call(X) hívássá alakítja át.
- Cél-ban további vezérlési eljárások is előfordulhatnak, pl. ; ;/2, ; ;/2. A Cél-bei vágó csak a call belsejében vág (szülője a call(Cél) hívás).

Deklaratív programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

VEZÉRLÉSI ELJÁRÁSOK

Vezérlési eljárások 288

call/1: példák

```
| ?- [user].
| kétszer(Hívás) :- call(Hívás), Hívás.
| ~D
| {consulted user in module user, 0 msec 224 bytes}
yes
? - kétszer(write(ba)), nl.
baba
yes
| ? _Cél = (kétszer(write(ba)), write(' ')), kétszer(_Cél), nl.
baba baba
yes
| ?- listing(kétszer).
| kétszer(A) :-
|   call(user:A),
|   call(user:A).
yes
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

call/1 példa: futási időt mérő meta-eljárás

```
% Kírítja Goal első megoldásának előállításához vagy a meghívásához
% szükséges időt, a Txt szöveg kíséretében (lásd: példák/call_koltsege.pl).
time(Txt, Goal) :-
    statistics(runtime, [T0,_]), % T0 az indítás óta eltelt CPU idő,
                                % msec-ban (személyűjtés nélkül).
    ( call(Goal) -> Res = true
      ; Res = false
    ),
    statistics(runtime, [T1,_]), T is T1-T0,
    format('~w futási idő = ~3d sec\n', [Txt,T]),
    Res = true.
```

A call/1 viszonylag költséges: egy 1414 hosszú lista megfordítása nrev-vel (kb 1 millió append hívás), minden append körül egy felesleges call-lal ill. anélkül:

	call nélkül	call-lal
lefordítva	0.220 sec	9.710 sec
interpretálva	1.850 sec	11.940 sec

Deklaratív Programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

További vezérlési eljárások

- A „nem-bizonyíthatóságot” ellenőrző \+ eljárás definíciója:


```
\+ X :- call(X), !, fail.
\+ _X.
```
- Az első megoldásra megszorító once beépített eljárás definíciója:


```
once(X) :- call(X), !.
```
- A true ill. fail beépített eljárás mindig sikerül ill. mindig meghiúsul.
- A repeat beépített eljárás egy végtelen választási pontot hoz létre:


```
repeat.
repeat :- repeat.
```
- A repeat eljárást mindig egy vágóval kell semlegesíteni. Példa:


```
bc :- repeat, read(Expr),
    ( Expr = end_of_file -> true
      ; Res is Expr, write(Expr = Res), nl, fail
    ), !.
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

Vezérlési szerkezetek mint eljárások

- A call/1 argumentumában azért szerepelhetnek vezérlési szerkezetek, mert ezek maguk beépített eljárásként is jelen vannak a Prolog rendszerben:
 - (;)/2: konjunkció.
 - (;)/2: diszjunkció.
 - (->)/2: if-then.
 - (;)/2: if-then-else.
- A call-ban szereplő vezérlési szerkezetek lényegében ugyanúgy futnak, mint az interpretált (consult-tal betöltött) kód.
 - Példa


```
| ? - kétszer((member(X, [a,b,c,d]), write(X), fail ; nl)).
abcd
abcd
abcd
true ?
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

15/16. előadás (logikai programozás)

DETERMINIZMUS ÉS INDEXELÉS

Determinizmus

- Egy eljárshívás *determinisztikus*, ha (legfeljebb) egyféleképpen sikerülhet.
- Egy eljárshívásnak egy sikeres végrehajtása *determinisztikusan futott le*:
 - ha nem hagyott választási pontot a híváshoz tartozó részfáiban:
 - választásmentesen futott le, azaz létre sem hozott választási pontot (figyelni: ez a Prolog megvalósítástól függ!); vagy
 - létrehozott ugyan választást, de megszüntette (kimerítette, levágta).

- A SICStus Prolog nyomkövetésében ? jelzi a *nem*determinisztikus lefutást:

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

Indexelés

- Mi az indexelés?
 - egy hívásra illeszthető klczók gyors kiválasztása,
 - egy eljárás klczainak fordítási idejű csoportosításával,
- A legtöbb Prolog rendszer, így a SICStus Prolog is, az első fejl-argumentum alapján indexel (first argument indexing).
- Az indexelés alapja az első fejl-argumentum külső funktora:
 - C szám vagy névkonstans esetén C/0;
 - R nevű és N argumentumú struktúra esetén R/N;
 - változó esetén nem értelmezett.
- Az indexelés megvalósítása:
 - Fordításkor a funktorokhoz elkészítjük az illeszthető klczók részhalmozatát.
 - Futáskor lenyegében konstans idő alatt választunk a részhalmozatok közül.
 - *Kontos*: ha egyetlen a részhalmoz, nem hozunk létre választási pontot!

A determinisztikus lefutás

- Mi a determinisztikus lefutás haszna?
 - a futás gyorsabb lesz,
 - a tárigény csökken,
 - más optimalizálások (pl. jobbrekurzió) alkalmazható.

- Hogyan ismeri fel a fordító azt, hogy nem kell választási pont?

- indexelés (indexing)
- vágók és feltételes szerkezetek

- Az alábbi definíciók esetén a p(*Monitor*, Y) hívás nem hoz létre választási pontot (a 2. definíció esetén a p(*Var*, Y) sem):

p(1, a).	p(1, Y) :- !,	p(X, Y) :-
p(2, b).	Y = a.	(X > 1 -> Y = a
	p(., b).	; Y = b
).

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(s, z). | / * (5) * / | |
- A p(A, B) hívással illesztendő klczokalmaz:
 - {(1) (2) (3) (4) (5)} ha A változó;
 - {(1) (2)} ha A = 0;
 - {(2) (3) (4)} ha A fő funktora s/1;
 - {(2) (5)} ha A = s;
 - {(2)} minden más esetben.
 - Példák hívásokra:
 - p(3, Y) nem hoz létre választási pontot.
 - p(s(1), Y) létrehoz választási pontot, de determinisztikusan fut le.
 - p(s(0), Y) nemdeterminisztikusan fut le.

Struktúrák, változók a fejargumentumban

- Azonos funktorú struktúrák az első fejargumentumban:
 - Ha a klózok szétválasztásához szükség van az első (struktúra) argumentum részére is, akkor érdemes segédeljártást bevezetni.
 - Például $p/2$ és $q/2$ ekvivalens, de q (*Monom*, Y) determinisztikusan fut le!

$p(0, a)$.	$q(0, a)$.	$q_segged(0, b)$.
$p(s(0), b)$.	$q(s(X), Y) :-$	$q_segged(1, c)$.
$p(s(1), c)$.	$q_segged(X, Y)$.	
$p(9, z)$.	$q(9, z)$.	
 - Fejillesztés kiváltása egyenlőséggel (vö. SML rétegelt minta)
 - Az indexelés figyelembe veszi a törzs elején szereplő egyenlőséget:
 $p(X, \dots) :- X = K1f, \dots$ esetén $K1f$ funktorra szerint indexel.
 - Példa: lista hosszának reciproka, üres lista esetén 0:


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

Deklaratív Programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

268

Listakezelő eljárások indexelése: példák

- Az `append/3` választásmentesen fut le (összefűzésre):


```
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).
```
- A `Last/2` közvetlen megfogalmazása nondeterminisztikusan fut le:


```
% Last(L, E): Az L lista utolsó eleme E.
Last([E], E).
Last([_|L], E) :- Last(L, E).
```
- Érdemes segédeljártást 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).
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

269

Indexelés és aritmetika, indexelés és listák

- Aritmetikai elágazások
 - Az indexelés nem foglalkozik aritmetikai vizsgálatokkal.
 - Pl. az $N = 0$ és $N > 0$ feltételek nem „zárják ki” egymást.
 - Az alábbi `fakt(N, F)` lefutása nondeterminisztikus:


```
fakt(0, 1).
fakt(N, F) :- N > 0, N1 is N-1, fakt(N1, F1), F is N*F1.
```
 - Listakezelő eljárások
 - Gyakran kell az üres és nem-üres lista esetét szétválasztani.
 - A bemenő lista-argumentumot célszerű az első argumentum-pozícióba tenni.
 - Az `[]` és `[...|...]` eseteket az indexelés megkülönbözteti (funktoruk: `?'[]'/0` ill. `?'|'/2`).
 - A két klóz sorrendje nem érdekes (feltéve, hogy zárt listával hívjuk az első pozíción) — de azért tegyük a leállító klózt mindig előre.

Deklaratív Programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

269

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)` nemi

$p(1, Y) :- 1, Y = 2.$	$q(1, 2) :- 1.$	$q(1)$
$p(X, X).$	$q(X, X).$	$q(2)$
$Arg1=1 \rightarrow (1), Arg1 \neq 1 \rightarrow (2)$	$Arg1=1 \rightarrow \{(1), (2)\}, Arg1 \neq 1 \rightarrow (2)$	
 - Csak akkor tudjuk fordításkor kizárni a vágót követő klózokat, ha garantált, hogy az adott fő funktor esetén a vágót elérjük. Ennek feltételei:
 - az első argumentumban konstans, vagy legáltalánosabb struktúra legyen,
 - a további argumentumok változók legyenek,
 - a fejben az összes változóelőfordulás különböző legyen,
 - a törzs első hívása a vágó (megengedve a fejillesztést kiváltó `=t`).
 - Példa: `p(s(A, B, C), D, E) :- 1, ...`
 - Ez egy újabb érve a vágás alapszabálya mellett:

A kimenő paraméterek értékadását mindig a vágó után végezzük!

Deklaratív Programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

300

A vágó és az indexelés hatékonysága

- Egy Fibonacciszervi sorozat: $f_1 = 1$, $f_2 = 2$; $f_n = f_{[n/4]} + f_{[2n/3]}$, $n > 2$

% determinisztikus	% determ. lefutási	% választásmentes
fib(1, 1).	fibc(1, 1) :- 1.	fibci(1, F) :- 1, F = 1.
fib(2, 2).	fibc(2, 2) :- 1.	fibci(2, F) :- 1, F = 2.
fib(N, F) :-	fibc(N, F) :-	fibci(N, F) :-
N > 2,	N2 is N*3//4,	N > 2,
N3 is N*2//3,	N3 is N*2//3,	N2 is N*3//4,
fib(N2, F2),	fibc(N2, F2),	N3 is N*2//3,
fib(N3, F3),	fibc(N3, F3),	fibci(N2, F2),
F is F2+F3.	F is F2+F3.	fibci(N3, F3),
		F is F2+F3.

- Futási idők $N = 2000$ esetén

	fib	fibc	fibci
futási idő	4410 ms	4060 ms	3820 ms
meghívásulási idő	730 ms	0 ms	0 ms
Összesen	5140 ms	4060 ms	3820 ms

Deklaratív programozás, BMTF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

JOBBREKURZÍÓ ÉS AKKUMULÁTOROK

Választás-mentesség diszjunktív feltételes szerkezetek esetén

- Feltételes szerkezet végrehajtásakor általában választási pont jön létre.
- A SICStus Prolog a „felt \rightarrow akkor ; egyébként)” szerkezetet választásmentesen hajtja végre, ha a felt konjunktív tagjai csak:
 - aritmetikai összehasonlító eljárás hívások (pl. <, =, =:=), és/vagy
 - kifejezés-típust ellenőrző eljárás hívások (pl. atom, number), és/vagy
 - általános összehasonlító eljárás hívások (ld. később, pl. @<, @=<, ==).
- Analóg módon választásmentes kód keletkezik a „felt, ;, akkor.” klózból, ha fej argumentumai különböző változók, és felt olyan mint fent.

- Például választásmentes kód keletkezik az alábbi definíciókból:

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

Deklaratív programozás, BMTF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Jobbrekurzív (farok-rekurzív, tail-recursion) optimalizálás

- Az általános rekurzív költséges, helyben és időben is.
- Jobbrekurzítoról beszélünk, ha
 - a rekurzív hívás a klóztörzs utolsó helyén van, vagy az utolsó helyen szereplő diszjunktív egyik tagjának utolsó helyén, stb., és
 - a rekurzív hívás pillanatában nincs választási pont a predikátumban, tehát a rekurzív hívást megelőző célok determinisztikusan futottak le.
- Jobbrekurzív optimalizálás: az utolsó hívás végrehajtása *előtt* a predikátum által lefoglalt hely felszabadul ill. szemétyűtésre alkalmassá válik.
- Ez az optimalizálás nemcsak rekurzív hívás esetén, hanem minden *utolsó* hívás esetén megvalósul — utolsó hívás optimalizálás (last call optimisation).
- A jobbrekurzív 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).

Deklaratív programozás, BMTF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Predikátumok jobbrekurzív alakra hozása — listaösszeg

- A listaösszegzés „természetes”, nem jobbrekurzív definíciója:


```
% sum(+L, ?S): Az L számlista elemeinek összege S.
sum([], 0).
sum([X|L], S):- sum(L,S0), S is S0+X.
```
- Első jobbrekurzív változat, csak ellenőrzésre használható:


```
% sum1(+L, +S): Az L számlista elemeinek összege S.
sum1([], 0).
sum1([X|L], S) :- /* S is S0+X helyett: */ S0 is S-X, sum1(L, S0).
```
- Második jobbrekurzív változat, csak kimni tudja az eredményt:


```
% sum2(+L): Az L számlista elemeinek összegét kiírja.
sum2(L):- sum2(L, 0).

% sum2(+L, +S0): Az L lista S0-lal növelt összegét kiírja.
sum2([], S) :- write(S), nl.
sum2([X|L], S0):- S1 is S0+X, sum2(L, S1).
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Akkumulátorok használata

- Az akkumulátorpárak a hagyományos, „változtatható” változók megfelelői.
- Az általános séma:


```
p(..., A0, A):-
    q0(..., A0, A1), ...,
    q1(..., A1, A2), ...,
    qn(..., An, A).
```
- A sum3/3 második kléza ilyen alakra hozva:


```
sum3([X|L], S0, S):- plus(X, S0, S1), sum3(L, S1, S).
plus(X, S0, S) :- S is S0+X.
```
- Akkumulátorváltozók elnevezési konvenciója: kezdőérték: *Vál0*; közbülső értékek: *Vál1*, ..., *VálN*; végérték: *VálL*.
- A Prolog akkumulátorpár nem más mint a funkcionális programozásból ismert gyűjtőargumentum és a függvény eredményének együttese.

Deklaratív Programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Jobbrekurzív listaösszeg — akkumulátorpár segítségével

- Harmadik változat: teljes értékű jobbrekurzív lista-összegző:


```
% sum3(+L, ?S): Az L számlista elemeinek összege S.
sum3(L, S):- sum3(L, 0, S).

% sum3(+L, +S0, ?S): Az L lista elemeit hozzáadva S0-hoz kapjuk S-et.
sum3([], S, S).
sum3([X|L], S0, S):-
    S1 is S0+X, sum3(L, S1, S).
```
- Az *akkumulátor* fogalma:
 - A sum3(L, S0, S) predikátumban az S0 és S argumentumok egy akkumulátorpárt alkotnak.
 - Az akkumulátorpár két része egy változó mennyiség (a példában az *összeg*) különböző időpontokban vett értékeit mutatja:
 - S0 az *összeg* értéke a sum3/3 *meghívásakor*;
 - S az *összeg* értéke a sum3/3 *lefutása után*.

Deklaratív Programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Akkumulátorok használata — folytatás

- Három lista összege


```
% sum_3_lists(+L, +LL, +LLL, +S0, ?S): Az L, LL, LLL számlisták
% összegeinek összege S-S0
sum_3_lists(L, LL, LLL, S0, S) :-
    sum3(L, S0, S1), sum3(LL, S1, S2), sum3(LLL, S2, S).
```
- Többszörös akkumulálás — listák összege és négyzetösszege


```
% sum12(+L, +S0, ?S, +Q0, ?Q): S-S0 =∑ Li, Q-Q0 = ∑ Li*Li
sum12([], S, S, Q, Q).
sum12([X|L], S0, S, Q0, Q):-
    S1 is S0+X, Q1 is Q0+X*X, sum12(L, S1, S, Q1, Q).
```
- Többszörös akkumulátorok összehasonlása


```
% sum12(+L, +S0/Q0, ?S/Q): S-S0 =∑ Li, Q-Q0 = ∑ Li*Li
sum12([], SQ, SQ).
sum12([X|L], S0/Q0, SQ):-
    S1 is S0+X, Q1 is Q0+X*X, sum12(L, S1/Q1, SQ).
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Korábbi listakezelő predikátumok

- A revapp mint akkumuláló eljárás


```
% revapp(Xs, L0, L): Xs megfordítását L0 elé fűzve kapjuk L-t.
% Másképpen: Xs megfordítása L-L0.
revapp([], L, L).
revapp([X|Xs], L0, L) :-
    L1 = [X|L0], revapp(Xs, L1, L).
```
- Az L-L0 jelölés (különbséglista): az a lista amelyet úgy kapunk, hogy L végétől elhagyjuk L0-t (előfeltétel: L0 szuffixuma L-nek).
- Az append is tekinthető akkumuláló eljárásnak (a 2. és 3. arg. felcserélv).

A változtatás: az L0 elejétől sorra elhagyjuk Xs elemét, végül marad L.

```
% append(Xs, L, L0): L0 elejétől Xs elemeit lehagyva marad L.
% Másképpen: Xs = L0-L.
append([], L, L).
append([X|Xs], L, L0) :-
    L0 = [X|L1], append(Xs, L, L1).
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

$a^n b^n$ alakú sorozatok (folyt.)

- Harmadik megoldás, n lépés


```
anbn(N, L) :-
    anbn(N, [], L).
```

% anbn(N, L0, L): Az L-L0 lista N db a-ból és azt követő N db b-ből áll.

```
anbn(0, L0, L) :- !, L = L0.
anbn(N, L0, [a|L]) :-
    N > 0,
    N1 is N-1,
    anbn(N1, [b|L0], L).
```
- A második klóz nem jobbrekurzív változata


```
anbn(N, L0, L) :-
    N > 0, N1 is N-1,
    L1 = [b|L0], % 1. lépés: L0 elé b => L1
    anbn(N1, L1, L2), % 2. lépés: L1 elé a~M1 b~N1 => L2
    L = [a|L2]. % 3. lépés: L2 elé a => L
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Egy mintafeladat: $a^n b^n$ alakú sorozat előállítás

- Első megoldás, $3n$ lépés


```
% anbn(N, L): Az L lista N db a-ból
% és azt követő N db b-ből áll.
anbn(N, L) :-
    an(N, a, AN),
    an(N, b, BN),
    append(AN, BN, L).
```
- Második megoldás, $2n$ lépés

<pre>% an(N, A, L): L az A elemet N-szer % tartalmazó lista an(0, _A, L) :- !, L = []. an(N, A, [A L]) :- N > 0, N1 is N-1, an(N1, A, L).</pre>	<pre>% an(N, A, L0, L): L-L0 az A % elemet N-szer tartalmazó lista an(0, _A, L0, L) :- !, L = L0. an(N, A, L0, [A L]) :- N > 0, N1 is N-1, an(N1, A, L0, L).</pre>
--	---

Deklaratív programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Összetettebb adatstruktúrák akkumulálása

- Az adatstruktúra:


```
% :- type bfa --> ures ; bfa(int, bfa, bfa).
```
- A fa csomópontjaiban tároljuk a számréteket, a levelek nem tárolnak információt.
- Egészék gyűjtése rendezett bináris fában
 - beszur(BFa0, E, BFa): Az E egész számnak a BFa0 fába való beszurása a BFa bináris fát eredményezi.
 - Itt BFa0 és BFa egy akkumulátor-pár, de az indexelés érdekében BFa0 az első argumentum-pozícióba kerül.
- Példafutás:


```
| ?- beszur(ures, 3, Fa0), beszur(Fa0, 1, Fa1), beszur(Fa1, 5, Fa2).
Fa0 = bfa(3,ures,ures),
Fa1 = bfa(3,bfa(1,ures,ures),ures),
Fa2 = bfa(3,bfa(1,ures,ures),bfa(5,ures,ures)) ?
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Akkumulálás bináris fákkal

● Elem beszúrása bináris fába

```
% beszur(BF0, E, BF): E beszúrása BF0 rendezett fába
% a BF rendezett fát adja
% :- pred beszur(bfa::in, int::in, bfa::out).
beszur(ures, Elem, bfa(Elem, ures, ures)).
beszur(BF0, Elem, BF):-
    BF0 = bfa(E,B,J), % az indexelés működik!
    ( Elem = E -> BF = BF0
    ; Elem < E ->
        BF = bfa(E,B1,J),
        beszur(B, Elem, B1)
    ; BF = bfa(E,B,J1),
        beszur(J, Elem, J1)
    ).
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Akkumulálás bináris fákkal — folyt.

● Bináris fa konverziója listává

```
% bfa_lista(BF, L0, L): A BF fa levelei az L-L0 listát adják.
% :- pred bfa_lista(bfa::in, list(int)::in, list(int)::out).
bfa_lista(ures, L, L).
bfa_lista(bfa(E, B, J), L0, L):-
    bfa_lista(J, L0, L1),
    bfa_lista(B, [E|L1], L).
```

● Rendezés bináris fával

```
% L lista rendezettje R.
% :- pred rendez(list(int)::in, list(int)::out).
rendez(L, R):-
    lista_bfa(L, ures, BF), bfa_lista(BF, [], R).

| ?- rendez([1,5,3,1,2,4], R).
R = [1,2,3,4,5] ? ;
no
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Akkumulálás bináris fákkal — folyt.

● Lista konverziója bináris fává

```
% lista_bfa(L, BF0, BF): L elemeit beszúrva BF0-ba kapjuk BF-t.
% :- pred lista_bfa(list(int)::in, bfa::in, bfa::out).
lista_bfa([], BF, BF).
lista_bfa([E|_], BF0, BF):-
    beszur(BF0, E, BF1),
    lista_bfa(L, BF1, BF).

| ?- lista_bfa([3,1,5], ures, BF).
BF = bfa(3,bfa(1,ures,ures),bfa(5,ures,ures)) ? ;
no

| ?- lista_bfa([3,1,5,1,2,4], ures, BF).
BF = bfa(3,bfa(1,ures,bfa(2,ures,ures)),
        bfa(5,bfa(4,ures,ures),ures)) ? ;
no
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

IMPERATÍV PROGRAMOK ÁTÍRÁSA PROLOGBA

Hogyan írjunk át imperatív nyelvű algoritmust Prolog programmá?

- Példafeladat: Hatékony hatványozási algoritmus
- Alaplépés: a kitevő felezése, az alap négyzetre emelése.
- Lényegében a kitevő kettes szárendszerbeli 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;
}

```

Deklaratív programozás, BME, 2001 tavaszi félév

17/18. előadás (logikai programozás)

A C ciklus és a Prolog eljárás kapcsolata

- A ciklust megvalósító Prolog eljárás minden pontján minden C változónak megfeleltethető egy Prolog változó (pl. h_0, h_1, \dots):
- A ciklusmag elején a C változók a megfelelő Prolog argumentumban levő változónak felelnek meg.
- Egy C értékadásnak egy új Prolog változó bevezetése felel meg, az ezután kódban az új változó felel meg a C változónak.
- Ha a diszjunkció egyik ága megváltoztat egy változót, akkor a többi ágon is be kell vezetni az új Prolog változót, a régiivel azonos értékkel (ld. `if (n & 1) ...`).
- A C ciklusmag végén a Prolog eljárást vissza kell hívni, argumentumaiban az egyes C változónak megfeleltetett Prolog változóval.
- A C ciklus *ciklus-invariánsa* nem más mint a Prolog eljárás fejkommentje, a példában:

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

Deklaratív programozás, BME, 2001 tavaszi félév

17/18. előadás (logikai programozás)

A hatv C függvénynek megfelelő Prolog eljárás

- A függvény eredménye a reláció utolsó arg.-a: `hatv(+A, +H, ?E): AH = E.`
- A ciklusnak segédeljárás felel meg: `hatv(+A0, +H0, +E0, ?E): A0H0 * E0 = E.`
- Az »a« és »h« C változónak az »+A« és »+H« bemenő paraméterek, az »e« C változónak az »+E0, ?E« *akkumulátor-pár* felel meg.

<pre> hatv(A, H, E) :- hatv(A, H, 1, E). </pre>	<pre> int hatv(int a, unsigned h) { int e = 1; while (h > 0) { if (h > 0) e *= a; h >>= 1; a *= a; goto ism; } return e; } </pre>
---	---

Deklaratív programozás, BME, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Programhelyesség-bizonyítás

- Egy algoritmus (függvény) specifikációja:
 - *előfeltétel*: a bemenő paramétereknek teljesíteniük kell ezeket,
 - *utófeltétel*: a paraméterek és az eredmény kapcsolatát írják le.
- Egy algoritmus *helyes*, ha minden, az előfeltételeket kielégítő adatra a függvény hibátlanul lefut, és eredményére fennállnak az utófeltételek.
- Példa: `x = mfoku_gyok(a, b, c)`
 - előfeltételek: `b*b-4*a*c >= 0, a ≠ 0`
 - utófeltétel: `a**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.

Deklaratív programozás, BME, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Ciklikus programok helyességének bizonyítása

- A ciklusokat „fel kell vágni” egy *ciklus-invariáns*-sal, amely:

- az előfeltételekből és a ciklust megelőző értékadásokból következik,
- ha a ciklus elején fennáll, akkor a ciklus végén is (indukció),
- belőle és a leállási feltételből következik a ciklus utófeltétele.

```
int hatv(int a0, unsigned h0)      /* utófeltétel: hatv(a0, h0) = a0^h0 */
{ int e = 1, a = a0, h = h0;
  while (h > 0)
  { /* ciklus-invariáns: a0^h0 == e*a^h */
    /* indukálaskor a kezdőértékek alapján triviálisan fennáll */
    if (h & 1) e *= a;           /* e' = e * a^{h&1} */
    h >>= 1;                    /* h' = (h-(h&1))/2 */
    a *= a;                      /* a' = a*a */
    /* indukció: e'*a^{h'} = ... = e*a^h */
  }
  return e;
} /* Az invariánstól h = 0 miatt következik az utófeltétel */
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Fibonacci sorozat — Prolog stílusban

- A Fibonacci sorozat teljes Prolog megvalósítása, és az ennek megfeleltethető C kód:

```
fib(N, F) :-
    fib(N, 0, 1, F).
    % unsigned fib(unsigned N)
    % { unsigned F0 = 0, F1 = 1, F2;
    %
    % ism:
    % if (N > 0)
    %   { --N;
    %     F2 = F0+F1;
    %     F0 = F1; F1 = F2; goto ism;
    %   }
    % return F0;
    % }
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Második példa: Fibonacci sorozat tagjainak hatékony számítása

- A C függvény

```
unsigned fib(unsigned n)
{ unsigned f = 0, fnxt = 1, t;
  while (n > 0) t = fnxt, fnxt += f, f = t, --n; /* (1) */
  return f;
}
```

- Az (1) ciklusnak bemenő változó: n , f , $fnxt$, kimenő változója: f .

- A ciklusnak megfelteletett Prolog eljárás: $fib(N, F0, FNXT, F)$: az $F0$ és $FNXT$ kezdőértékű Fibonacci sorozat N -edik tagja F .

% "betű szerinti" Prolog átirás:	% Leegyszerűsített alak:
fib(N, F0, FNXT, F) :- N > 0, !,	fib(N, F0, FNXT, F) :- N > 0, !,
T = FNXT, FNXT1 is FNXT+F0,	FNXT1 is FNXT+F0,
F1 = T, N1 is N-1,	N1 is N-1,
fib(N1, F1, FNXT1, F).	fib(N1, FNXT, FNXT1, F).
fib(_, F0, _, F0).	fib(_, F0, _, F0).

Deklaratív programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

MEGOLDÁSOK GYŰJTÉSE ÉS FELSOROLÁSA

Keresési feladat Prologban — felsorolás vagy gyűjtés?

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

<pre>% páros_elemei(L, Pk): Pk az L % lista páros elemeinek listája. páros_elemei([], []). páros_elemei([X L], Pk) :- X mod 2 =\= 0, !, páros_elemei(L, Pk). páros_elemei([_ L], [P Pk]) :- páros_elemei(L, Pk).</pre>	<pre>% páros_eleme(L, P): P egy páros % eleme az L listának. páros_eleme([_ L], P) :- P mod 2 =:= 0. páros_eleme(_ L], P) :- páros_eleme(L, P).</pre>
<pre>páros_elemei(L, Pk) :- member(P, L), P mod 2 =:= 0.</pre>	<pre>páros_eleme2(L, P) :- member(P, L), P mod 2 =:= 0.</pre>

Deklaratív Programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

A gyűjtő és felsoroló sémák összehasonlítása

- A páros elemeket gyűjtő ill. felsoroló eljárások alapján adjunk meg egy általános sémát a kétféle eljárástípusra!
- Az általános esetben a keresésnek lehet egy vagy több Param paramétere. Például, kereshetjük az Param-mal osztható elemeket.
- A közös építőelem: következő(V0, Param, E, V1): A V0 kifejezéssel jellemzett keresési térben az első megoldás E, és a fennmaradó keresési tér V1, a Param paraméter-érték mellett.

A gyűjtő séma:	A felsoroló séma:
<pre>% A V0 keresési térben a Param % paraméterű megoldások listája L. megoldások(V0, Param, L) :- következő(V0, Param, E, V1), !, L = [E L1], megoldások(V1, Param, L1). megoldások(_, _, []).</pre>	<pre>% A V0 keresési térben E egy % Param paraméterű megoldás. megoldás(V0, Param, E) :- következő(V0, Param, E0, V1), (E = E0 ; megoldás(V1, Param, E)).</pre>

Deklaratív Programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Mi a közös a felsoroló és gyűjtő megoldásokban?

- Keresünk meg a közös részt a páros_eleme(I) eljárásokban!
- Mindkettőben át kell lépni a páratlan elemeket:


```
% köv_páros(L0, P, L) :- Az L0 első páros eleme P, a maradék L.
köv_páros([X|L0], P, L) :-
    X mod 2 =\= 0, !, köv_páros(L0, P, L).
köv_páros([_|L], P, L).
```

- A köv_páros eljárásra épülő páros_eleme(I) eljárások:

<pre>% páros_elemei(L, Pk): Pk az L % lista páros elemeinek listája. páros_elemei(L0, Pk) :- köv_páros(L0, P, L1), !, Pk = [P Pk1], páros_elemei(L1, Pk1). páros_elemei(_, []).</pre>	<pre>% páros_eleme(L, P): P egy páros % eleme az L listának. páros_eleme(L0, P) :- köv_páros(L0, P0, L1), (P = P0 ; páros_eleme(L1, P)).</pre>
---	--

Deklaratív Programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Egy összetettebb példa: fennsíkok felsorolása

- Egy listában fennsíknak nevezünk:
 - egy csupa azonos elemből álló, legalább kételemeű, folytonos részlistát;
 - amely az ilyenek között maximális (egyik irányba sem kiterjeszthető).
- A feladat: felsorolandók egy lista fennsíkjai és kezdőpozícióijuk.

- Egy gyorsprogramozási módszerrel készült megoldás:


```
% Az L listában az F pozíción egy H hosszú fennsík van.
fennsíkok(L, F, H) :-
    Teste = [E,F|_],
    append(Eleje, Teste, L),
    \+ last(Eleje, E),
    length(Eleje, F0), F is F0+1,
    kezdethossz(Teste, H).
% kezdethossz/2 definícióját
% lásd korábban
```

<pre>fennsíkok(L, F, H) :- Teste = [E,F _], append(Eleje, Teste, L), \+ last(Eleje, E), length(Eleje, F0), F is F0+1, (append(EK, Farok, Teste), \+ Farok = [E _] -> length(EK, H)).</pre>	
---	--

Deklaratív Programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Fennsíkek felsorolása — 2., hatékony megoldás

- Használjuk a megoldás-felsoroló sémát: megoldás(*V0*, *Param*, *E*)!
 - *V0*: »L, P«, a bejárandó lista és első elemének pozíciója;
 - *Param*: üres;
 - *E*: »F, H«, a megoldás-fennsíkek kezdőpozíciója és hossza.
- Az *L* listában az *F* pozíción egy *H* hosszú fennsíks van.


```
fennsíks(L, F, H) :-
    fennsíks(L, 1, F, H).
```
- A *P0*-tól számozott *L0* listában az *F* pozíción egy *H* hosszú fennsíks van.


```
fennsíks(L0, P0, F, H) :-
    % az első fennsíks jellemzői F0 és H0, az utána levő maradék L1:
    első_fennsíks(L0, P0, F0, H0, L1),
    ( F = F0, H = H0
    ; % L1 kezdőpoz.:ja, P1 = előző megoldás kezdőpoz.:ja+hossza:
      P1 is F0+H0, fennsíks(L1, P1, F, H)
    ).
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

Fennsíkek felsorolása — 2., hatékony megoldás (folyt.)

```
% első_fennsíks(+L0, +P0, -F, -H, -L): A P0-tól számozott L0 listában az
% első fennsíks az F. pozíción van és hossza H, a fennsíks után fennmaradó
% rész pedig az L lista.
első_fennsíks([E, E|L1], P0, F, H, L) :-
    !, F = P0, azonosaks(L1, E, 2, H, L).
első_fennsíks([_|L1], P0, F, H, L) :-
    P1 is P0+1,
    első_fennsíks(L1, P1, F, H, L).
```

```
% azonosaks(+L0, +E, +H0, -H, -L): Az L0 lista elejéről a maximális számú
% E-vel azonos elemet leahagyva marad L, a leahagyott elemek száma H-H0.
azonosaks([X|L0], E, H0, H, L) :-
    E = X, !,
    H1 is H0+1,
    azonosaks(L0, E, H1, H, L).
azonosaks(L, _, H, H, L).
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

17/18. előadás (logikai programozás)

MEGOLDÁSGYŰJTŐ BEÉPÍTETT ELJÁRÁSOK

Gyűjtés és felsorolás kapcsolata

- Korábban láttuk, hogyan lehet egy keresési feladat gyűjtő és felsoroló eljárásait egy közös magból előállítani.
- Most vizsgáljuk meg, hogyan lehet egy felsoroló eljárást visszavezetni a gyűjtőre, és fordítva:
 - felsorolás gyűjtésből: a `member/2` könyvtári eljárás segítségével, pl.


```
páros_eleme(L, P) :-
    páros_elemei(L, Pk), member(P, Pk).
```

 Természetesen ez így nem hatékony!
 - gyűjtés felsorolásból: a megoldásgyűjtő beépített eljárások segítségével, pl.


```
páros_elemei(L, Pk) :-
    % A páros_eleme(L, P) cél összes P megoldásának listája Pk:
    findall(P, páros_eleme(L, P), Pk).
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

18. előadás (logikai programozás)

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

- Az eljárás végrehajtása:
 - a Cél kifejezést eljárásnévként értelmezi, meghívja (A :+ annotáció meta- (azaz eljárás) argumentumot jelen);
 - minden egyes megoldásához előállítja Gyűjtő egy *másolatát* (a megoldásból változók, ha vannak, szisztematikusan újakkal helyettesítődnek);
 - Az összes Gyűjtő értéket egy listába összegyűjti, és ezt egyesíti Lista-val.
- Példák az eljárás használatára:


```
| ?- findall(X, (member(X, [1,7,8,3,2,4]), X>3), L).
    => L = [7,8,4] ? ; no
| ?- findall(X-Y, (between(1, 3, X), between(1, X, Y)), L).
    => L = [1-1,2-1,2-2,3-1,3-2,3-3] ? ; no
```

- Az eljárás jelentése: Lista = { Gyűjtő másolat | $\exists X \dots Z$ Cél igaz } ahol X, ..., Z a findall hívásban levő szabad változók (azaz olyan, a hívás pillanatában behelyettesíthető változók, amelyek a Cél-ban előfordulnak de a Gyűjtő-ben nem).

A bagof megoldásgyűjtő eljárás (folyt.)

- Explicit kvantorok
 - bagof(Gyűjtő, V1 ~ ... ~ Vn ~ Cél, Lista) alakú hívása a V1, ..., Vn változókat egzisztenciálisan kötöttnek tekinti, nem sorolja fel.
 - jelentése: Lista = { Gyűjtő | $\exists V_1, \dots, V_n$ Cél igaz } $\neq \square$.


```
| ?- gráf(_G), bagof(B, A_member(A-B, _G), VegP).
    => VegP = [b,c,c,d,d] ? ; no
```
- Egymásba ágyazott gyűjtések
 - szabad változók esetén a bagof nemdet. lehet, így skatulyázható:


```
% A G irányított gráf fokszámlistája FL:
% FL = { A-N | N = { { V | A-V ∈ G } } }
fokszámai(G, FL) :-
  bagof(A-N, Vk~(bagof(V, member(A-V, G), Vks),
    length(Vks, N)
  ), FL).
| ?- gráf(_G), fokszámai(_G, FL). => FL = [a-2,b-2,c-1] ? ; no
```

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

- Az eljárás végrehajtása:
 - a Cél kifejezést eljárásnévként értelmezi, meghívja;
 - összegyűjti a megoldásait (a Gyűjtő-t és a szabad változókat);
 - a szabad változók összes behelyettesítését *felsorolja* és mindegyikhez a Lista-ban megadja az összes hozzá tartozó Gyűjtő értéket.
- Példák az eljárás használatára:


```
gráf([a-b,a-c,b-c,c-d,b-d]).
| ?- gráf(_G), findall(B, member(A-B, _G), VegP).
    => VegP = [b,c,c,d,d] ? ; no
| ?- gráf(_G), bagof(B, member(A-B, _G), VegP).
    => A = a, VegP = [b,c] ? ;
    A = b, VegP = [c,d] ? ;
    A = c, VegP = [d] ? ; no
```
- Az eljárás jelentése: Lista = { Gyűjtő | Cél igaz }, Lista $\neq \square$.

A bagof megoldásgyűjtő eljárás (folyt.)

- Fokszámlista hatékonyabb előállítása
 - a vezérlési szerkezeteket célszerű elkerülni a meta-argumentumokban
 - segédeljárás bevezetésével a kvantor is szükségtelemmé válik:


```
% Az A pont foka a G irányított gráfban M>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, LO), sort(LO, Lista),
- itt sort/2 egy univerzális rendező eljárás (lásd később), amely
- az eredménylistát rendezzi (az ismétlődések kiszűrésevel).

- Példa a setof/3 eljárás használatára:

```
gráf([a-b,a-c,b-c,c-d,b-d]).
```

```
% Gráf egy pontja P.
```

```
pontja(P, Gráf) :- member(A-B, Gráf), ( P = A ; P = B).
```

```
% A G gráf pontjainak listája Pk.
```

```
gráf_pontjai(G, Pk) :- setof(P, pontja(P, G), Pk).
```

```
| ?- gráf(_G), gráf_pontjai(_G, Pk). => Pk = [a,b,c,d] ? ; no
```

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

- A Prolog kifejezések pillanatnyi behelyettesítettségű állapotát vizsgáló eljárások (értelmszerűen sorrendfüggők):

- kifejezések osztályozása (1)

```
| ?- var(X) /* X változó? */ , X = 1. => X = 1
```

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

- kifejezések rendezése (4)

```
| ?- X @< 3 /* X megelőzői 3-tól? */ , X = 4. => X = 4
```

```
% a változók megelőzőik a nem változó kifejezéseket
```

```
| ?- X = 4, X @< 3. => no
```

- Prolog kifejezéseket szétszedő vagy összerakó eljárások:

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

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

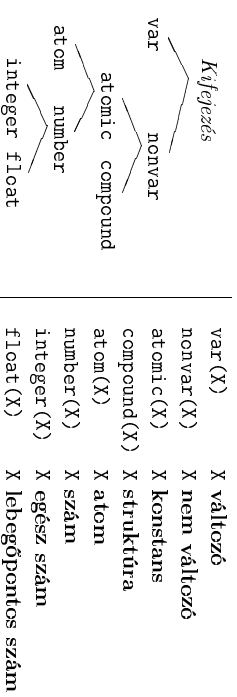
- atomok és számok \iff karaktereik (3)

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

META-LOGIKAI ELJÁRÁSOK

Kifejezések osztályozása

- Kifejezés-osztályok fastruktúrája — osztályozó beépített eljárások



- SICStus-specifikus osztályozó eljárások:

- simple(X): X nem összetett (konstans vagy változó);
- ground(X): X tömör, azaz nem tartalmaz behelyettesítetlen változót.

- Az osztályozó eljárások használata — példák

- var, nonvar — többirányú eljárásokban a különböző irányok elágaztatása
- number, atom, ... — nem-megkülönböztetett ünök feldolgozása

Osztályozó eljárások: elágaztatás behelyettesítettég alapján

- Példa: a `length/2` befejezett eljárás megvalósítása (SICStus kód!)

```
% length(?L, ?N): Az L lista N hosszú.
length(L, N) :- var(N), !, length(L, 0, N).
length(L, N) :-
    % length(?L, +I0, -I):
    % Az L lista I-I0 hosszú.
    length([], I, I).
    length([_:_, I0, I] :-
        I1 is I0+1,
        length(L, I1, I).
| ?- length([1,2], Len).      (length/3) => Len = 2 ? ; no
| ?- length([1,2], 3).      (length/3) => no
| ?- length(L, 3).          (length/3) => L = [_A,_B,_C] ? ; no
| ?- length(L, Len).        (length/3) => L = [], Len = 0 ? ;
                               L = [_A], Len = 1 ? ; L = [_A,_B], Len = 2 ?
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

19. előadás (logikai programozás)

Struktúrák szétszedése és összerakása: az *univ* eljárás

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

• **Példák**

```
| ?- el(a,b,10) =.. L.      => 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]]
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

19. előadás (logikai programozás)

Osztályozó eljárások: nem-megkülönböztetett írók kezelése

- Példa: egy formula természetes módon ábrázolható Prologban, pl. $x*y+y+1$. A formula típusának leírásához nem-megkülönböztetett írókat kell használni:
 - `% :- type form == atom \\/ int \\/ { form+form } \\/ { form-form } ...`

- Példa formulakezelésre: szimbolikus derivált előállítás

```
% deriv(+Kif, +X, ?D): Kif-nek az X atom szerinti deriváltja D.
deriv(X, X, D) :- !, D = 1.
deriv(C, _X, D) :- atomc(C), !, D = 0.
deriv(U+V, X, DU+DV) :-
    deriv(U+V, X, DU+DV) :-
    deriv(U-V, X, DU-DV) :-
    deriv(U*X, X, DU*V + U*Dv) :-
    deriv(U/V, X, (DU*V - U*Dv)/(V*V)) :-
    deriv(U, X, DU), deriv(V, X, DV).
| ?- deriv(x**y+1, x, DX), deriv(x**y+1, y, DY).
    => DX = 1*y**x+0+0, DY = 0*y**x**1+0 ? ; no
| ?- deriv((x+y)*(2+x), x, D).
    => D = (1+0)*(2+x)+(x+y)*(0+1) ? ; no
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

19. előadás (logikai programozás)

Struktúrák szétszedése és összerakása: a *functor* eljárás

- *functor/3*: kifejezés funktorának; adott funktorú kifejezésnek az előállítása
 - Hívási minták: `functor(-Kif, +Név, +Arghszám)`
`functor(+Kif, ?Név, ?Arghszám)`
 - Jelentése: igaz, ha Kif egy *Név*/Arghszám 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]
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

19. előadás (logikai programozás)

Struktúrák szétszedése és összerakása: az arg eljárás

- `arg/3`: kifejezés adott sorszámú argumentuma.
 - Hírási minta: `arg(+Sorszám, +StrKif, ?Arg)`
 - Jelentése: A `StrKif` struktúra `Sorszám`-adik argumentuma `Arg`.
 - Végrehajtása: `Arg`-ot az adott sorszámú argumentummal *egyesíti* (kétirányúság!).
- Példák:


```
| ?- arg(3, el(a, b, 23), Arg).      => Arg = 23
| ?- K=el( _, _), arg(1, K, a),
   arg(2, K, b), arg(3, K, 23).    => K = el(a,b,23)
| ?- arg(1, [1,2,3], A).          => A = 1
| ?- arg(2, [1,2,3], B).          => B = [2,3]
```
- Az *univ* visszavezethető a functor és `arg` eljárásokra (és viszont), például:


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

Deklaratív programozás, BMEF, 2001 tavaszi félév

18. előadás (logikai programozás)

Az univ alkalmazása: ismétlődő sémák összevonása (folyt.)

- Kifejezés-egyszerűsítés, 2. megoldás, *univ* segítségével


```
egysz(X, EX) :- atomic(X), !, EX = X.
egysz(Kif, EKif) :-
    Kif =.. [Muv,U,V],      % Kif = Muv(U,V)
    egysz(U, EU), egysz(V, EV),
    EUV =.. [Muv,EU,EV],  % EUV = Muv(EU,EV)
    kiszamol(EUV, EU, EV, EKif).
```
- Kifejezés-egyszerűsítés, általánosítás tetszőleges tömör kifejezésekre:


```
egysz1(Kif, EKif) :-
    Kif =.. [M|Arg[]], egysz1_lista(ArgL, EArgL), EKif0 =.. [M|EArgL],
    % catch(:+Cél, ?Kiv, :+KÖel, :+KÖel): ha Cél kivétel dob, KÖel-t futtatja.
    catch(EKif is EKif0, _, EKif = EKif0).
```

```
egysz1_lista([], []).
egysz1_lista([K|Kk], [EK|Ek]) :- egysz1(K, E), egysz1_lista(Kk, Ek).
| ?- egysz1(f(1+2+a, exp(3,2), a+1+2), E). => E = f(3+a,9.0,a+1+2)
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

18. előadás (logikai programozás)

Az univ alkalmazása: ismétlődő sémák összevonása

- A feladat: egy szimbolikus aritmetikai kifejezésben a kiértékelhető (infix) részkifejezések helyettesítése az értékükkel.
 - 1. megoldás, *univ* nélkül:


```
% Az X szimbolikus kifejezés egyszerűsítése EX.
egysz0(X, EX) :- atomic(X), !, EX = X.
egysz0(U+V, EKif) :-
    egysz0(U, EU), egysz0(V, EV), kiszamol(EU+EV, EU, EV, EKif).
egysz0(U*KV, EKif) :-
    egysz0(U, EU), egysz0(V, EV), kiszamol(EU*EV, EU, EV, EKif).
%...
% EU és EV részekből képzett EUV egyszerűsítése EKif.
kiszamol(EUV, EU, EV, EKif) :-
    number(EUV), number(EV), !, EKif is EUV.
kiszamol(EUV, _, _, EUV).
```
 - | ?- `deriv((x+y)*(2+x), x, D)`, `egysz0(D, ED)`.


```
=> D = (1+0)*(2+x)+(x+y)*(0+1), ED = 1*(2+x)+(x+y)*1 ? ; no
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

18. előadás (logikai programozás)

Univ alkalmazása általános kifejezés-bejárára: kintatás

- A feladat: egy tetszőleges kifejezés kintatása úgy, hogy
 - a kétargumentumú operátorok zárójellezett infix formában,
 - minden más alap-struktúra alakban jelenjék meg.
- Kifejezés-egyszerűsítés, 2. megoldás, *univ* segítségével


```
Kif =.. [Func, A1|ArgL],
    (current_op(_, Kind, Func), (Kind=xfy;Kind=yfx;
    ArgL = [A2] -> % kétargumentumú operátor
    write('(', ki(A1), format('~w', [Fun]), ki(A2), write(')'))
    ; write(Func), write('(', ki(A1), arglistaki(ArgL), write(')'))
    ),
    ki(Kif) :- write(Kif).
```

```
% Az [A1,...,An] listát ",A1,...,An" alakban kintatja.
arglistaki([]).
arglistaki([A|AL]) :- write(' ', ki(A), arglistaki(AL)).
| ?- ki(f(+a, X*c*X, e)). => f(+a), ((_117 * c) * _117), e)
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

18. előadás (logikai programozás)

Unio alkalmazása általános kifejezés-bejárásra: változómentesítés

- A SICStus Prologban beépített `numbervars(?Kif, +NO, ?N)` eljárás hatása:
 - A tetszőleges `Kif` minden változóját `'$VAR'(I)` alakú kifejezéssel helyettesíti, $I = NO, \dots, N-1$ (azaz `Kif`-ben $N-NO$ különböző változó van).
 - `A : '$VAR'(0), '$VAR'(1), \dots` kifejezések `write`-tal való kiíráskor változónévként `(A, B, \dots)` jelennek meg.
- Ezek speciális opciókkal `write_term`-mel „eredetiben” is megjelentethetők:


```
| ?- _Kif = [F(X),g(_),_X], numbervars(_Kif, 0, N), write(_Kif), nl,
      write_term(_Kif, [quoted(true),numbervars(false)]).
      => [F(A),g(B),A]
      [F('$VAR'(0)),g('$VAR'(1)),$VAR'(0)]
      N = 2
```
- A feladat: elkészítendő egy `numbervars1/3` eljárás, amely `'$VAR'` helyett `'$nyvar'` funktort használ.

Deklaratív Programozás, BMEF, 2001 tavaszi félév

18. előadás (logikai programozás)

Unio alkalmazása: részkifejezések keresése

- A feladat: egy tetszőleges kifejezéshez soroljuk fel a benne levő számokat, és minden számon eseten adjuk meg annak a *kiválasztóját*!
- Egy részkifejezés kiválasztója egy olyan lista, amely megadja, mely argumentumpozíciók mentén juthatunk el hozzá.
- Az $[i_1, i_2, \dots, i_k]$ lista egy `Kif`-ből az i_1 -edik argumentum i_2 -edik argumentumának, \dots i_k -edik argumentumát választja ki.
- Pl. `a*b+f(1,2,3)/c`-ben `b` kiválasztója `[1,2]`, `3` kiválasztója `[2,1,3]`.


```
% kif_szám(?kif, ?M, ?Kiv): Kif Kív kiválasztójú része az M szám.
kif_szám(X, N, Kiv) :- number(X), !, N = X, Kiv = [].
kif_szám(X, N, [_|Kiv]) :-
    compound(X), % a változó kizárása miatt fontos!
    functor(X, F, N), between(1, N, I), arg(I, X, X1),
    kif_szám(X1, N, Kiv).
| ?- kif_szám(f(1,[b,2]), N, K). => K=[1],N=1? ; K=[2,2,1],N=2? ; no
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

18. előadás (logikai programozás)

Általános kifejezés-bejárás unio-val : saját változómentesítés

- ```
% A Term kifejezésben levő változókat '$nyvar(I)' stb.
% struktúrákkal helyettesíti be, I = NO, ... N-1.
numbervars1(Term, NO, N) :- var(Term), !,
 Term = '$nyvar'(NO), N is NO+1.
numbervars1(Term, NO, N) :-
 Term =.. [_|Args], numbervars1_list(Args, NO, N).

% numbervars1_list(L, NO, N): Az L listában levő változókat
% '$nyvar(I)' stb. struktúrákkal helyettesíti be, I = NO, ... N-1.
numbervars1_list([], N, N).
numbervars1_list([_|As], NO, N) :-
 numbervars1(A, NO, N1), numbervars1_list(As, N1, N).

| ?- Kif = [F(X),g(_),_X], numbervars1(Kif, 0, N).
 => N = 2,
 Kif = [F('$nyvar'(0)),g('$nyvar'(1)),$nyvar'(0)]
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

18. előadás (logikai programozás)

## Atomok szétzsedése és összerakása

- `atom_codes/2`: atom és karakterkód-lista közötti átalakítás
  - Hívási minták: `atom_codes(+Atom, ?Kódlista)`  
`atom_codes(-Atom, +Kódlista)`
  - Jelentése: Igaz, ha `Atom` karakterkódjainak a listája `Kódlista`.
  - Végrehatás:
    - Ha `Atom` bemenő, és a  $c_1c_2\dots c_n$  karakterekből áll, akkor `Kódlista = [k1, k2, ..., kn]`, ahol  $k_i$  a  $c_i$  karakter kódja.
    - Ha `Atom` kimenő, akkor a `Kódlista` karakterkód-listából összerak egy atomot, és azt egyesíti `Atom`-mal.
- Példák:
 

```
| ?- atom_codes(ab, Cs). => 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
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

18. előadás (logikai programozás)



## Atomok szétszedése és összerakása — alkalmazási példák

- Keresés atomokban

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

```
dadogó_rész(Atom, Rész) :-
```

```
 atom_codes(Atom, Cs), dadogó(Cs, Ds), atom_codes(Rész, Ds).
```

```
% L-ben a D nem üres részlista kétszer ismétlődik (lásd korábban).
```

```
dadogó(L, D) :- D = [_|_],
```

```
 append(_, Farok, L), append(D, Vég, Farok), append(D, -, Vég).
```

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

- Atomok összeffűzése

```
% atom_concat(+A, +B, ?C): A és B atomok összeffűzése C.
```

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

```
atom_concat(A, B, C) :- atom_codes(A, Ak), atom_codes(B, Bk),
```

```
 append(Ak, Bk, Ck), atom_codes(C, Ck).
```

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

Deklaratív programozás, BNMF, 2001 tavaszi félév

19. előadás (logikai programozás)

Meta-logikai eljárások

355

## Kifejezések rendezése: szabványos sorrend

Legyen  $X$  és  $Y$  két tetszőleges Prolog kifejezés, ha  $X$  megelőzi  $Y$ -t, azt írjuk, hogy  $X \prec Y$ .

1. Ha  $X$  és  $Y$  azonos, akkor sem  $X \prec Y$  sem  $Y \prec X$  nem igaz és fordítva.
2. Ha  $X$  és  $Y$  különböző kifejezésosztályba tartozik, akkor az osztály dönt: *ülkőző*  $\prec$  *lebegőpontos szám*  $\prec$  *egész szám*  $\prec$  *nőv*  $\prec$  *struktúra*.
3. Ha  $X$  és  $Y$  változó, akkor az eredmény rendszerfüggő.
4. Ha  $X$  és  $Y$  lebegőpontos vagy egész szám, akkor  $X \prec Y \Leftrightarrow X < Y$ .
5. Ha  $X$  és  $Y$  név, akkor sorrendjük megegyezik az abc sorrenddel.
6. Ha  $X$  és  $Y$  struktúrák:

6.1. Ha  $X$  és  $Y$  ariása különböző,  $X \prec Y \Leftrightarrow X$  ariása kisebb mint  $Y$  ariása.

6.2. Egyébként, ha a rekordok neve különböző,  $X \prec Y \Leftrightarrow X$  neve  $\prec Y$  neve.

6.3. Egyébként balról az első nem azonos argumentum dönt.

Végtelen (ciklikus) kifejezésekre a fenti rendezés nem érvényes.

Deklaratív programozás, BNMF, 2001 tavaszi félév

19. előadás (logikai programozás)

## Számok szétszedése és összerakása

- number\_codes/2: szám és karakterkód-lista közötti átalakítás

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

```
number_codes(-Szám, +Kódlista)
```

- Jelentése: Igaz, ha Szám tízes számrendszerbeli alakja a Kódlista karakterkód-listának felel meg.

- Végrehajtása:

- Ha Szám bemenő, és tízes számrendszerben a  $c_1c_2\dots c_n$  karakterekből áll, akkor Kódlista =  $[k_1, k_2, \dots, k_n]$ , ahol  $k_i$  a  $c_i$  karakter kódja.

- Ha Szám kimenő, akkor a Kódlista karakterkód-listából összerak egy számot, és azt egyesíti Szám-mal.

- Példák:

```
| ?- 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)
```

Deklaratív programozás, BNMF, 2001 tavaszi félév

19. előadás (logikai programozás)

Meta-logikai eljárások

356

## Kifejezések összehasonlítása — beépített eljárások

- Két tetszőleges kifejezés összehasonlítását végző eljárások:

| Hívás                         | igaz, ha                                                                                       |
|-------------------------------|------------------------------------------------------------------------------------------------|
| <code>Kif1 == Kif2</code>     | <code>Kif1</code> $\neq$ <code>Kif2</code> $\wedge$ <code>Kif2</code> $\neq$ <code>Kif1</code> |
| <code>Kif1 \== Kif2</code>    | <code>Kif1</code> $\prec$ <code>Kif2</code> $\vee$ <code>Kif2</code> $\prec$ <code>Kif1</code> |
| <code>Kif1 @&lt; Kif2</code>  | <code>Kif1</code> $\prec$ <code>Kif2</code>                                                    |
| <code>Kif1 @=&lt; Kif2</code> | <code>Kif2</code> $\neq$ <code>Kif1</code>                                                     |
| <code>Kif1 @&gt; Kif2</code>  | <code>Kif2</code> $\prec$ <code>Kif1</code>                                                    |
| <code>Kif1 @&gt;= Kif2</code> | <code>Kif1</code> $\neq$ <code>Kif2</code>                                                     |

- Az összehasonlító eljárások logikailag nem tiszták:

```
| ?- X @< 3, X = 4. => X = 4
| ?- X = 4, X @< 3. => no
```

- Az összehasonlítás mindig a belső ábrázolás szerint történik:

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

Deklaratív programozás, BNMF, 2001 tavaszi félév

19. előadás (logikai programozás)

## A meta-logikai eljárások egy komplex alkalmazása: $\prec$ megvalósítása

```

% I1 megelőzi I2-t a szabványos sorrendben (lényegében I1 @< I2)
precedes(T1, T2) :- \+ \+ (numbervars1(T1-T2, 0, _), prec(T1, T2)).
% I1 megelőzi I2-t, a változók ndr '$gyvar'(n) konstansokra cseréltek.
prec(T1, T2) :- class(T1, C1), class(T2, C2),
 (C1 == C2 ->
 (C1 == 1 -> T1 < T2 % 4. szabály (lebegőpontos szám)
 ; C1 == 2 -> T1 < T2 % 4. szabály (egész szám)
 ; struct_prec(T1, T2) % 3., 5. és 6. szabály
)
 ; C1 < C2 % (változó, név, struktúra)
)
).
% class(+T, -C): A T kifejezés a C-edik kifejezésosztályba tartozik.
class(T, C) :- (/*vált*/ T='$_gyvar'(_) -> C=0
; /*szám*/ float(T) -> C=1 ; integer(T) -> C=2
; /*atom*/ atom(T) -> C=3 ; /*struktúra*/ C=4
).

```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

18. előadás (logikai programozás)

## A $\prec$ reláció megvalósítása (folyt.)

```

% S1 megelőzi S2-t (struktúra-kifejezésekre és atomokra).
struct_prec(S1, S2) :- functor(S1, F1, M1), functor(S2, F2, M2),
 (M1 < M2 -> true
 ; M1 = M2, (F1 = F2 -> args_prec(1, M1, S1, S2)
 ; atom_prec(F1, F2)
)
).
% Az S1 struktúra-kifejezés N0, ..., N sorozatú argumentumai
% lenikogratifikusan megelőzik S2 azonos sorozatú argumentumait.
args_prec(N0, N, S1, S2) :- NO =< N, arg(NO, S1, A1), arg(NO, S2, A2),
 (A1 = A2 -> M1 is NO+1, args_prec(M1, N, S1, S2)
 ; prec(A1, A2)
).

```

```

% A1 atom megelőzi A2 atomot (előfeltétel: A1 \= A2).
atom_prec(A1, A2) :-
 atom_codes(A1, C1), atom_codes(A2, C2), struct_prec(C1, C2).

```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

18. előadás (logikai programozás)

## A Prolog egyszerűség-szerű beépített eljárásai

- $U = V$ :  $U$  egyesítendő  $V$ -vel.  
Soha sem jelez hibát.  
| ?- X = 1+2.       $\implies$  X = 1+2  
| ?- 3 = 1+2.       $\implies$  no
- $U = V$ :  $U$  azonos  $V$ -vel.  
Soha sem jelez hibát és soha sem helyettesít be.  
| ?- X == 1+2.       $\implies$  no  
| ?- 3 == 1+2.       $\implies$  no  
| ?- +(1,2)==1+2  $\implies$  yes
- $U ::= V$ : Az  $U$  és  $V$  aritmetikai kifejezések értéke megegyezik.  
Hibát jelez, ha  $U$  vagy  $V$  nem (tömör) aritmetikai kifejezés.  
| ?- X ::= 1+2.       $\implies$  hiba  
| ?- 1+2 ::= X.       $\implies$  hiba  
| ?- 2+1 ::= 1+2.  $\implies$  yes  
| ?- 2.0 ::= 1+1.  $\implies$  yes  
| ?- 2.0 is 1+1.       $\implies$  no
- $U$  is  $V$ :  $U$  egyesítendő a  $V$  aritmetikai kifejezés értékével.  
Hiba, ha  $V$  nem (tömör) aritmetikai kifejezés.  
| ?- X is 1+2.       $\implies$  X = 3  
| ?- 1+2 is X.       $\implies$  hiba  
| ?- 3 is 1+2.       $\implies$  yes  
| ?- 1+2 is 1+2.       $\implies$  no
- $(U =.. V$ :  $U$ , „szétszedettje” a  $V$  lista)  
| ?- 1+2 =.. X.       $\implies$  X = [+, 1, 2]  
| ?- X =.. [f, 1].  $\implies$  X = f(1)

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## EGYENLŐSÉGFAJTÁK — ÖSSZEFOGLALÁS

## A Prolog nem-egyenlőség jellegű beépített eljárásai

- A nem-egyenlőség jellegű eljárások sohasem helyettesítenek be változóit!
  - $U \setminus = V$ :  $U$  nem egyesíthető  $V$ -vel.
 

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

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

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

 Soha sem jelez hibát.
  - $U \setminus = V$ : Az  $U$  és  $V$  aritmetikai kifejezések értéke különbözők. Hibát jelez, ha  $U$  vagy  $V$  nem (főmór) aritmetikai kifejezés.
 

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

Deklaratív Programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

## A Prolog (nem-)egyenlőség jellegű beépített eljárásai — példák

|       | <i>Egységítés</i> | <i>Azonoság</i>   | <i>Aritmetika</i> |                   |          |                   |                   |
|-------|-------------------|-------------------|-------------------|-------------------|----------|-------------------|-------------------|
|       | $U = V$           | $U \setminus = V$ | $U == V$          | $U \setminus = V$ | $U := V$ | $U \setminus = V$ | $U \text{ is } V$ |
| $U$   | $V$               | no                | yes               | no                | no       | yes               | no                |
| 1     | 2                 | no                | yes               | no                | no       | yes               | no                |
| a     | b                 | no                | yes               | no                | error    | error             | error             |
| $1+2$ | $+(1,2)$          | yes               | no                | yes               | yes      | no                | no                |
| $1+2$ | $2+1$             | no                | yes               | no                | yes      | no                | no                |
| $1+2$ | 3                 | no                | yes               | no                | yes      | no                | no                |
| 3     | $1+2$             | no                | yes               | no                | yes      | no                | yes               |
| X     | $1+2$             | $X=1+2$           | no                | no                | error    | error             | $X=3$             |
| X     | Y                 | $X=Y$             | no                | no                | error    | error             | error             |
| X     | X                 | yes               | no                | yes               | no       | error             | error             |

Jelmegnyarazati: yes — siker; no — meghiúsulás, error — hiba.

Deklaratív Programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

## MODULARITÁS

- A SICStus Prolog modulfogalmának jellemzői:
  - Minden modul külön állományba kell kerülnön.
  - Az állomány első programeleme egy modul-parancs kell legyen:
 

```
:- module(ModulNév, [ExpFunktor1, ExpFunktor2, ...]).
ExpFunktor = az exportálandó eljárás funkтора (név/argumentumszám)
```
  - Pl. :- module(plato, [fennsik/3]). % plato állomány első sora
  - Modul-betöltésre szolgáló beépített eljárások:
    - use\_module(ÁllományNév)
    - use\_module(ÁllományNév, [ImpFunktor1, ImpFunktor2, ...])
    - ImpFunktor — az importálandó eljárás funkтора
    - ÁllományNév lehet atom, vagy pl. library(KönyvtárNév):
 

```
:- use_module(plato).
% a fenti modul betöltése
```
  - A modulfogalom nem szigorú: platok:első\_fennsik(...) meghívható!
  - Modulkvalifikált hívási forma: ModulNév:EljárásNév(Argumentumok ...).

Deklaratív Programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

## Meta-eljárások modularizált programban

- Eljárások átadása paraméterként modulközi hívásban gondot okozhat:
  - `m1.pl` állománya:
 

```
:- module(m1, [kétyszer/1]).
kétyszer(X) :- X, X.
p :- write(bu).
```
  - `m2.pl` állománya:
 

```
:- module(m2, [q/0, r/0]).
:- use_module(m1).
q :- kétyszer(p). r :- kétyszer(m2:p).
p :- write(ba).
```
  - Futtatás:
 

```
| ?- [m1, m2].
| ?- q. => bubu
| ?- r. => baba
```
  - Automatikus modul-kvalifikáció meta-predikátum deklarációival:
 

**Ha `m1.pl`-be beszurjuk: :- meta\_predicate kétyszer(:), akkor**

```
| ?- q. => baba!
```

Deklaratív Programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

## Meta-predikátum deklaráció, modulnév-kiterjesztés

- Meta-predikátum deklaráció
  - Formája:
 

```
:- meta_predicate (eljárásnév)(<módspec1>, ..., <módspecn>), ...
```

    - `<módspeci>` lehet `'::'`, `'+'`, `'-'`, vagy `'?'`.
    - A `'::'` mód azt jelzi, hogy az adott argumentumot betöltéskor ún. modulnév-kiterjesztésnek kell alávetni.
  - Egy *klf* kifejezés modulnév-kiterjesztése:
    - ha *M:X* alakú (vagy egy olyan változó, amely az adott eljárás fejében meta-argumentum pozíción szerepelt) akkor változatlanul hagyjuk;
    - egyébként helyettesítjük *CurMod:klf*-fel (CurMod a kurrens modul).
  - Példa folyt. (az `m1`-beli kétyszer meta-predikátumnak deklarált)
 

```
:- module(m2, [négyezer/1, q/0]). :- use_module(m1).
q :- kétszer(p). => q :- kétszer(m2:p).
:- meta_predicate négyezer(:).
négyezer(X) :- kétszer(X), kétszer(X). => változatlan
```

Deklaratív Programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

## MAGASABBRENDŰ ELJÁRÁSOK

### Magasabbrendű eljárások — listakezelés

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

```
% Az L egész-lista páros elemeinek listája Pk.
páros_elemei(L, Pk) :-
 findall(X, (member(X, L), X mod 2 == 0), Pk).

% Az L számlista elemei négyzeteknek listája Nk.
négyzetei(L, Nk) :-
 findall(Y, (member(X, L), Y is X*X), Nk).

| ?- páros_elemei([1, 2, 3, 4], Pk). => Pk = [2, 4]
| ?- négyzetei([1, 2, 3, 4], Nk). => Nk = [1, 4, 9, 16]
```

Deklaratív Programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

## Listakezelő meta-eljárások megoldásgyűjtő eszközökkel

- Lista szűrése (vö. a filter SML függvényei!)
  - % Az L lista X elemének Pred szerinti szűrése FL.*
  - `:- meta_predicate filter(+, ?, :, -).`
  - `filter(L, X, Pred, FL) :-`
  - `findall(X, (member(X, L), call(Pred)), FL).`
  - `| ?- filter([1,2,3,4], X, X mod 2 =:= 0, Pk). => Pk = [2,4]`
- Lista leképezése (vö. a map SML függvényei!)
  - % Az L lista X elemeit Pred-del Y-ba képezve kapjuk az ML listát.*
  - `:- meta_predicate map(+, ?, :, ?, -).`
  - `map(L, X, Pred, Y, ML) :-`
  - `findall(Y, (member(X, L), Pred), ML).`
  - `| ?- map([1,2,3,4], X, Y is X*X, Y, NK). => NK = [1,4,9,16]`
- A példákban a szűrést az  $\langle X, \text{Pred} \rangle$  argumentumpár, a leképezést az  $\langle X, \text{Pred}, Y \rangle$  hármas határozza meg. Ezek egy-egy-ill. kétargumentumú predikátumot adnak meg (vö. a funkcionális nyelvek  $\lambda$ -kifejezéseivel).

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## Részlegesen paraméterezett eljárások — segédesszközök

- A call/1 eljárás általánosítása: a call/2, call/3, ... eljárások.
- call(RPred, A1, A2, ...) végrehajtása: az RPred hívást kiegészíti az A1, A2, ... argumentumokkal, és meghívja.
- A call/M eljárások sok Prologban beépítettek, SICStusban definiáltak:
  - `:- meta_predicate call(:, ?), call(:, ?, ?), .....`
  - % Pred az A utolsó argumentummal meghívva igaz.*
  - `call(M:Pred, A) :-`
  - `Pred =.. FAs0, append(FAs0, [A], FAs1), Pred1 =.. FAs1,`
  - `call(M:Pred1).`
  - % Pred az A és B utolsó argumentumokkal meghívva igaz.*
  - `call(M:Pred, A, B) :-`
  - `Pred =.. FAs0, append(FAs0, [A,B], FAs2), Pred2 =.. FAs2,`
  - `call(M:Pred2).`
  - ...

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## Részlegesen paraméterezett eljárások

- A listát elemenként négyzetreemelő eljárás egy másik változata:
  - `négyzete(X, Y) :- Y is X*X.`
  - `négyzeteik(Xk, Yk) :- map(Xk, X, négyzete(X, Y), Y, Yk).`
- A lista elemekre az  $x \rightarrow x^2 + Px + Q$  hozzárendelést alkalmazó eljárás:
  - `másodfokú_képe(P, Q, X, Y) :- Y is X*X + P*X + Q.`
  - `másodfokú_képek(P, Q, Xk, Yk) :-`
  - `map(Xk, X, másodfokú_képe(P, Q, X, Y), Y, Yk).`
- Konvenció: a meta-alkalmazásban változó paramétereket az eljárás végére tesszük — így egyszerűsíthető a meta-eljárás hívása. Példa: map/3:
  - `map(Xk, RészlPred, Yk) :-`
  - `RészlPred =.. L0, append(L0, [X, Y], L), Pred =.. L,`
  - `findall(Y, (member(X, Xk), Pred), Yk).`
  - `másodfokú_képek(P, Q, Xk, Yk) :- map(Xk, másodfokú_képe(P, Q), Yk).`

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## Részlegesen paraméterezett eljárások — rekurzív map/3

- map/3 rekurzív definíciója:
  - % map(Xs, Pred, Ys): Az Xs lista elemekre a Pred transzformációt*
  - % alkalmazva kapjuk az Ys listát.*
  - `map([X|Xs], Pred, [Y|Ys]) :-`
  - `call(Pred, X, Y), map(Xs, Pred, Ys).`
  - `map([], -, []).`
- Példák:
  - `| ?- map([1,2,3,4], négyzete, L).`
  - `=> L = [1,4,9,16]`
  - `| ?- map([1,2,3,4], másodfokú_képe(2,1), L).`
  - `=> L = [4,9,16,25]`
- A call/M-re épülő megoldás előnyei:
  - hatékonyabb és általánosabb mint a findall-ra épülő;
  - alkalmazható akkor is, ha az elemekre elvégzendő műveletek nem függetlenek, pl. földl.

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## Rekurzív meta-eljárások — foldl és foldr

```
% foldl(Xs, Pred, Y0, Y): Az Xs elemekre balról jobbra alkalmazott,
% a Pred által leírt kétargumentumú függvény Y0 kezdőértékre
% alkalmazott eredménye Y.
foldl([X|Xs], Pred, Y0, Y) :-
 call(Pred, X, Y0, Y1), foldl(Xs, Pred, Y1, Y).
foldl([], _, Y, Y).

jegyzőháza(A, J, E0, E) :- E is E0*A+J.
| ?- foldl([1,2,3], jegyzőháza(10), 0, E). => E = 123

% foldr(Xs, Pred, Y0, Y): Az Xs elemekre jobbról balra alkalmazott, a
% Pred által leírt függvény Y0 kezdőértékre alkalmazott eredménye Y.
foldr([X|Xs], Pred, Y0, Y) :-
 foldr(Xs, Pred, Y0, Y1), call(Pred, X, Y1, Y).
foldr([], _, Y, Y).

| ?- foldr([1,2,3], jegyzőháza(10), 0, E). => E = 321
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

Dinamikus adatbáziskezelés

375

## Dinamikus predikátumok

- A dinamikus predikátum jellemzői:
  - a program szövegében lehet 0 vagy több klóza;
  - futási időben hozzáadhatunk és elvehetünk klózokat belőle;
  - végrehajtása mindenképpen interpreált.
- Létrehozása
  - programszövegbeli deklarációnal:
    - :- dynamic(Eljárásnév/Argumentumszám).
 (Ha van klóza a programban, akkor az első előtt — ilyenkor kötelező);
  - futási időben, adatbáziskezelő beépített eljárással
- Adatbáziskezelő eljárások („adatbázis” = a program klózainak összessége):
  - klóz felvétele első, utolsó helyre: asserta/1, assertz/1
  - klóz törlése (illesztéssel, többszörösen sikerülhet): retract/1
  - klóz lekérdezése (illesztéssel, többszörösen sikerülhet): clause/2

Deklaratív programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## DINAMIKUS ADATBÁZISKEZELÉS

Dinamikus adatbáziskezelés

376

## Klóz felvétele: asserta/1, assertz/1

- asserta(:@Klóz)
  - A Klóz kifejezést klózként értelmezve felveszi a programba az adott predikátum *első* klózáként.
  - A ‘@’ mód jelentése: tisztán bemenő paraméter, az eljárás a paraméterbeli változókat nem helyettesíti be (a ‘+’ mód speciális esete).
  - A ‘:’ mód modul-kvalifikált paramétert jelez.
- assertz(:@Klóz)
  - A Klóz kifejezést az adott predikátum *utolsó* klózáként veszi fel
- Példa:
  - | ?- assertz((p(1,X):-q(X))), asserta(p(2,0)),
  - assertz((p(2,Z):-r(Z))), listing(p).
  - p(2, 0).

Deklaratív programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

```
p(1, A) :- q(A).
p(2, A) :- r(A).
```

## Klóz törlése: retract/1

- `retract(:@Klóz)`

- A Klóz klóz-kifejezésből megállapítja a predikátum funktorát.
- Az adott predikátum klózeit sorra megpróbálja illeszteni Klóz-zal.
- Ha az illesztés sikerült, akkor kitörli a klózt és sikeresen lefut.
- Visszalépés esetén folytatja a keresést (illeszt, töröl, sikerül, stb.)

- Példa (folytatás):

```
| ?- listing(p), retract((p(2,_) :- _)), listing(p), fail. => no
```

- A futás kimenete:

|                                                           |                                                  |
|-----------------------------------------------------------|--------------------------------------------------|
| <pre>p(2, 0). p(1, A) :-   q(A). p(2, A) :-   r(A).</pre> | <pre>p(1, A) :-   q(A). p(2, A) :-   r(A).</pre> |
|-----------------------------------------------------------|--------------------------------------------------|

## Alkalmazási példa — egyszerűsített findall

- A `findall/3` eljárás hatása megegyezik a beépített `findall/3`-al, de nem működik helyesen, ha a Cél-ban újabb `findall` hívás van.
- `:- dynamic(megoldás/1).`

```
% findall1(Minta, Cél, L): Cél összes megoldására Minta listaja L.
findall1(Minta, Cél, _MegaIDL) :-
 call(Cél), asserta(megoldás(Minta)), fail. % fordított sorrend!
findall1(_Minta, _Cél, MegaIDL) :-
 megoldás_lista([], MegaIDL).
```

```
% A megoldás/1 tényállításkönm tárolt kifejezések fordított listája L-L0.
megoldás_lista(L0, L) :-
 retract(megoldás(M)), !, megoldás_lista([M|L0], L).
megoldás_lista(L, L).
```

```
| ?- findall1(Y, (member(X, [1,2,3]), Y is X*X), ML). => ML = [1,4,9]
```

## Klóz lekérdezése: clause/2

- `clause(:@Fej, ?Törzs)`
- A Fej alapján megállapítja a predikátum funktorát.

- Az adott predikátum klózeit sorra megpróbálja illeszteni a Fej :- Törzs kifejezéssel (tényállítás esetén Törzs = true).

- Ha az illesztés sikerült, akkor sikeresen lefut.
- Visszalépés esetén folytatja a keresést (illeszt, sikerül, stb.)

- Példa:

```
:- listing(p), clause(p(2, 0), T).

p(2, 0).
p(1, A) :-
 q(A).
p(2, A) :-
 r(A).
```

|                                         |                                         |
|-----------------------------------------|-----------------------------------------|
| <pre>T = true ? ; T = r(0) ? ; no</pre> | <pre>T = true ? ; T = r(0) ? ; no</pre> |
|-----------------------------------------|-----------------------------------------|

## A clause eljárás alkalmazása: egyszerű nyomonkövető interpreter

```
% interp(G, D): A G cél futását D bekezdésű nyomonkövetéssel mutatja.
interp(true, _) :- !.
interp((G1, G2), D) :- !,
 interp(G1, D), interp(G2, D).
interp(G, D) :-
 (trace(G, D, call)
 ; trace(G, D, fail)) fail % követi a fail kaput, tovább-nyúl
),
 D2 is D+2, clause(G, B), interp(B, D2),
 (trace(G, D, exit)
 ; trace(G, D, redo)) fail % követi a redo kaput, tovább-nyúl
).

% A G cél áthaladását a Port kapun D bekezdésű nyomonkövetéssel mutatja.
trace(G, D, Port) :-
 /*D szöveget ír ki:*/ tab(D), write(Port), write(' '), write(G), nl.
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## Nyomonkövető interpreter - példafutás

```
:- dynamic app/3, app/4.

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

app(L1, L2, L3, L123) :-
 app(L1, L23, L123),
 app(L2, L3, L23).

L = [b] ?
| ?- interp(app(_, [b,c], L, [c,b,c,b]), 0).
call: app(_203, [b,c], _253, [c,b,c,b])
call: app(_203, 666, [c,b,c,b])
exit: app([], [c,b,c,b], [c,b,c,b])
call: app([b,c], _253, [c,b,c,b])
fail: app([b,c], _253, [c,b,c,b])
redo: app([], [c,b,c,b], [c,b,c,b])
call: app(_873, 666, [b,c,b])
exit: app([], [b,c,b], [b,c,b])
exit: app([c], [b,c,b], [c,b,c,b])
call: app([b,c], _253, [b,c,b])
call: app([c], _253, [b])
exit: app([], [b], [b])
exit: app([c], [b], [c,b])
exit: app([b,c], [b], [b,c,b])
exit: app([c], [b,c], [b], [c,b,c,b])
L = [b] ?
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## „HAGYOMÁNYOS” BEÉPÍTETT ELJÁRÁSOK

## Aritmetikai beépített eljárások

- X is Kif: Kif aritmetikai kifejezés kell legyen, értékét egyesíti X-szel.
- Kif1 ρ Kif2: Kif1 és Kif2 aritmetikai kifejezések kell legyenek, értékeik között elvégzi a ρ összehasonlítást (ρ lehet =, \=, <, =<, >, >=).
- Aritmetikai kifejezésekben felhasználható funktorok:

| Infix operátorok   |                         |             |                            |
|--------------------|-------------------------|-------------|----------------------------|
| + összeadás        | // egész osztás         | \           | bitenkénti és              |
| - kivonás          | ** hatványozás          | \           | bitenkénti vagy            |
| * szorzás          | mod modulus képzés      | <<          | bitenkénti balra léptetés  |
| / osztás           | rem maradék képzés      | >>          | bitenkénti jobbra léptetés |
| Prefix operátorok: | - negáció               | \           | bitenkénti negáció         |
| Függvény jelölések |                         |             |                            |
| abs/1              | exp/1                   | floor/1     | sign/1                     |
| atan/1             | float/1                 | log/1       | sin/1                      |
| ceiling/1          | float_fractional_part/1 | max/2_min/2 | sqrt/1                     |
| cos/1              | float_integer_part/1    | round/1     | truncate/1                 |

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)



## Listakezelő beépített eljárások

- Lista hossza: `length(L, ?N)`
- Jelentése: az `L` lista hossza `N`.
- `length(-L, +N)` módban adott hosszúságú, csupa különböző változóból álló listát hoz létre.
- `length(-L, -N)` módban rendre felsorolja a `0`, `1`,  $\dots$  hosszú listákat.
- Megvalósítását lásd korábban.
- Lista rendezése: `sort(@L, ?S)`
  - Jelentése: az `L` lista `@<` szerinti rendezése `S`, (`==/2` szerint azonos elemek ismétlődését kiszűrve).
- Lista kulcs szerinti rendezése: `keysort(@L, ?S)`
- Az `L` argumentum kulcs-Érték alakú kifejezések listája.
- Az eljárás jelentése: az `S` lista az `L` lista kulcs értékei szerinti szabványos (`@<` általi) rendezése, ismétlődéseket nem szűr.

Deklaratív programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

```
„Hegymányos” beépített eljárások
```

387

## Kifejezések kírása — felhasználó vezérelte formázás

- `print(@X)`: Alapértelmezésben azonos `write-tal`. Ha a felhasználó definiál egy `portray/1` eljárást, akkor a rendszer minden a `print-tel` kinyomatolandó rész kifejezésre meghívja `portray-t`. Ennek sikere esetén feltételezni, hogy a kírás megtörtént, meghívás esetén maga írja ki a rész kifejezést. A rendszer a `print` eljárást használja a változó-behelyettesítések és a nyomonkövetés kírásaira!
- `portray(@Kif)` (felhasználó által definiálandó ún. *kampó eljárás*): Igaz, ha `Kif` kifejezést a Prolog rendszernek *nem* kell kírnia (és ekkor maga a `portray` kell, hogy elvégezze a kírást).

- Példa:

```
portray(Matrix) :-
 Matrix = [[_|_|_|_|_|],
 (member(Row, Matrix),
 n1, print(Row), fail
);
 true
],
 [5,6] ?
```

|       |                               |
|-------|-------------------------------|
| X =   | ? - X = [[1,2], [3,4], [5,6]] |
| [1,2] |                               |
| [3,4] |                               |
| [5,6] | ?                             |

Deklaratív programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

## Kifejezések kírása

- `write(@X)`: Kírja `X-et`, ha szükséges operátorokat, zárójelket használva.
- `writeln(@X)`: Mint `write(X)`, csak gondoskodik, hogy szükség esetén az atomok idézőjelek közé legyenek téve.
- `write_canonical(@X)`: Mint `writeln(X)`, csak operátorok nélkül, minden struktúra szabványos alakban jelenik meg.
- `write_term(@X, +Opciók)`: Az Opciók opciólista szerint kírja `X-et`.
- `format(@Formatum, @Adatlista)`: A `Formatum-nak` megfelelő módon kírja `Adatlista-t`. A formázójelke alakja: `{(szám esetleg)}{formázójel}`.
 

```
| ? write('Helló világ'). => Helló világ
| ? writeln('Helló világ?'). => 'Helló világ?'
| ? write_canonical('*? - ?/?'). => -(*,?/?)
| ? write_canonical([1,2]). => ?(1,?(2,[]))
| ? write_term([1,2,3], [max_depth(2)]). => [1,2|...]
| ? format('X=-s --- ?3d s?', [[0?j,0?d],3245]). => X=jó --- 3.245 s
```

Deklaratív programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

```
„Hegymányos” beépített eljárások
```

388

## Karakterek kírása és beolvassása

- `put_code(+Kód)`: Kírja az adott kódú karaktert.
- `tab(+N)`: Kír `N` szóközt feltéve, hogy `N > 0`.
- `n1`: Kír egy soremelést.
- `get_code(?Kód)`: Beolvas egy karaktert és (karakterkódját) egyesíti `Kód-dal`. (File végénél `Kód = -1`.)
- `peek_code(?Kód)`: A soronkövetkező karakter kódját egyesíti `Kód-dal`. A karaktert nem távolítja el a bemenetről. (File végénél `Kód = -1`.)

- Példa:

```
% L a következő sor karakterkódjainak listája.
rd_line(L) :- peek_code(0:\n), !, get_code(_), L = [].
rd_line([_|_]) :- get_code(C), rd_line(L).
| ? rd_line(L), tab(20), member(X, L), put_code(X), tab(1), fail ; n1.
| : Hello world!
```

```
H e l l o w o r l d !
```

Deklaratív programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

```
% számbe(Szám): a Szám szám következik az input-folyamban.
számbe(Szám) :-
 számjegy(Érték), számbe(Érték, Szám).
% Az eddig beolvasott Szám0-val együtt az input-folyamban következő
% szám értéke Szám.
számbe(Szám0, Szám) :-
 számjegy(E, !, Szám1 is Szám0*10+E, számbe(Szám1, Szám)).
számbe(Szám, Szám).
% Érték értékű számjeggy következnek.
számjegy(Érték) :-
 peek_code(Kar), Kar >= 0'0, Kar <= 0'9, get_code(_),
 Érték is Kar - 0'0.
| ?- számbe(X), get_code(_), számbe(Y).
| : 123 456
 => X = 123, Y = 456
```

## Be- és kiviteli csatornák

- Csatornák megnyitása és kezelése:
  - `open(@Filenév, @Mód, -Csatorna)`: Megnyitja a Filenév nevű állományt Mód módban (`read`, `write` vagy `append`). A Csatorna argumentumban visszaadja a megnyitott csatorna „nyelét”.
  - `set_input(@Csatorna), set_output(@Csatorna)`: Az ezt követő beviteli/kiviteli eljárások Csatorna-t használják majd („jelenlegi csatorna”).
  - `current_input(?Csatorna), current_output(?Csatorna)`: A jelenlegi beviteli/kiviteli csatornát egyesíti Csatorna-val.
  - `close(@Csatorna)`: Lezárja a Csatorna csatornát.
- Explicit csatornamegadás be- és kiviteli eljárásokban
  - Az eddig ismertetett összes be- és kiviteli eljárásnak van egy eggyel több argumentumú változata, amelynek első argumentuma a csatorna. Ezek: `write/2`, `writeln/2`, `write_canonical/2`, `write_term/3`, `print/2`, `read/2`, `read_term/3`, `format/3`, `put_code/2`, `tab/2`, `nl/1`, `get_code/2`, `peek_code/2`.

## Kifejezések beolvasása

- `read(?Kif)`: Beolvas egy ponttal lezárt kifejezést és egyesíti Kif-fel. (File végénél Kif = `end_of_file`)
- `read_term(?Kif, +Opciók)`: Mint `read/1`, de az Opciók opciólistát is figyelembe vesz.
  - Példa — botcsinálta programbeolvasó:
 

```
consult_body :-
 repeat, read(Term),
 (Term = end_of_file -> true
 ; assertz(Term), fail
), !.
| ?- listing([p/1]).
p(A) :-
 q(A),
 r(A).
| : ?D
yes
```

## Egy egyszerűbb be- és kiviteli szervezés: DECIO I/O

- `see(@Filenév)`, `tell(@Filenév)`: Megnyitja a Filenév file-t olvasásra/írásra és a jelenlegi csatornává teszi. Újabb híváskor csak a jelenlegi csatornává teszi.
- `seeing(?Filenév)`, `telling(?Filenév)`: A jelenlegi beviteli/kiviteli csatorna állománynevét egyesíti Filenév-vel.
- `seen`, `told`: Lezárja a jelenlegi beviteli/kiviteli csatornát.
- Példák — nagyon egyszerű `consult` variánsok:
 

|                                                                                                                                                                                   |                                                                                                                                                                              |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>consult_dec10_style(File) :-     seeing(Old), see(File),     repeat, read(Term),     ( Term = end_of_file     -&gt; seen     ; assertz(Term), fail     ), !, see(Old).</pre> | <pre>consult_with_streams(File) :-     open(File, read, S),     repeat, read(S, Term),     ( Term = end_of_file     -&gt; close(S)     ; assertz(Term), fail     ), !.</pre> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## Hibakezelési beépített eljárások

- Hibahelyzetet beépített eljárás rossz argumentumokkal való meghívása, vagy a `throw/1` (`raise_exception/1`) eljárás válthat ki.
- Minden hibahelyzetet egy `Prolog` kifejezés (m. `hiba-kifejezés`) jellemez.
- `Hiba`, `dobása`: `throw(@HibaKif), raise_exception(@HibaKif)`
- `Hatása`: Kiváltja a `HibaKif` hibahelyzetet.
- `Hiba`, `elkapása`: `catch(:+Cél1, ?Minta, :+Hibaág), on_exception(?Minta, :+Cél1, :+Hibaág)`
- `Hatása`: Futtatja a `Cél1` hívást.
  - Ha `Cél1` végrehajtása során hibahelyzet nem fordul elő, futtása azonos `Cél1`-al.
  - Ha `Cél1`-ban hiba van, a `hiba-kifejezést` egyesíti `Minta`-val.
  - Ha ez sikeres, meghívja a `Hibaág`-at.
  - Ellenkező esetben továbbdobja a `hiba-kifejezést`, hogy a további körülvévő `catch` eljárások esetleg elkaphassák azt.

Deklaratív programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

„Hagyományos” beépített eljárások

385

## Programfejlesztési eljárások (folytatás)

- `statistics`: Különbféle statisztikákat ír ki az aktuális kimenetre.
- `statistics(?Fajta, ?Érték)`: Érték a `Fajta` fajtaú mennyiség értéke.
  - **Példa**: `statistics(runtime, E) => E=[Tdiff, T], Tdiff az előző lekérdezés óta, T a rendszerindítás óta eltelt idő, ezredmásodpercben.`
- `break`: Egy új interakciós szintet hoz létre.
- `abort, halt`: Kilép a legkülső interakciós szintre ill. a `Prolog` rendszerből.
- `trace`: Elindítja az interaktív nyomonkövetést.
- `debug, zip`: Elindítja a `szelektív nyomonkövetést`, csak `spion-pontoknál` áll meg. (A `zip` mód gyorsabb, de nem gyűjt annyi információt mint a `debug` mód.)
- `nodebug, notrace, nozip`: Leállítja a nyomonkövetést.
- `spy(:@ELjárásSpec)`: Spion-pontot tesz a megadott eljárásokra.
- `nospy(:@ELjárásSpec)`: Megszünteti a megadott spion-pontokat.
- `nospyall`: Az összes spion-pontot megszünteti.

Deklaratív programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

## Programfejlesztési beépített eljárások (SICStus specifikusak)

- `set_prolog_flag(+Jelző, @Érték)`: Jelző értékét `Érték-re` állítja.
- `current_prolog_flag(?Jelző, ?Érték)`: Jelző pillanatnyi értéke `Érték`.
- **Néhány fontos `Prolog` jelző**:
  - `language`: végrehajtási mód (`sicstus`, `iso`).
  - `argv`: csak olvasható, a paraméterbeli argumentumok listája.
  - `unknown`: viselkedés definiálatlan eljárás hívásakor (`trace`, `fail`, `error`).
  - `source_info`: forrásszintű nyomonkövetés (`on`, `off`, `emacs`).
- `consult(:@Files), [:@File, ...]`: Betölti a `File(ek)`at, interpretált alakban.
- `compile(:@File)`: Betölti a `File(ek)`at, lefordított alakot hozva létre.
- `listing`: Kijűa az összes interpretált eljárást az aktuális kimenetre.
- `listing(:@ELjárásSpec)`: Kijűa a megnevezett interpretált eljárásokat.
- **Itt és később**: `ELjárásSpec` — név vagy funktor, esetleg `modul-kvalifikációval` ellátva, ill. ezek listája, pl. `listing(p), listing([m:q,p/1])`.

Deklaratív programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

## NYELVTANI ELEMZÉS PROLOGBAN

## Egy egyszerű nyelvtani elemzési példa

- Bináris számok nyelvtana
 

```

<szám> ::= <számjegy> / <számmaradék>
<számmaradék> ::= <számjegy> / <számmaradék> | ε
<számjegy> ::= 0 | 1

```
- Ugyanez DCG (Definite Clause Grammar) jelöléssel:
 

```

szám --> számjegy, számmaradék.
számmaradék --> számjegy, számmaradék | "".
számjegy --> "0" | "1".

```
- A definit klóz nyelvtan (DCG):
  - egy általános nyelvtani formalizmus,
  - amely egyszerűen Prologra fordítható,
  - a legtöbb Prolog rendszer része (bár a szabványának nem).

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## A DCG szabályok lefordított alakja

- A DCG mint szintaktikai édesítőszerszerű példa:

```

szám --> számjegy, számmaradék. % A | B ≡ A ; B
számmaradék --> számjegy, számmaradék | "". % "" ≡ []
számjegy --> "0" | "1". % "0" ≡ [48]

```

- A fenti DCG szabályok betöltésekora következő Prolog kód keletkezik:
 

```

szám(L0, L) :- számjegy(L0, L1), számmaradék(L1, L).
számmaradék(L0, L) :- (számjegy(L0, L1), számmaradék(L1, L)
 ; L = L0
).
számjegy(L0, L) :- ('0'(L0, 48, L) ; '1'(L0, 49, L)).

% A DCG elemző futtatása:
| ?- szám("101", []). => yes % "101" == [0'1,0'0,0'1]
| ?- szám("102", L). => L = [0'2] ; L = [0'0,0'2] ; no

```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## Nyelvtani elemzés „bevitése” Prologba

- Nyelvtani elemzés: annak eldöntése, hogy egy (Prolog listában tárolt) jelsoorolat megfelel-e egy adott nem-terminális nyelvtani fogalomnak.
- A lista lehet karakterkódok listája, lexikai elemek (token-ek) listája stb.
- Egy nem-terminálisnak egy kétargumentumú Prolog szabály felel meg:
 

```

szám --> számjegy, számmaradék.
szám(L0, L) :- számjegy(L0, L1), számmaradék(L1, L).

% Az L0 kódlistánól "lelemezhető" egy <szám>, marad L ha
% L0-ról lelemezhető egy <számjegy>, marad L1, és
% L1-ról lelemezhető egy <számmaradék>, marad L.

```
- Általánosan: az adott nem-terminálisnak megfelelő jelsoorozatot „lelemezve” (lehagyva) egy L0 lista elejétől marad egy L lista.
- Terminális szimbólumok esetén egyetlen elemet kell leahagyni a listáról, erre szolgál a 'C'/3 beépített eljárás. Definiója: 'C'(L0, X, L) ≡ L0 = [X|L] (A SICStus fordító a két hívást pontosan ugyanúgy fordítja).

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## Vezérlési szerkezetek DCG szabályokban

- DCG szabályokban használható: vágó, diszjunktív szerkezet.
- Ezek változtatás nélkül átkerülnek a Prolog alakba. Példák:
 

```

% Lelemezhető számjegyek egy MAXIMÁLIS (esetleg üres) lista.
számmaradék --> (számjegy -> számmaradék
 ; []
).

% Ugyanez vágósóval
számmaradék --> számjegy, !, számmaradék.
számmaradék --> [].

% Az utóbbi Prolog alakja:
számmaradék(L0, L) :- számjegy(L0, L1), !, számmaradék(L1, L).
számmaradék(L0, L) :- L = L0.

| ?- számmaradék("102", L). => L = [0'2] ; no

```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## Prolog hívás beillesztése DCG szabályba

- Általánosabb példa: decimális számjegyek elemzése

```
szám_jegy --> "0" ; "1" ; "2" ; "3" ; "4" ;
 "5" ; "6" ; "7" ; "8" ; "9" .

% Ugyanez általánosabban és egyszerűbben:
szám_jegy --> [K],
 {decimális_jegy_kódja(K)}.

% K egy számjegy kódja.
decimális_jegy_kódja(K) :- K >= 0'0, K =< 0'9.
```

- A fenti DCG szabály Prolog megfelelője:

```
% Lelemezhető egy számjegy kódja.
szám_jegy(L0, L) :-
 'C'(L0, K, L), % K a következő kód
 decimális_jegy_kódja(K). % megfelel-e a K?
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## A DCG nyelvtani szabályok szerkezete — összefoglalás

- A DCG szabály alakja:  $\langle Baloldal \rangle \rightarrow \langle Jobboldal \rangle$ .
- $\langle Baloldal \rangle$ : egy nem-terminális, ami esetleg terminálisok listája követi.
- $\langle Jobboldal \rangle$ : konjunkció ( $\wedge$ ), diszjunkció ( $\vee$ ), ha-akkor ( $\rightarrow$ ) és negáció ( $\neg$ ) segítségével épült terminálisokból, nem-terminálisokból és Prolog hívásokból.
- Nem-terminális: tetszőleges *hívható* kifejezés (atom vagy struktúra).
- Terminális: *tetszőleges* Prolog kifejezés; 0, 1 vagy több terminális jel sorozata *listaként* helyezhető el a DCG szabályokban.
- Prolog hívás:  $\{ \}$  zárójelbe zárva helyezhető el (vágó köré nem kell zárójel).
- DCG  $\equiv$  egy „ingyven” akkumulátor (akkum. lépés:  $\text{?C}'$ , egy elem levétele):

```
p(A, ...) -->
 q0(B, ..., [X], q1(C, ...), ..., {Gél}, ..., qn(D, ...)).

p(A, ..., L0, L) :-
 q0(B, ..., L0, L1), ..., ?C'(L1-1, X, L1), q1(C, ..., L1, L1+1), ...,
 Gél, ..., qn(D, ..., Ln, L1).
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## Az elemző kiegészítése argumentumokkal

- Egy DCG szabály argumentumaiban egy „belső” alakot építhet:

```
% Lelemezhető egy Sz értékű decimális számjegy-sorozat
szám(Sz) --> szám_jegy(J), számaradék(J, Sz).

% Lelemezhető számjegyek egy esetleg üres listája, amelynek
% az eddig lelemzett Sz0-val együtt vett értéke Sz.
számaradék(Sz0, Sz) -->
 szám_jegy(J), I, {Sz1 is Sz0*10+J}, számaradék(Sz1, Sz).
számaradék(Sz0, Sz0) --> [].

% Lelemezhető egy J értékű számjegy.
szám_jegy(J) --> [K], {decimális_jegy_kódja(K), J is K-0'0}.

I ?- szám(Sz, "102 56", L). => L = " 56", Sz = 102; no
```

- A számaradék DCG szabály Prolog alakja:

```
számaradék(Sz0, Sz, L0, L) :-
 szám_jegy(J, L0, L1), I, Sz1 is Sz0*10+J, számaradék(Sz1, Sz, L1, L),
 számaradék(Sz0, Sz0, L0, L) :- L=L0.
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## DCG példa: kifejezés kiértékelése

```
% kif(Z, L0, L): L0 elején egy Z értékű aritm. kifejezés áll, marad L.
kif(Z) --> tag(X), "+", kif(Y), {Z is X + Y}.
kif(Z) --> tag(X), "-", kif(Y), {Z is X - Y}.
kif(X) --> tag(X).

% tag(Z, L0, L): L0-ból lelemezhető egy Z értékű tag, marad L.
tag(Z) --> szám(X), "*", tag(Y), {Z is X * Y}.
tag(Z) --> szám(X), "/", tag(Y), {Z is X / Y}.
tag(X) --> szám(X).
```

```
I ?- kif(Z, "10*10-6*6", ""). => Z = 64 ; no
I ?- kif(Z, "10*10-6*6", L). => L = [], Z = 64 ;
 L = [42, 54], Z = 94 ; ...

I ?- kif(Z, "4-2+1", []). => Z = 1 % Jobbról balra elemezi!

% Egy lehetséges javítás:
kif(Z) --> tag(X), kifmaradék(X, Z).
kifmaradék(X0, Z) --> "+", tag(X1), {X is X0 + X1}, kifmaradék(X, Z).
...
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## DCCG példa: „természetes” nyelvű beszélgetés

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

% mondat(Alany, Áll, L0, L) : L0-L kiemelhető egy Alany alanyból és Áll
% állítmányból álló mondatot. Alany lehet első vagy második személyű
% névmás, vagy egyetlen szóból álló (harmadik személyű) alany.
mondat(Alany, Áll) --> {én_te(Alany, Ige)}, én_te_perm(Alany, Ige, Áll),
mondat(Alany, Áll) --> szó(Alany), szavak(Áll).

% én_te(Alany, Ige) :
% Az Alany első/második személyű névmásnak megfelelő létige az Ige.
én_te("én", "vagyok").
én_te("te", "vagy").

% én_te_perm(Ki, Ige, Áll, L0, L) : L0-L kiemelhető egy Ki
% névmásból, Ige ígealaktól és Áll állítmányból álló mondatot.
én_te_perm(Alany, Ige, Áll) --> szó(Alany), szó(Ige), szavak(Áll).
én_te_perm(Alany, Ige, Áll) --> szó(Alany), szavak(Áll), szó(Ige).
én_te_perm(Alany, Ige, Áll) --> szavak(Áll), szó(Ige), szó(Alany).
én_te_perm(Alany, Ige, Áll) --> szavak(Áll), szó(Ige).
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## Példa: „természetes” nyelvű beszélgetés — párbeszéd-szervezés

```
% :- type mondas ---> kérdez(szó) ; kijelent(szó, list(szó)) ; un.

% Megvalósít egy párbeszédet.
párbeszéd :-
 repeat, rd_line(L),
 (
 menet(Mondas, L, []) -> feldolgoz(Mondas)
 ;
 write('Nem értem\n'), fail
),
 Mondás = un, !.

% menet(Mondas, L0, L) : Az L0-L kiemezett alakja Mondás.
menet(kérdez(Alany)) --> {kérde(Szó)}, mondat(Alany, [Szó]), "?".
menet(kijelent(Alany, Áll)) --> mondat(Alany, Áll), " ".
menet(un) -->
 szó("unlak"), " ".

% kérde(Szó) : Szó egy kérdészavó.
kérdő("mi").
kérdő("ki").
kérdő("kicsoda").
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## Példa: „természetes” nyelvű beszélgetés — szavak elemzése

```
% szó(Sz, L0, L) : L0-L egy Sz betűsorozatból álló (nem üres) szó.
szó(Sz) --> betű(B), számaradék(SzW), {illik([B|SzW], Sz)}, köz.

% számaradék(Sz, L0, L) : L0-L egy Sz kódlistából álló (esetleg üres) szó.
számaradék([B|Sz]) --> betű(B), !, számaradék(Sz).
számaradék([]) --> [].

% illik(Szó0, Sz0) : Sz00 = Sz0, vagy a kezdő kis-nagy betűen különböznek.
illik([B0|_], [B|_]) :- (B = B0 -> true ; abs(B-B0) =:= 32).

% köz(L0, L) : L0-L nulla, egy vagy több szóköz.
köz --> (" " -> köz ; "").

% betű(K, L0, L) : L0-L egy K kódú "betű" (különbözik a " ?" jelektől).
betű(K) --> [K], {non_member(K, " ?")}.

% szavak(SzL, L0, L) : L0-L egy SzL szó-lista.
szavak([Sz|SzK]) --> szó(Sz), (
 ; {SzK = []}
).

% dynamic tudom/2.
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## Példa: „természetes” nyelvű beszélgetés — válaszok előállítása

```
% :- dynamic tudom/2.

% feldolgoz(Mondas) : feldolgozza a felhasználótól érkező Mondás üzenetet.
feldolgoz(un) :- write('Én is.\n').
feldolgoz(kijelent(Alany, Áll)) :-
 assertz(tudom(Alany, Áll)), write('Felfogtam.\n').
feldolgoz(kérdez(Alany)) :-
 tudom(Alany, _), !, válasz(Alany).
feldolgoz(kérdez(_)) :-
 write('Nem tudom.\n').

% felsorolja az Alany ismert tulajdonságait.
válasz(Alany) :- tudom(Alany, Áll),
 (
 member(Szó, Áll), format('~s ', [Szó]), fail
 ;
 n1, fail
),
 válasz(_).
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## Beszélgetős DCG példa — egy párbeszéd

|                                                                                                                                                                                                                                                                   |                                                                                                                                                                                                                                                     |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> I ?- párbeszéd. I : Magyar Legény vagyok én. Felfogtam. I : Ki vagyok én? Magyar Legény I : Péter kicsoda? Nem tudom. I : Péter tanuló. Felfogtam. I : Péter jó tanuló. Felfogtam. I : Péter kicsoda? tanuló jó tanuló I : Boldog vagyok. Felfogtam. </pre> | <pre> I : Én vagyok Jeromos. Felfogtam. I : Te egy Prolog program vagy. Felfogtam. I : Ki vagyok én? Magyar Legény Boldog Jeromos I : Okos vagy. Felfogtam. I : Ki vagy te? egy Prolog program Okos I : Valóban? Nem értem I : Unlak. Én is. </pre> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## A DCG formalizmus felhasználása elemzésen kívül

- DCG szabályok kényelmesen használhatók általános akkumulálásra
  - Listák akkumulálása (nemsak elemzés, építés is)

```

% anbn(+N, ?L) : Az L lista N db a-ból és azt követő N db b-ből áll.
anbn(N, L) :- anbn(N, L, []).
% anbn(N, L0, L) : L0-L N db a-ból és azt követő N db b-ből áll.
anbn(0) --> !.
anbn(N) --> {N > 0, M1 is N-1}, [a], anbn(M1), [b].
% a fenti DCG szabály kifejtve:
anbn(N, L0, L) :-
 N > 0, M1 is N-1, L0=[a|L1], anbn(M1, L1, L2), L2=[b|L].

```
  - Egyébként az elemi akkumulálási lépést DCG-n kívül kell megírni:

```

% sum(L, S0, S) : L összege S-S0.
sum([], S0) --> S0.
sum([X|L], S0) -->
 sum(L, S1) :- sum(L, 0, S1).
plus(X, S0, S) :- S is S0+X.

```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

Fejlettebb nyelvi és rendszerelemek 412

## Külső nyelvi interfész

- Hagyományos (pl. C nyelvű) programrészek meghívásának módja:
  - A Prolog rendszer elvégzi az átalakítást a Prolog alak és a külső nyelvi alak között. Kényelmesebb, biztonságosabb mint a másik módszer, de kevésbé hatékony. Többnyire csak egyszerű adatokra (egész, valós, atom). (MProlog)
  - A külső nyelvi rutin pintereket kap Prolog adatstruktúrákra, valamint hozzáférési algoritmusokat ezek kezelésére. Nehézkesebb, veszélyesebb, de jóval hatékonyabb mint az előző megoldás. Összetett adatok adásvételére is jó. (SWI, SICStus)

## FEJLETTEBB NYELVI ÉS RENDSZERELEMENK

Deklaratív Programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

## Külső nyelvű interfész — példa

Prologban az `index_keys` (*Spec*, *K<sub>if</sub>*, *Kulcs*, *Szám*) eljárást szeretnénk meghívni, aminek a jelentése:

- Ha *Spec* és *K<sub>if</sub>* különböző funktorú kifejezések, akkor *Szám* = -1 és *Kulcs* = `[]`.
- Egyébként, ha *Spec* valamelyik argumentuma + és *K<sub>if</sub>* megfelelő argumentuma változó, akkor *Szám* = -2 és *Kulcs* = `[]`.
- Egyébként *Szám* a *Spec* argumentumaként előforduló + atomok száma, *Kulcs* pedig *K<sub>if</sub>* megfelelő argumentumok *kiönatából* képzett lista. A kiönat lényegében az argumentum funktora, azzal az eltéréssel, hogy a konstansok kiönata maga a konstans, strukturák esetén pedig a struktúra neve és az ariása külön elemként kerül a kiönat-listába.

Deklaratív programozás, BMEF, 2001 tavaszi félév

20. eladás (logikai programozás)

Féjletlebb nyelvű és rendszerelenk

415

## Külső nyelvű interfész — a C kód (ixkeys.c állomány)

```
#include <stdbool.h>
#define MA -1 /* not applicable */
#define NI -2 /* instantiatédness */
long ixkeys(SP_term_ref spec,
 SP_term_ref term, SP_term_ref list)
{
 unsigned long sname, tname, plus;
 int sarity, tarity, i;
 long ret = 0;
 SP_term_ref arg = SP_new_term_ref(),
 tmp = SP_new_term_ref();
 SP_get_functor(spec, &sname, &sarity);
 SP_get_functor(term, &tname, &tarity);
 if (sname != tname || sarity != tarity)
 return NI;
 plus = SP_atom_from_string("++");
 for (i = sarity; i > 0; --i) {
 unsigned long t;
 SP_get_arg(i, spec, arg);
 SP_get_atom(arg, &t); /* no check */
 if (t != plus) continue;
 SP_get_arg(i, term, arg);
 switch (SP_term_type(arg)) {
 case SP_TYPE_VARIABLE:
 return NI;
 case SP_TYPE_COMPOUND:
 SP_get_functor(arg, &tname, &tarity);
 SP_put_integer(tmp, (long)tarity);
 SP_cons_list(list, tmp, list);
 SP_put_atom(arg, tname);
 break;
 case SP_TYPE_LIST:
 SP_cons_list(list, arg, list); ++ret;
 }
 }
 return ret;
}
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

20. eladás (logikai programozás)

## Külső nyelvű interfész — példa

- A példaeljárás használata

```
| ?- [ixtest].
| ?- index_keys(f(+, -, +, +),
 f(12.3, -, s(1, -, z(2)), t),
 L, X).
L = [12.3, s, 3, t], X = 3 ?
yes
```

- Az `ixtest.pl` file.

```
foreign(ixkeys, index_keys(+term, +term, -term, [-integer])).
foreign_resource(ixkeys, [ixkeys]).
:- load_foreign_resource(ixkeys).
```

- A C programot elő kell készíteni a Prolog számára az `splfr` eszköz segítségével:
 

```
splfr ixkeys ixtest.pl +c ixkeys.c
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

20. eladás (logikai programozás)

Féjletlebb nyelvű és rendszerelenk

416

## Hasznos lehetőségek SICStus Prolog-ban

- Tetszőleges nagyságú egész számok pl.:
 

```
| ?- fakt(40, F).
```

```
F = 815915283247897734345611269596115894272000000000 ?
```

- Globális változók (Blackboard)

```
bb_put(Kulcs, Érték)
A Kulcs kulcs alatt eltárolja Érték-et, az előző értékét, ha van, törölve.
(Kulcs egy (kls) egész szám vagy atom lehet.)
```

```
bb_get(Kulcs, Érték)
```

```
Előhívja Érték-be a Kulcs értékét.
```

```
bb_delete(Kulcs, Érték)
```

```
Előhívja Érték-be a Kulcs értékét, majd kitörli.
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

20. eladás (logikai programozás)



## Hasznos lehetőségek SICStus Prolog-ban (folytatás)

- Visszaleítható módon változtatható kifejezések

```
create_mutable(Adat, ValtkiF)
```

Adat kezdőértékkel létrehoz egy új változtatható kifejezést, ez lesz ValtkiF.

Adat nem lehet üres változó.

```
get_mutable(Adat, ValtkiF)
```

Adat-ba előveszi ValtkiF pillanatnyi értékét.

```
update_mutable(Adat, ValtkiF)
```

A ValtkiF változtatható kifejezés új értéke Adat lesz. Ez a változtatás visszalépéskor visszacsinálódik. Adat nem lehet üres változó.

- Takarító eljárás

```
call_cleanup(Hivas, Tiszto)
```

Meghívja call(Hivas)-t és ha az véglegesen befejezte futását, meghívja Tiszto-t. Egy eljárás akkor fejezte be véglegesen a futását, ha további alternatívák nélkül sikerült, meghívásult vagy kivételt dobott.

## Blokk-deklarációk (folytatás)

- Generál-és-ellenőriz típusú programok gyorsítása

- általában nem hatékonyak (pl megra\_jozlja\_1), mert túl sok visszalépést használnak

- korutínszervezéssel a generáló és ellenőrző rész „automatikusán” összefűsíthető

- ehhez az ellenőrző részt kell előre tenni és megfelelően blokkolni

- Korutínszervezésre építő programok

- Példa: egyszerűsített Hamming feladat

- Keressük a  $2^i * 3^j$  ( $i \geq 1, j \geq 1$ ) alakú számok közül az első N darabot nagyság szerint rendezve.

- „stream-and-parallelism” közelítésmódot használva korutínszervezéssel egyszerűen lehet megoldani

## Fejlett vezérlési lehetőségek SICStusban: Blokk-deklarációk

- Példa:

```
:- block p(-, ?, -, ?, ?).
```

Jelentése: ha az első és a harmadik argumentum is behelyettesíthetőn változó (blokkolási feltétel), akkor a p hívás felfüggesztődik.

Úgyanarra az eljárásra több vagy/ugos feltétel is szerepelhet, pl.

```
:- block p(-, ?, ?), p(?, -).
```

- Végtelen választási pontok kiküszöbölése blokk-deklarációval

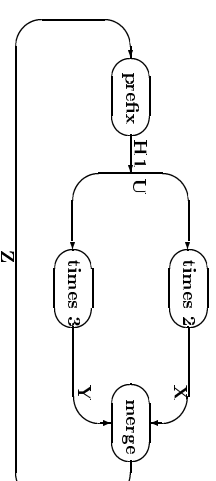
```
:- block append(-, ?, -).
```

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

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

```
append(L1, L2, L3).
```

## Hamming probléma



% A H lista az első N, csak a 2 és 3 tényezőkből álló szám.

```
hamming(N, H) :-
```

```
U = [1|H], times(U, 2, X), times(U, 3, Y),
```

```
merge(X, Y, Z), prefix(N, Z, H).
```

```
% times(X, M, Z): A Z lista az X elemeinek M-szerese
```

```
:- block times(-, ?, ?).
```

```
times([_|X], M, Z) :- B is M*A, Z = [B|U], times(X, M, U).
```

```
times([], _, []).
```

## Hamming probléma (folyt.)

```
% merge(X, Y, Z): Z az X és Y összerésztülése.
:- block merge(-, ?, ?), merge(?, -, ?).
% Csak akkor fusson, ha az első két argumentum ismert
merge([A|X], [B|Y], V) :-
 A < B, !, V = [A|Z], merge(X, [B|Y], Z).
merge([A|X], [B|Y], V) :-
 B < A, !, V = [B|Z], merge([A|X], Y, Z).
merge([A|X], [A|Y], [A|Z]) :-
 merge(X, Y, Z).
merge([], X, X) :- !.
merge(_, [], []).

% prefix(N, X, Y): Az X lista első N eleme Y.
prefix(0, _, []) :- !.
prefix(N, [A|X], [A|Y]) :-
 N > 0, N1 is N-1, prefix(N1, X, Y).
```

Deklaratív Programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

## SICStus könyvtárak

- Könyvtár betöltése
  - :- use\_module(library(*könyvtárnév*)).
- A legfontosabb könyvtárak
  - arrays Logaritmikus elérési idejű kiterjeszhető tömbök megvalósítását tartalmazza.
  - assoc AVL fák segítségével valósítja meg az „asszociációs listák”, azaz véges Prolog kifejezésekkel definiált kiterjeszhető leképezések fogalmát.
  - atts tetszőleges attribútumokat enged a Prolog változókhoz rendelni, ezeket tárolóeszközként és a Prolog egyesítési mechanizmusának módosítására is engedi használni.
  - heaps A bináris kazal (heap) fogalmát valósítja meg, amely főként prioritásos sorok (priority queue) megvalósítására használható.
  - lists Biztosítja a listakezelő alapműveleteket.

Deklaratív Programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

## Korútszervező eljárások

- freeze(X, Hivas)
  - Hivast felfüggeszti mindaddig, amíg X behelyettesíthetően változó.
- frozen(X, Hivas)
  - Az X változó miatt felfüggesztett hívás(ok)at egyesíti Hivas-sal.
- dif(X, Y)
  - X és Y nem egyesíthető. Mindaddig felfüggesztődik, amíg ez el nem dönthető.
- call\_residue(Hivas, Maradék)
  - Hivas-t végrehajtja, és ha a sikeres lefutás után maradnak felfüggesztett hívások, akkor azokat visszaadja Maradékban. Pl.
 

```
| ?- call_residue(dif(X, f(Y)), Maradék).
 => Maradék = [X]-(prolog:dif(X,f(Y)))
| ?- call_residue((dif(X, f(Y)), X=f(Z)), Maradék).
 => X = f(Z), Maradék = [[Y,Z]-(prolog:dif(f(Z),f(Y)))]
```

Deklaratív Programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

- terms Külföldi kifejezéskezelő eljárásokat tartalmaz.
- ordsets Halmazműveleteket definiál, ahol a halmazokat a Prolog szabványos rendezése szerint (compare) rendezett listákkal ábrázolja.
- queues Sorokra (queue, FIFO store) vonatkozó műveleteket definiál.
- random Egy véletelenszám-generátort tartalmaz.
- system Külföldi operációsrendszer-szolgáltatások elérését biztosítja.
- trees Az arrays könyvtárhoz hasonló, de nem-kiterjeszhető logaritmikus elérési idejű tömbfogalmat valósít meg, bináris fák segítségével (kicsit hatékonyabb mint az arrays könyvtár).
- ugrraphs Irányított és irányítatlan gráf fogalmat valósít meg, élcimkék nélkül.
- wgraphs Olyan irányított és irányítatlan gráf fogalmat valósít meg, ahol minden él egy egészértékű súllyal rendelkezik.
- sockets A socket-ek kezelésére szolgáló eljárásokat biztosít.
- linda/client és linda/server Linda-szerű processzorkommunikációs eszközöket ad.

Deklaratív Programozás, BME, 2001 tavaszi félév

20. előadás (logikai programozás)

- `bdb` Felhasználó által definiált többszörös indexelést lehetővé tevő, Prolog kifejezések lemezen való tárolására szolgáló adatbázis-rendszer.
- `clpb` Boole-értékekre vonatkozó feltétel-megoldó (`constraint solver`).
- `clpbq` és `clpr` Feltétel-megoldó a `Q` (racionális számok) ill. `R` (valós számok) tartományán.
- `clpfd` Végess tartományokra vonatkozó feltétel-megoldó (`constraint solver`).
- `cltcl` A *Tcl/Tk* nyelv és eszközkészlet elérését biztosítja.
- `geauge` Prolog programok a profilvizására szolgáló, a `tcltk -n` alapulóg grafikus interfésszel rendelkező eszköz.
- `charsio` Karaktársorozatból olvasó ill. abba író be- és kiviteli eljárások gyűjteménye.
- `timeout` Lehetőséget ad arra, hogy célok futási idejét korlátozzuk.
- `xref` A nyomkövetés és a program-analízis segítésére használható keresztreferencia készítő program.

Deklaratív programozás, BMEF, 2001 tavaszi félév

20. előadás (logikai programozás)

Kiírás, nyomkövetés 427

## Kiírás

- `{TextIC }print` : `string -> unit`  
`print s = kiirja az s értékét a standard kimenetre, és azonnal kiírja a puffert.`

- `{Meta.}printVal` : `'a -> 'a`  
`printVal e = kiirja az e kifejezés értékét a standard kimenetre pontosan úgy, ahogyan az SML értelmező írja ki a „legfelső szinten”, és azonnal kiírja a puffert. Eredménytül visszaadja az e kifejezés értékét. Csak interaktív módban használható.`

- **Példák:**

```
- print("alma"~"Korte\n");
 almaKorte
> val it = () : unit
```

*Megjegyzés.* A kapesos zárójelk – { és } – között opcionálisan megadható moduhév áll.

Például `{TextIC }print` azt jelenti, hogy a `TextIC` modulban van definiálva, de az `SML`-értelmező a `print` nevet rövid alakban is felismeri.

Deklaratív programozás, BMEF, 2001 tavaszi félév

22. előadás (funkcionális programozás)

## KIÍRÁS, NYOMKÖVETÉS

Kiírás, nyomkövetés 428

### Kiírás (folyt.)

- `printVal`-al tetszőleges típusú érték íratható ki. További példák:

```
- printVal (3, 5.0);
(3, 5.0)> val it = (3, 5.0) : int * real

- printVal ["#\"A\"", "#\"Z\"", "#\".\""];
["#\"A\"", "#\"Z\"", "#\".\""]> val it = ["#\"A\"", "#\"Z\"", "#\".\""] : char list

- datatype t = L | B of t * t;
 New type names: =t
 datatype t = (t,{con B : t * t -> t, con L : t})
 con B = fn : t * t -> t
 con L = L : t

- val fa = B(B(B(L,B(L,B(L,L))),L),B(L,L));
 > val fa = B(B(B(L, B(L, B(L, B(B(L, L))))), L), B(L, L)) : t
- printVal fa;
B(B(B(L, B(L, B(B(L, L))))), L), B(L, L))> val it = B(B(B(L, B(L, B(B(L, L),
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

22. előadás (funkcionális programozás)

## Kírtás (folyt.)

- Az utolsó példában a kírt sor túl hosszú lett, jó lenne eltörni a > jel előtt. Hogyan írhatunk ki egy újsor-jelét úgy, hogy az eredmény a fa érték maradjon? Például így, de ez elég körülményes:

```
- let val res = printVal fa;
 val _ = print "\n"
in
 res
end;
B(B(B(L, B(L, B(B(L, L))), L), B(L, L)))
> val it = B(B(B(L, B(L, B(B(L, L))), L), B(L, L))) : t
```

- A before operátort az ilyen és hasonló dolgok kezelésére találták ki.

## Szekvenciális kifejezés (;)

- Az (x; y) szekvenciális kifejezés, akárcsak az x before y, szintaktikai édesítőszert. Az (x; y) helyett írhatjuk, hogy:

```
let val _ = x in y end;
```

## Szekvenciális kifejezés (before)

- Az x before y kifejezés az ún. *szekvenciális kifejezés* egy változata. {General }before : 'a \* 'b -> 'a  
x before y = először az x-et, majd az y-t értékel ki, eredménye az x értéke. Precedenccsintűe 0.

- Példa before használatára:

```
- printVal fa before print "\n";
B(B(B(L, B(L, B(B(L, L))), L), B(L, L)))
> val it = B(B(B(L, B(L, B(B(L, L))), L), B(L, L))) : t
```

- Az x before y-hoz hasonló a (x; y) szekvenciális kifejezés, amely azonban az *utolsó* részkifejezésének az értékét adja eredményül.

```
- (print "A fa változó értéke =\n"; printVal fa before print "\n");
A fa változó értéke =
B(B(B(L, B(L, B(L, B(B(L, L))), L), B(L, L)))
> val it = B(B(B(L, B(L, B(B(L, L))), L), B(L, L))) : t
```

## Kírtás (folyt.)

- Hosszú lista, ill. egymásba skatulyázott adatszerkezetek esetén printVal (és maga az SML-értelmező is) alapesetben csak az első 200 listaelemet, ill. legfeljebb 20 szintet ír ki. A hosszat a printLength, a szintek számát a printDepth *frissíthető változó szabályozza*. Mindkét érték felülírható.

```
printLength : int ref | printLength := 7; | printLength;
printDepth : int ref | printDepth := 3; | printDepth;
```

- Példák:

```
- printVal [1,2,3,4,5,6,7,8,9,10] before print "\n";
[1, 2, 3, 4, 5, 6, 7, ...]
> val it = [1, 2, 3, 4, 5, 6, 7, ...] : int list
- printVal fa before print "\n";
B(##, ##)
> val it = B(##, ##) : t
```

- **Figyelem:** a printLength és a |printLength kifejezések különböznek!  
- printLength;           | - |printLength;  
> val it = ref 7 : int ref   |> val it = 7 : int

## Kiráds (folyt.)

- Kétféle típusú egyszerű értékeket alakítanak át füzérré a `toString` függvények:

```
Char.toString : char -> string
Int.toString : int -> string
Real.toString : real -> string
Bool.toString : bool -> string
Word.toString : word -> string
```

## Nyomkövetés (folyt.)

- A `length` iteratív változata

```
fun lengthi xs = let fun len (i, _::xs) = len(i+1, xs)
 | len (i, []) = i
 in len(0, xs)
 end;
```

- A `length` iteratív változata kiegészítve függvényekkel

```
fun lengthi xs =
 let fun len (i, _ : int) :: xs) =
 len((print " "; printVal(printVal i before print " $ ") + 1)),
 (print " & "; printVal xs)
)
 before print "#\n"
 in len(i, []) = (print " * "; printVal i before print " %\n")
 end;
```

## Nyomkövetés

- Az `MOSML`-ben nyomkövetés csak a program szövegébe beírt kiegészítőkkel lehetséges.

- Példa: a `length` függvény két változatának kiértékelése

- A `length „naiv”` változata

```
fun length (_::xs) = 1 + length xs
 | length [] = 0;
```

- A `length „naiv”` változata kiegészítve függvényekkel

```
fun lengthi ((_ : int) :: xs) =
 printVal(1 + (print " & "; printVal(length(printVal xs)))
 before print " $ "
)
 before print "#\n"
 | lengthi [] = (print " * "; printVal 0 before print " %\n");
```

## Nyomkövetés

- `length` és egy alkalmazása

```
fun lengthi ((_ : int) :: xs) =
 printVal(1 + (print " & "; printVal(length(printVal xs)))
 before print " $ "
)
 before print "#\n"
 | lengthi [] = (print " * "; printVal 0 before print " %\n");
```

```
length [1,2,3];
& [2, 3] & [3] & [] * 0 %
0 $ 1 #
1 $ 2 #
2 $ 3 #
```

## Nyomkövetés (folyt.)

- `length` és `xs` alkalmazása

```

fun length xs =
 let fun len (i, (_, _ : int) :: xs) =
 len((print " "; printVal((printVal i before print " $ ") + 1)),
 (print " & "; printVal xs)
)
 before print "#\n"
 in len(0, xs)
 end;

length [1,2,3];
0 $ 1 & [2, 3] 1 $ 2 & [3] 2 $ 3 & [] * 3 %
#
#
#

```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

22. előadás (funkcionális programozás)

## Nyomkövetés (folyt.)

- `length` és `length` kiértékelésének összehasonlítása

```

length [1,2,3]; length [1,2,3];
& [2, 3] & [3] & [] * 0 % 0 $ 1 & [2, 3] 1 $ 2 & [3] 2 $ 3 & [] * 3 %
0 $ 1 # #
1 $ 2 # #
2 $ 3 # #

```

- További példák a 22f.p. sml állományban
- `nodes` és `akkumulátor` használó `nodes` változata
- `depth` és `akkumulátor` használó `depth` változata

Deklaratív Programozás, BMEF, 2001 tavaszi félév

22. előadás (funkcionális programozás)

## Kivételkezelés

- A leggyakoribb belső kivételek (többek között `!d` a `General` könyvtárat)

| Megnevezés             | Művelet, amely a kivételt kiválthatja                                                                   |
|------------------------|---------------------------------------------------------------------------------------------------------|
| <code>Bind</code>      | Értékdeklarációban a jobb oldali kifejezés nem illeszkedik a bal oldali mintára.                        |
| <code>Chr</code>       | <code>chr pred succ</code>                                                                              |
| <code>Div</code>       | <code>/ div mod</code>                                                                                  |
| <code>Domain</code>    | Az érték kívül az értelmezési tartományból.                                                             |
| <code>Empty</code>     | <code>hd tl last</code>                                                                                 |
| <code>Fail</code>      | <code>compile load loadOne</code>                                                                       |
| <code>Interrupt</code> | Megszakítás <code>ctrl/c</code> -vel.                                                                   |
| <code>Io</code>        | Ki/beviteli hiba. <code>!o of {function : string, name : string, cause : exn }</code>                   |
| <code>Match</code>     | Mintaillesztési hiba <code>case</code> és <code>handle</code> kifejezésben, vagy függvényalkalmazásban. |
| <code>Option</code>    | Hiba egy <code>Option</code> könyvtárbeli függvény alkalmazásakor.                                      |
| <code>Ord</code>       | <code>Pl. Nj93. ord "" váltja ki; elavult.</code>                                                       |
| <code>Overflow</code>  | <code>~ + - * / div mod abs ceil floor round trunc</code>                                               |
| <code>Size</code>      | <code>~ array concat fromList implode tabulate translate vector</code>                                  |
| <code>Subscript</code> | <code>copy drop extract nth sub substring take update</code>                                            |

## KIVÉTELKEZELÉS

Deklaratív Programozás, BMEF, 2001 tavaszi félév

22. előadás (funkcionális programozás)

## Kivételkezelés (folyt.)

- Kivételt az `exception` kulcsszóval deklarálunk, a `raise` kulcsszóval jelöljük, a `handle` kulcsszóval bevezetett kifejezésben kezelünk.
- A kivételeket leggyakrabban hibák jelzésére használjuk.
- A kivételkonstruktor lehet állandó vagy függvény.
- A kivételkonstruktorállandó, ill. a kivételkonstruktorfüggvény típusa: `exn`.
- Az `exn` speciális típus:
  - a kivételkonstruktorok halmaza *bővíthető*,
  - az `exn` típust tartalmazó ún. *kivételcsomag* minden típusal kompatibilis:
 

```
- fun // {den = 0, ...} = raise Domain
 | // {num = n, den = d} = (real n) / (real d);
 > val // = fn : {den : int, num : int} -> real
pedig
- Domain;
> val it = Domain : exn
```

Deklaratív Programozás, BME, 2001 tavaszi félév

22. előadás (funkcionális programozás)

## Kivételkezelés (folyt.)

- A `raise` kulcsszó olyan *kivételcsomagot* hoz létre, amelyben `exn` típusú értékek is van.
- A kivétel kezelése a `case`-szerkezetre emlékeztet:
 

```
E handle P1 => E1 | ... | Pn => En.
```
- Ha E „közönséges” értéket ad eredményül, a kivételkezelő egyszerűen továbbadja az eredményt.
- Ha E *kivételcsomagot* eredményez, akkor az SML-futtatórendszer megpróbálja a  $P_1 \dots P_n$  mintákra illeszteni.
  - Ha az első illeszkedő minta a  $P_i$  ( $i = 1, 2, \dots, n$ ), akkor a kivételkezelő eredménye az  $E_i$  kifejezés eredménye.
  - Ha egyetlen minta sem illeszkedik a kivételcsomagra, akkor a kivételkezelő továbbpasszolja a kivételcsomagot az előző hívási szintre.

Deklaratív Programozás, BME, 2001 tavaszi félév

22. előadás (funkcionális programozás)

## Egyszerű műveletek bináris fákban (folyt.)

- `fulltree n` mélységű teljes bináris fát épít, és a fa csomópontjait  $1$ -től  $2^n - 1$ -ig beszámozza. Egy teljes bináris fában minden csomópontból pontosan két él indul ki, és minden levelének ugyanaz a szintje.
 

```
(* fulltree n = n mélységű teljes fa
 fulltree : int -> 'a tree *)
fun fulltree n =
 let fun ftree (_, 0) = L
 | ftree (k, n) = N(k, ftree(2*k, n-1), ftree(2*k+1, n-1))
 in ftree(1, n)
 end;
```
- `reflect` a fát a függőleges tengelye mentén tükrözi.
 

```
(* reflect =
 reflect : 'a tree -> 'a tree *)
fun reflect (N(v,t1,t2)) = N(v, reflect t2, reflect t1)
 | reflect L = L;
```

## BINÁRIS FÁK

Deklaratív Programozás, BME, 2001 tavaszi félév

23. előadás (funkcionális programozás)

**Nyomkövetés: nodes (akkumulátort nem használ)**

```
(* tab : string -> string
 tab i = a sorok behúzásához használandó i függő szökökkel kiegészítve
*)
fun tab i = i ^ " "
 ;

fun nodes f =
 let (* nodes0 i f = a csomópontok száma f-ben; i a behúzásához használt függő
 nodes0 : string -> 'a tree -> int *)
 fun nodes0 i (N(a, t1, t2)) =
 (print("\n" ^ i ^ "<""); printVal a : int; print "> "
 printVal(1 +
 nodes0 (tab i) (printVal t2 before print " *") +
 nodes0 (tab i) (printVal t1 before print " %")
 before print "$ "
)
 before print("#\n" ^ i)
)
 in
 | nodes0 i L = (print("\n" ^ i); 0)
 end;
 nodes0 "" f
end;
```

Deklaratív Programozás, BMF, 2001 tavaszi félév

23. előadás (funkcionális programozás)

**KIRÁCS, NYOMKÖVETÉS**

Kirács, nyomkövetés 447

**Nyomkövetés: nodesa (akkumulátort használ)**

```
fun nodesa f =
 let (* nodes0 i (f, n) = n + a csomópontok száma f-ben;
 i a behúzásához használt függő
 nodes0 : string -> 'a tree * int -> int
 *)
 fun nodes0 i (N(a, t1, t2), n) =
 (print("\n" ^ i ^ "<""); printVal a : int; print "> "
 nodes0 (tab i) (printVal t1 before print("%\n" ^ (tab i))),
 nodes0 (tab i) (printVal t2 before print("%\n" ^
 (tab i)),
 printVal(n+1) before print " $"
)
 before print("#" ^ i)
)
 in
 | nodes0 i (L, n) = (* print("\n" ^ i); n *) n
 end;
 nodes0 "" (f, 0)
end;
```

Deklaratív Programozás, BMF, 2001 tavaszi félév

23. előadás (funkcionális programozás)

**nodes és nodesa alkalmazása hét csomópontból álló teljes fára**

```
f7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree

- nodes f7;
| - nodesa f7;
|
| <1> N(3, N(6, L, L), N(7, L, L)) * | <1> N(2, N(4, L, L), N(5, L, L)) %
| <3> N(7, L, L) * | N(3, N(6, L, L), N(7, L, L)) *
| <7> L * | L $
| L % | <3> N(6, L, L) %
| $ 1 # | N(7, L, L) *
| N(6, L, L) % | 2 $
| <6> L * | <7> L %
| L % | L *
| $ 1 # | 3 $ #
| $ 3 # | <6> L %
| N(2, N(4, L, L), N(5, L, L)) % | L *
| | 4 $ #
| | #
```

**● Felváltása a következő lapon.**

Deklaratív Programozás, BMF, 2001 tavaszi félév

23. előadás (funkcionális programozás)



**nodes és nodesa alkalmazása ... (folyt.)**

```

f7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree
(nodes f7)	(nodesa f7)
<2> N(5, L, L) *	
<5> L *	
L %	
$ 1 #	
N(4, L, L) %	
<4> L *	
L %	
$ 1 #	
$ 3 #	
$ 7 #	
-----	-----
> val it = 7 : int | > val it = 7 : int

```

Deklaratív Programozás, BMF, 2001 tavaszi félév

23. előadás (funkcionális programozás)

**Nyomonkövetés: deptha (akkumulátort használ)**

```

fun deptha f =
 let (* depth0 i (f, d) = d + az f fa mélysége; i a beháztáshoz használt figyelő
 depth0 : string -> 'a tree * int -> int *)
 fun depth0 i (N(a : int, t1, t2), d) =
 (print("\n" ~ i ~ "<""); printVal a : int; print "> ";
 printVal(Int.max(depth0 (tab i) (printVal t2 before print(" * \n" ~
 (tab i))),
 printVal(d+1) before print " $ "
),
 depth0 (tab i) (printVal t1 before print(" % \n" ~
 (tab i))),
 printVal(d+1) before print " & "
)
)
 before print("# \n" ~ i)
)
 | depth0 i (L, d) = (print("\n" ~ i); d);
 in depth0 "" (f, 0)
end;

```

Deklaratív Programozás, BMF, 2001 tavaszi félév

23. előadás (funkcionális programozás)

**Nyomonkövetés: depth (akkumulátort nem használ)**

```

fun depth f =
 let (* depth0 i f = az f fa mélysége; i a beháztáshoz használt figyelő
 depth0 : string -> 'a tree -> int
 *)
 fun depth0 i (N(a : int, t1, t2)) =
 (print("\n" ~ i ~ "<""); printVal a : int; print "> ";
 printVal(1 +
 Int.max(depth0 (tab i) (printVal t2 before print " *"),
 depth0 (tab i) (printVal t1 before print " %"))
)
)
 before print("# \n" ~ i))
 in
 | depth0 i L = (print("\n" ~ i); 0)
 end;
 in
 depth0 "" f
 end;

```

- **Megjegyzés:** Az itt alkalmazott nodes, nodesa, depth és deptha függvények nyomonkövetés nélküli változatát az előző előadásokon ismertettük.

Deklaratív Programozás, BMF, 2001 tavaszi félév

23. előadás (funkcionális programozás)

**depth és deptha alkalmazása hét csomópontból álló teljes fára**

```

f7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree
- depth f7: | - deptha f7: |
|-----|-----| |-----|-----|
<1> N(3, N(6, L, L), N(7, L, L)) * | | | <1> N(3, N(6, L, L), N(7, L, L)) * |
<3> N(7, L, L) * | | | 1 $ |
<7> L * | | | <3> N(7, L, L) * |
L % | | | 2 $ |
1 # | | | <7> L * |
N(6, L, L) % | | | L % |
<6> L * | | | 3 $ |
L % | | | 1 % |
1 # | | | 3 & |
2 # | | | 3 # |
N(2, N(4, L, L), N(5, L, L)) % | | | N(6, L, L) % |
|-----|-----| |-----|-----| |
| | | <6> L * | | | 2 & |
| | | 3 $ | | | 3 $ |
| | | 1 % | | | 3 & |
| | | 3 # | | | 3 # |
| | | N(2, N(4, L, L), N(5, L, L)) % | | | 1 & |
|-----|-----| |-----|-----|

```

- Folytatása a következő lapon.

Deklaratív Programozás, BMF, 2001 tavaszi félév

23. előadás (funkcionális programozás)

## depth és deptha alkalmazása hét csomópontból álló teljes fára

```
f7 = N(1, N(2, N(4, L, D), N(5, L, D)), N(3, N(6, L, D), N(7, L, D))) : int tree
```

```
(depth f7)	(deptha f7)
<2> N(5, L, D) *	<2> N(5, L, D) *
<5> L *	2 $
L %	<5> L *
1 #	3 $
N(4, L, D) %	L %
<4> L *	3 &
L %	3 #
1 #	N(4, L, D) %
2 #	2 &
3 #	<4> L *
3 $	
L %	
3 &	
3 #	
> val f7 = 3 : int | > val f7a = 3 : int
```

## Lista előállítás bináris fa elemeiből

- preorder, inorder és postorder *bináris fából listát* állít elő. A három függvény abban különbözik egymástól, hogy az egy csomópontból az ott tárolt értéket mikor veszik ki, és milyen sorrendben járják be a bal, ill. a jobb részfát.
  - preorder először az értéket veszi ki, majd bejárja a bal, és aztán a jobb részfát.
  - inorder először bejárja a bal részfát, majd kiveszi az értéket, és végül bejárja a jobb részfát.
  - postorder először bejárja a bal, majd a jobb részfát, és utoljára veszi ki az értéket.
- A következő megvalósítások egyszerűek, érthetőek, de nem elég hatékonyak a @ operátor használata miatt.

## BINÁRIS FÁK

## Lista előállítás bináris fa elemeiből (folyt.)

- Akkumulátor nem használó változatok
  - (\* preorder f = az f fa elemeinek preorder sorrendű listája  
preorder : 'a tree -> 'a list \*)  
fun preorder (N(v,t1,t2)) = v :: preorder t1 @ preorder t2  
| preorder L = [];
  - (\* inorder f = az f fa elemeinek inorder sorrendű listája  
inorder : 'a tree -> 'a list \*)  
fun inorder (N(v,t1,t2)) = inorder t1 @ (v :: inorder t2)  
| inorder L = [];
  - (\* postorder f = az f fa elemeinek postorder sorrendű listája  
postorder : 'a tree -> 'a list \*)  
fun postorder (N(v,t1,t2)) = postorder t1 @ postorder t2 @ [v]  
| postorder L = [];
- Az akkumulátort használó változatok nehezebben érthetőek, de *hatékonyabbak*.

## Lista előállítás a bináris fa elemeiből (folyt.)

- (\* preorder(f, vs) = az f fa elemeinek a vs lista elé fűzött, preorder sorrendű listája >>> rev postord |  

```
preord : 'a tree * 'a list -> 'a list *)
fun preord (N(v,t1,t2), vs) = v::preord(t1, preord(t2,vs))
 | preord (L, vs) = vs;
```
- (\* inord(f, vs) = az f fa elemeinek a vs lista elé fűzött, inorder sorrendű listája  

```
inord : 'a tree * 'a list -> 'a list *)
fun inord (N(v,t1,t2), vs) = inord(t1, v::inord(t2,vs))
 | inord (L, vs) = vs;
```
- (\* postord(f, vs) = az f fa elemeinek a vs lista elé fűzött, postorder sorrendű listája  

```
postord : 'a tree * 'a list -> 'a list *)
fun postord (N(v,t1,t2), vs) = postord(t1, postord(t2, v::vs))
 | postord (L, vs) = vs;
```

Deklaratív Programozás, BMF, 2001 tavaszi félév

23. előadás (funkcionális programozás)

## Bináris fa előállítás a lista elemeiből: take'ndrop

- Injunk take'ndrop néven olyan függvényt, amelynek egy xs listából és egy k egészről álló pár az argumentuma, és egy olyan pár az eredménye, amelynek első tagja a lista első k db eleme, második tagja pedig a lista többi eleme.  

```
(* take'ndrop(xs, k) = olyan pár, amelynek első tagja xs első k db
 eleme, második tagja pedig xs maradéka
 take'ndrop : 'a list * int -> 'a list * 'a list
 *)
fun take'ndrop (xs, k) =
 let fun td (xs, 0, ts) = (rev ts, xs)
 | td (x::xs, k, ts) = td(xs, k-1, x::ts)
 | td ([], _, ts) = (rev ts, [])
 in
 td(xs, k, [])
 end;
```
- take'ndrop felhasználása, nevezetesen az eredményül átdott pár miatt módosítani kell balpreorder felépítésén.

Deklaratív Programozás, BMF, 2001 tavaszi félév

23. előadás (funkcionális programozás)

## Bináris fa előállítás a lista elemeiből: balPreorder

- Listát *kiegyensúlyozott* (balanced) *bináris fa*vá alakítani a következő függvények: balPreorder, balInorder és balPostorder; a különbség közöttük most is a bejárás sorrendben van.
- (\* balPreorder xs = az xs lista elemeiből álló, preorder bejárású, kiegyensúlyozott fa  

```
balPreorder: 'a list -> 'a tree
 *)
fun balPreorder (x::xs) =
 let val k = length xs div 2
 in
 N(x, balPreorder(list.take(xs, k)),
 balPreorder(list.drop(xs, k)))
 end
 | balPreorder [] = L;
```
- A hatékonyságot kisebb mértékben rontja, hogy list.take és list.drop egymástól függetlenül *kétszer* mennek végig a lista első felén.

Deklaratív Programozás, BMF, 2001 tavaszi félév

23. előadás (funkcionális programozás)

## Bináris fa előállítás a lista elemeiből: balPreorder, újra

- Ez volt:  

```
fun balPreorder (x::xs) =
 let val k = length xs div 2
 in N(x, balPreorder(list.take(xs, k)), balPreorder(list.drop(xs, k)))
 end
 | balPreorder [] = L;
```
- Ez lett:  

```
(* balPreorder xs = az xs lista elemeiből álló, preorder bejárású, ...
 balPreorder: 'a list -> 'a tree *)
fun balPreorder (x::xs) =
 let val k = length xs div 2
 val (ts, ds) = take'ndrop(xs, k)
 in N(x, balPreorder ts, balPreorder ds)
 end
 | balPreorder [] = L;
```

Deklaratív Programozás, BMF, 2001 tavaszi félév

23. előadás (funkcionális programozás)

## Bináris fa előállítása lista elemeiből

- (\* ballInorder xs = az xs lista elemeiből álló, inorder bejárású, kiegyensúlyozott fa  
ballInorder: 'a list -> 'a tree  
\*)  

```
fun ballInorder (xxs as x::xs) =
 let val k = length xxs div 2
 val ys = List.drop(xxs, k)
 in N(nd ys, ballInorder(List.take(xxs, k)), ballInorder(tl ys))
 end
 | ballInorder [] = L;
```
- (\* balPostorder xs = az xs lista elemeiből álló, postorder bejárású, kiegyensúlyozott fa  
balPostorder: 'a list -> 'a tree  
\*)  

```
fun balPostorder xs = balPreorder(rev xs);
```
- ballInorder take'ndrop-pal való definiálását meghagyjuk gyakorló feladatnak.

Deklaratív programozás, BMF, 2001 tavaszi félév

23. előadás (funkcionális programozás)

## Elem törlése bináris fából (folyt.)

- A remove rendezetlen bináris fából törli az  $i$  értékű elem összes előfordulását.
- A join-nal egyesítjük a törlés hatására létrejövő két részfat: a bal részfat lebontja, és közben az elemeit egyesével berakja a jobb részfatba.  

```
(* join(b, j) = a b és a j fák egyesítésével létrehozott fa
 join : 'a tree * 'a tree -> 'a tree *)
fun join (N(v, lt, rt), tr) = N(v, join(lt, tr), tr)
 | join (L, tr) = tr;
```
- (\* remove(i, f) =  $i$  összes előfordulását törli  $f$ -ből  

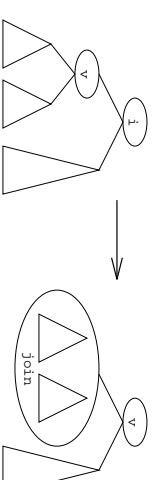
```
remove : 'a * 'a tree -> 'a tree *)
fun remove (i, N(v,lt,rt)) =
 if i<>v then N(v, remove(i,lt), remove(i,rt))
 else join(remove(i,lt), remove(i,rt))
 | remove (i, L) = L;
```

Deklaratív programozás, BMF, 2001 tavaszi félév

23. előadás (funkcionális programozás)

## Elem törlése bináris fából

- Adott értékű *elemet* rekurzív módszerrel *megkeresni* egyszerű feladat.
- *Új elemet beszúrni* sem nehéz: rekurzív módszerrel keressünk egy levelet, és ennek a helyére berakjuk az új értéket. Ha a fa rendezve van, ügyelnünk kell arra, hogy a rendezettség megmaradjon.
- Adott értékű *elemet* vagy *elemeket* rekurzív módszerrel *kitörölni* valamilyen nehezebb: ha a törlendő érték az éppen vizsgált részfa gyökerében van, a két részre széteső fa részfat *egyesíteni* kell, mintán a törlést a két részfat már végrehajtottuk.



- Megtehetjük, hogy előbb egyesítjük a két részfat, majd az eredményül kapott fából töröljük az adott értékű elemet.

Deklaratív programozás, BMF, 2001 tavaszi félév

23. előadás (funkcionális programozás)

## LISTÁK HASZNÁLATA

## A „jó” számok” előállítása SML-függvényvel

- „Jó” számok: keressük azokat a számokat, amelyek négyzete háromjegyű, és a szám fordítottjával kezdődnek (vö. Prolog-előadások).

```
(* jOSzamok i = azoknak az i és 100 közötti kétjegyű számoknak a listája,
 amelyek négyzete háromjegyű, és a szám fordítottjával kezdődnek
 jOSzamok : int -> int list
*)
fun jOSzamok i =
 if i < 100
 then if i * i div 10 = i mod 10 * 10 + i div 10
 then i :: jOSzamok (i+1)
 else jOSzamok (i+1)
 else [];
```

```
jOSzamok 10;
```

- Írjunk átkalnosabb megoldást: emeljük ki a szám jó voltának és a felső határ elérésének a vizsgálatát!

Deklaratív Programozás, BMF, 2001 tavaszi félév

24. előadás (funkcionális programozás)

Listák használata

467

## A „jó” számok” előállítása SML-függvényvel (folyt.)

- jOSzamok egy szokásos megvalósítása

```
(* jOSzamok lim f i = a (lim max)-ot és az f-et kielégítő i egészek
 listája, ahol (lim max) a felső határ elérését,
 f pedig i jó szám voltát vizsgálja
 jOSzamok : (int -> bool) -> (int -> bool) -> int -> int list
*)
fun jOSzamok lim f i =
 if lim i
 then if f i
 then i :: jOSzamok lim f (i+1)
 else jOSzamok lim f (i+1)
 else [];
```

```
jOSzamok (lim 100) jsz1 10;
jOSzamok (lim 300) jsz2 10;
```

Deklaratív Programozás, BMF, 2001 tavaszi félév

24. előadás (funkcionális programozás)

## A „jó” számok” előállítása SML-függvényvel (folyt.)

- A jsz és a lim segédfüggvények

```
(* jsz1 i = igaz, ha a kétjegyű i négyzete háromjegyű és a
 fordítottjával kezdődik
 jsz1 : int -> bool *)
fun jsz1 i = i * i div 10 = i mod 10 * 10 + i div 10;
```

```
(* jsz2 i = igaz, ha a háromjegyű i egyes és százazs helyiértéki
 jegyei egyenlők
 jsz2 : int -> bool *)
fun jsz2 i = i > 100 andalso
 (i mod 10, i div 100) = (i div 100, i mod 10);
```

```
(* lim x i = igaz, ha i kisebb x-nél
 lim : int -> int -> bool *)
fun lim x i = i < x;
```

Deklaratív Programozás, BMF, 2001 tavaszi félév

24. előadás (funkcionális programozás)

Listák használata

468

## A „jó” számok” előállítása SML-függvényvel (folyt.)

- jOSzamok jobbrekurzív változata

```
(* jOSzamok lim f i = a (lim max)-ot és az f-et kielégítő i egészek
 listája, ahol (lim max) a felső határ elérését,
 f pedig i jó szám voltát vizsgálja
 jOSzamok : (int -> bool) -> (int -> bool) -> int -> int list *)
fun jOSzamok lim f i =
 let fun jsz i zs =
 if lim i
 then jsz (i+1) (if f i then i :: zs else zs)
 else rev zs
 in
 jsz i []
 end;
```

```
jOSzamok (lim 100) jsz1 10;
jOSzamok (lim 300) jsz2 10;
```

Deklaratív Programozás, BMF, 2001 tavaszi félév

24. előadás (funkcionális programozás)

## Leállási feltétel kezelése

- Háromféle megoldást mutatunk be:
  - igazságérték – true, false – visszaadásával,
  - az 'a' option típus alkalmazásával,
  - kivételkezeléssel.
- Példa: „jű” számok előállítása
- A következő érték előállítására és a felső határ elérésének vizsgálatára *speciális* függvényt írunk, háromféle változatban:
 

```
kov x i jelzi, hogy i kisebb-e az x felső határnál, és ha igen, az i után
következő értéket adja eredményül, egyébként az eredmény tetszőleges.
```

## LEÁLLÁSI FELTÉTEL KEZELÉSE

Leállási feltétel kezelése 471

### A kov függvény háromféle változatban

- Igazságérték visszaadásával:

```
(* kov11 x i = (i+1, true), ha i < x (felső határ), egyébként (i, false)
 kov11 : int -> int -> int * bool *)
fun kov11 x i = if i < x then (i+1, true) else (i, false);
```

- int option alkalmazásával:

```
(* kov21 x i = SOME(i+1), ha i < x (felső határ), egyébként NONE
 kov21 : int -> int -> int option *)
fun kov21 x i = if i < x then SOME(i+1) else NONE;
```

- Kivételjelzéssel:

```
exception Limit;
(* kov31 x i = i+1, ha i < x (felső határ), egyébként a Limit kivétel
 kov31 : int -> int -> int *)
fun kov31 x i = if i < x then i+1 else raise Limit;
```

## Leállási feltétel kezelése igazságértékkel

- `kov : 'a -> 'a -> 'a * bool` (\* nzt = kov x \*)

```
(* findAll11 nzt f i = az f i összes megoldásának listája a felső határ elérését
 vizsgáló és a következő értéket eredményező nzt függvény segítségével
 findAll11 : ('a -> 'a * bool) -> ('a -> bool) -> 'a -> 'a list *)
fun findAll11 nzt f i =
 let fun fAll f z zs =
 let val (j, b) = nzt z
 in
 if b then fAll f j (if f z then z::zs else zs)
 else rev zs
 end
 in
 fAll f i []
 end;
 findAll11 (kov11 100) jsz1 10;
 findAll11 (kov11 300) jsz2 100;
```

## Leállási feltétel kezelése az 'a option típus alkalmazásával

```
● kov : 'a -> 'a option (* nxt = kov x *)

(* findA112 nrt f i = az f i összes megoldásának listája a felső határ elérését
 vizsgáló és a következő értéket eredményező nrt függvény segítségével
 findA112 : ('a -> 'a option) -> ('a -> bool) -> 'a -> 'a list
 *)
fun findA112 nrt f i =
 let fun fAll f z zs =
 case nrt z of
 SOME j => fAll f j (if f z then z::zs else zs)
 | NONE => rev zs
 in
 fAll f i []
 end;
 findA112 (kov21 100) jsz1 10;
findA112 (kov21 300) jsz2 100;
```

Deklaratív Programozás, BMF, 2001 tavaszi félév

24. előadás (funkcionális programozás)

## Leállási feltétel kezelése kivételkezeléssel

```
● kov : 'a -> 'a option (* nxt = kov x *)

(* findA113 nrt f i = az f i összes megoldásának listája a felső határ elérését
 vizsgáló és a következő értéket eredményező nrt függvény segítségével
 findA113 : ('a -> 'a) -> ('a -> bool) -> 'a -> 'a list
 *)
fun findA113 nrt f i =
 let fun fAll f z zs = fAll f (nrt z) (if f z then z::zs else zs)
 handle Limit => rev zs
 in
 fAll f i []
 end;
 findA113 (kov31 100) jsz1 10;
findA113 (kov31 300) jsz2 100;
```

Deklaratív Programozás, BMF, 2001 tavaszi félév

24. előadás (funkcionális programozás)

## Listák rendezése

- insort (beszűrő rendezés),
- quicksort (gyorsrendezés),
- tmsort (felülről lefelé haladó összefésülő rendezés),
- bmsort (alulról felfelé haladó összefésülő rendezés),
- smsort (simarendezés).

## LISTÁK RENDEZÉSE

Deklaratív Programozás, BMF, 2001 tavaszi félév

24. előadás (funkcionális programozás)

## Beszűrő rendezés

- Az `ins` segédfüggvény az `x` elemet a megfelelő helyre rakja be az `ys` listában:

```
(* ins (x, ys) = az x értékkel a <= reláció szerint bővített ys
 ins : real * real list -> real list
 PRE: ys a <= reláció szerint rendezett *)
fun ins (x, y::ys) = if x <= y then x::y::ys else y::ins(x, ys)
| ins (x : real, []) = [x];
```

- `insort`-tal rekurzívan rendezzük a lista maradékát; végrehajtási ideje  $O(n^2)$ :

```
(* insort f xs = az xs eleminek az f függvény segítségével
 rendezett listája
 insort : ('a * 'b list -> 'b list) -> 'a list -> 'b list *)
fun insort f (x::xs) = f(x, insort f xs)
| insort _ [] = [];
```

- Példa `insort` alkalmazására:

```
insort ins [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0];
```

## Beszűrő rendezés, generikus változat

- Az `ins` függvényt generikussá tesszük:

```
(* ins cmp (x, ys) = az x értékkel a cmp reláció szerint bővített ys
 ins : ('a * 'a -> bool) -> 'a * 'a list -> 'a list
 PRE: ys a cmp reláció szerint rendezve van *)
fun ins cmp (x, ys) =
 let fun ins0 (y::ys) = if cmp(x, y) then x::y::ys else y::ins0 y
 | ins0 [] = [x]
 in ins0 ys
 end;
```

- Ezzel `insort` egy újabb változata:

```
(* insort cmp xs = az xs eleminek a cmp reláció szerint rendezett listája
 insort : ('a * 'a -> bool) -> 'a list -> 'a list *)
fun insort cmp (x::xs) = ins cmp (x, insort cmp xs)
| insort _ [] = [];
```

## LISTÁK RENDEZÉSE

## Beszűrő rendezés, generikus változat (folyt.)

- `insort` eddigi változatai előbb elemekre szedik szét a rendezendő listát, majd hátulról visszatelél haladva, rendezés közben építik fel az újat.

- A jobbrekurzíót és akkumulátort használó változatnak (`insort2`) kisebb veremre van szüksége, mivel a listáról leválasztott elemeket balról jobbra haladva azonnal berakja a helyükre az eredménylistában. (A két megoldás futási idejét később összehasonlíjuk).

```
fun insort2 cmp xs =
 (* sort xs zs = az xs már feldolgozott eleminek a cmp
 reláció szerint rendezett listája zs
 let fun sort (x::xs) zs = sort xs (ins cmp (x, zs))
 | sort [] zs = zs
 in
 sort xs []
 end;
```



## Beszűrő rendezés foldr-rel és foldl-lal

- A második argumentumát akkumulátorként használó foldl kisebb vermet használ foldr-nél, ezért insortL hosszabb listákat tud rendezni:

```
fun insortR cmp = foldr (ins cmp) [] ;
fun insortL cmp = foldl (ins cmp) [] ;
```

- Példák insort-lal és insort2-vel:

```
insort op<= [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0] ;
insort2 op>= [4, 4, 5, 1, 0, 8] ;
insort op< (explode "qwerty") ;
```

- Példák foldr és foldl felhasználásával:

```
fun insortRi cmp = foldr (ins cmp) [] ;
fun insortLi cmp = foldl (ins cmp) ([] : real list) ;
insortRi op>= [4, 4, 5, 1, 0, 8] ;
insortLi op>= [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0] ;
```

Deklaratív Programozás, BME, 2001 tavaszi félév

25. előadás (funkcionális programozás)

Listák rendezése 483

## A futási idők összehasonlítása (folyt.)

- A futási időt az alábbi függvényvel mérjük meg:

```
fun futido (sort, sortFn) (cmp, cmpFn) (xs, kind) =
 let val starttime = Timer.startCPUTimer()
 val zs = sort cmp xs
 val fusr=tim,...} = Timer.checkCPUTimer starttime
 in
 "Int sort with " ^ sortFn ^ ", " ^ cmpFn ^
 ", length = " ^ Int.toString(length xs) ^ " (" ^
 kind ^ "), time = " ^ Time.fmt 2 tim ^ " sec\n"
 end ;
```

```
val t1N = futido (insort, "insort") (op>=, "op>=") (xs2000N, "increasing") ;
val t2N = futido (insort2, "insort2") (op>=, "op>=") (xs2000N, "increasing") ;
val t1R = futido (insort, "insort") (op>=, "op>=") (xs2000R, "random") ;
val t2R = futido (insort2, "insort2") (op>=, "op>=") (xs2000R, "random") ;
```

Deklaratív Programozás, BME, 2001 tavaszi félév

25. előadás (funkcionális programozás)

## A futási idők összehasonlítása

- 2000 elemet tartalmazó, véletlenszerűen előállított, illetve eredetileg éppen fordított sorrendű listák rendezéséhez szükséges futási időt mérünk.

- Véletlen elosztású egészlistát állít elő a Random könyvtárbeli rangelist függvény:

```
val xs2000R = Random.rangelist (1, 100000) (2000, Random.newgen()) ;
```

- Növekvő sorrendű egészlistát állít elő a -- operátor:

```
infix -- ;
fun fm -- to =
 let
 fun upto to zs = if to < fm then zs else upto (to-1) (to::zs)
 in
 upto to []
 end ;
val xs2000N = 1 -- 2000 ;
```

Deklaratív Programozás, BME, 2001 tavaszi félév

25. előadás (funkcionális programozás)

Listák rendezése 484

## A futási idők összehasonlítása (folyt.)

- A 2000 elemű, fordított sorrendű lista rendezése az akkumulátort nem használó insort-változatokkal több mint 5 s-ig, az akkumulátort használó változatokkal csak 0.01 s-ig tart (Inux, 233 MHz-es Pentium).

```
Int sort with insort, op>=, length = 2000 (increasing), time = 5.18 sec
Int sort with insort2, op>=, length = 2000 (increasing), time = 0.01 sec
Int sort with insortRi, op>=, length = 2000 (increasing), time = 5.14 sec
Int sort with insortLi, op>=, length = 2000 (increasing), time = 0.01 sec
```

- Eltűnik a különbség, ha ugyanolyan hosszú, de véletlenszerűen előállított listákat rendezünk.

```
Int sort with insort, op>=, length = 2000 (random), time = 2.39 sec
Int sort with insort2, op>=, length = 2000 (random), time = 2.26 sec
Int sort with insortRi, op>=, length = 2000 (random), time = 2.40 sec
Int sort with insortLi, op>=, length = 2000 (random), time = 2.24 sec
```

Deklaratív Programozás, BME, 2001 tavaszi félév

25. előadás (funkcionális programozás)

## Gyorsrendezés, akkumulátor használata nélkül

```
(* quicksort1 cmp xs = az xs elemeinek cmp szerint rendezett listája
 quicksort1 : ('a * 'a -> order) -> 'a list -> 'a list *)
fun quicksort1 cmp xs =
 let (* qs : 'a list -> 'a list
 qs ys = az ys elemeinek cmp szerint rendezett listája *)
 fun qs (m::ys) =
 let (* partition : 'a list * 'a list * 'a list -> 'a list
 partition (xs, ls, rs) = ... *)
 fun partition (x::xs, ls, rs) =
 if cmp(x, m) = LESS then partition(xs, x::ls, rs)
 else partition(xs, ls, x::rs)
 | partition ([], ls, rs) = qs ls @ (m::qs rs)
 in
 partition ([], ls, rs)
 end
 | qs [] = []
 in
 qs xs
 end;
 end;
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

25. előadás (funkcionális programozás)

## A futási idők összehasonlítása

```
val t1 = futido (insert2, "insert2") (op>=, "op>=") (xs2000R, "random");
(* ~ 2 M összehasonlítás! *)
val t3 = futido (quicksort2, "quicksort2")
 (Int.compare, "Int.compare") (xs2000R, "random");
val t4 = futido (listsort.sort, "listsort.sort")
 (Int.compare, "Int.compare") (xs2000R, "random");
(* ~ 300 F összehasonlítás *)

Int sort with insert2, op>=, length = 2000 (random), time = 2.30 sec

Int sort with quicksort1, Int.compare, length = 20000 (random), time = 2.18 sec
Int sort with quicksort2, Int.compare, length = 20000 (random), time = 1.72 sec
Int sort with listsort.sort, Int.compare, length = 20000 (random), time = 1.76 sec

Int sort with quicksort2, Int.compare, length = 200000 (random), time = 27.13 sec
Int sort with quicksort1, Int.compare, length = 200000 (random), time = 32.59 sec

val t7 = futido (listsort.sort, "listsort.sort") (Int.compare, "Int.compare")
 (Random.randgen1ist (1, 100000) (200000, Random.newgen ()), "random");
! Uncaught exception:
! Out_of_memory
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

25. előadás (funkcionális programozás)

## Gyorsrendezés, akkumulátor használatával

```
(* quicksort2 cmp xs = az xs elemeinek cmp szerint rendezett listája
 quicksort2 : ('a * 'a -> order) -> 'a list -> 'a list *)
fun quicksort2 cmp xs =
 let (* qs : 'a list -> 'a list
 qs ys = az ys elemeinek cmp szerint rendezett listája *)
 fun qs (m::ys) zs =
 let (* partition : 'a list * a' list * 'a list -> 'a list
 partition (xs, ls, rs) = ... *)
 fun partition (x::xs, ls, rs) =
 if cmp(x, m) = LESS then partition(xs, x::ls, rs)
 else partition(xs, ls, x::rs)
 | partition ([], ls, rs) = qs ls (m :: qs rs zs)
 in
 partition ([], ls, rs)
 end
 | qs [] zs = zs
 in
 qs xs []
 end;
 end;
```

Deklaratív programozás, BMEF, 2001 tavaszi félév

25. előadás (funkcionális programozás)

## Összefésülő rendezések

- Az összefésülő rendezéshez kell egy olyan függvény, amely két listát növekvő sorrendben egyesít:
 

```
(* merge(xs, ys) = xs és ys elemeinek <= szerint egyesített listája
 merge : int list * int list -> int list
 *)
fun merge (xxs as x::xs, yys as y::ys) =
 if x <= y
 then x::merge(xs, yys)
 else y::merge(xxs, ys)
 | merge ([], ys) = ys
 | merge (xs, []) = xs;
```
- Hatékonyágronlást okoz, hogy a részeredményeket a veremben tároljuk. Iteratív megoldás esetén meg kell fordítani az eredménylistát.

Deklaratív programozás, BMEF, 2001 tavaszi félév

25. előadás (funkcionális programozás)

## Főlülről lefelé haladó összefésülő rendezés

- A fölülről lefelé haladó összefésülő rendezés (*top-down merge sort*) akkor hatékony, ha közel azonos hosszúságú az a két lista, amelyekre a rendezendő listát szétszedjük.

```
(* tmsort xs = az xs elemeinek a <= reláció szerint rendezett listájá
tmsort : int list -> int list
*)
fun tmsort xs = let val h = length xs
 val k = h div 2
 in
 if h > 1
 then merge(tmsort(List.take(xs, k)),
 tmsort(List.drop(xs, k)))
 else xs
 end;
```

- A legrösszabb esetben  $O(n \cdot \log n)$  lépésre van szükség.

Deklaratív programozás, BMF, 2001 tavaszi félév

25. előadás (funkcionális programozás)

Listák rendezése 491

## Alulról fölfelé haladó összefésülő rendezés (folyt.)

- bmsort a sorting segédfüggvényt használja, amelynek
  - első argumentuma a rendezendő lista,
  - második argumentuma a már rendezett részhlistákat gyűjti,
  - harmadik argumentuma az adott lépésben összefuttatandó elem sorszáma.

```
(* bmsort xs = az xs elemeinek a <= reláció szerint rendezett listája
bmsort : int list -> int list
*)
fun bmsort xs = sorting(xs, [], 0);
```

Deklaratív programozás, BMF, 2001 tavaszi félév

25. előadás (funkcionális programozás)

## Alulról fölfelé haladó összefésülő rendezés

- Az alulról fölfelé haladó összefésülő rendezés (*bottom-up merge sort*) legegyszerűbb változata az eredeti  $k$  hosszúságú listák  $k$  darab egyelemű listára bontja, majd a szomszédos listákat összehatja, így 2, 4, 8, 16 stb. elemű listákat állít elő.

- R. O'Keefe algoritmus (1982) lépésről lépésre futtatja össze az egyforma hosszú részhlistákat, de csak az utolsó lépésben rendezzi az összet. Az alábbi példában az összefuttatott részhlistákat *egymás mellé írással* jelöljük:

```
AB C D E F G H I J K
AB CD E F G H I J K
ABCD E F G H I J K
ABCD EF GH I J K
ABCD EFGH I J K
ABCD EFGH I J K
ABCDEFGHI I J K
...

```

Deklaratív programozás, BMF, 2001 tavaszi félév

25. előadás (funkcionális programozás)

Listák rendezése 492

## Alulról fölfelé haladó összefésülő rendezés (folyt.)

- Ha a rendezendő lista ( $xs$ ) még nem foglyott el, soron következő eleméből sorting egyelemű listát  $[x]$  képez, és ezt a már rendezett részhlisták listája ( $lss$ ) elé fűzve meghívja a mergepairs segédfüggvényt. mergepairs az argumentumként átadott lista két azonos hosszúságú bal oldali részhlistáját fűzi egybe, feltéve persze, hogy vannak ilyenek.  $k$  az éppen átadott elem sorszáma. Ha a rendezendő lista kiürült, sorting a kétszintű lista egyetlen elemét, a rendezett listát adja eredményül.

```
(* sorting(xs, lss, k) = a még rendezetlen xs lista elemeit berakja
sorting : int list * int list * int -> int list
PRE: k >= 0
*)
fun sorting (x::xs, lss, k) = sorting(xs, mergepairs([x]::lss, k+1), k+1)
 | sorting ([], lss, k) = hd(mergepairs(lss, 0));
```

Deklaratív programozás, BMF, 2001 tavaszi félév

25. előadás (funkcionális programozás)

## Alhírói föléle haladó összefésülő rendezés (folyt.)

- mergepairs egyetlen listában gyűjti a már összeruttatott részhistákat. Az éppen átadott elem  $k$  sorszámából dönti el, hogy mit kell csinálnia a következő részhistával.

```
(* mergepairs(l1ss, n)= az n elemet tartalmazó, már rendezett l1ss lista
 első két részhistáját, ha egyforma a hosszuk, összeruttatja
 mergepairs : int list list -> int list list
 PRE: n >= 0
*)
fun mergepairs (l1ss as l1::l2::lss, n) = (* legalább két elemű a lista *)
 if n mod 2 = 1 then l1ss
 | mergepairs (lss, _) = lss (* egy elemű a lista *)
```

- Ha  $n$  páratlan, mergepairs a listát változtatás nélkül adja vissza, ha páros, akkor az l1ss lista elején álló két, egyforma hosszú listát egyetlen rendezett listává futtatja össze.  $n=0$ -ra mergepairs az összes listák listáját olyan listává futtatja össze, amelynek egyetlen eleme maga is lista.

Deklaratív Programozás, BME, 2001 tavaszi félév

25. előadás (funkcionális programozás)

Listák rendezése 465

## Alhírói föléle haladó összefésülő rendezés (folyt.)

- A következő táblázatos elrendezés
    - mergepairs mindkét argumentumát,
    - a rekurzív sorting hívás itt  $j$ -vel jelölt 3. argumentumát,  $k+1$ -et, és
    - bináris számként  $k$ -t mutatja lépésről lépésre.
  - A sorting függvény hívja mergepairs-t azokban a sorokban, amelyekben a  $j$  új értéket vesz föl, a többi helyen mergepairs hívása rekurzív.
  - Ne feledjük, hogy mergepairs-nek listák listája az első argumentuma!
  - A táblázat utolsó oszlopa a vonatkozó magyarázatra hivatkozik.
  - Vegyük észre, hogy kapcsolat van az l1ss első eleme utáni listaelemek hossza és a  $k$  bitjei között! Ha  $k$  valamelyik bitje 1, akkor (balról jobbra haladva) az l1ss megfelelő listaelemének a hossza az adott bit helyértékével egyenlő. A 0 értékű biteknek megfelelő listaelemek „hiányoznak” l1ss-ből.
- ```
fun sorting (x::xs, lss, k) = sorting(xs, mergepairs([k]::lss, k+1), k+1)
  | sorting ([], lss, k) = hd(mergepairs(lss, 0));
```

Deklaratív Programozás, BME, 2001 tavaszi félév

25. előadás (funkcionális programozás)

Alhírói föléle haladó összefésülő rendezés (folyt.)

- A legrosszabb esetben $O(n \cdot \log n)$ lépésre van szükség.
- A függvények működését egy példán is bemutattuk. A kezdőhívás legyen


```
hmsort [1,2,3,4,5,6,7,8,9] ----> sorting ([1,2,3,4,5,6,7,8,9], [], 0)
```
- Amíg sorting első argumentuma a nem üres ($x::xs$) lista, sorting saját magát hívja meg. A rekurzív hívás
 - első argumentuma a lépésenként egyre rövidülő xs lista,
 - második argumentuma a mergepairs($[k]::lss, k+1$) függvényalkalmazás eredménye, ahol kezdetben $lss = []$,
 - harmadik argumentuma ($k+1$) a már feldolgozott listaelemek száma.

```
fun sorting (x::xs, lss, k) = sorting(xs, mergepairs([k]::lss, k+1), k+1)
  | sorting ([], lss, k) = hd(mergepairs(lss, 0));
```

Deklaratív Programozás, BME, 2001 tavaszi félév

25. előadás (funkcionális programozás)

Listák rendezése 466

Alhírói föléle haladó összefésülő rendezés (folyt.)

l1ss	n	j	k	fun sorting (x::xs, lss, k) = sorting(xs, mergepairs([k]::lss, k+1), k+1) sorting ([], lss, k) = hd(mergepairs(lss, 0));		
[1,1]	1	1	0	m1		
[2], [1]	2	2	1	m2		
[1, 2]	1	1	3	m3		
[3], [1, 2]	3	3	10	m3		
[4], [3], [1, 2]	4	4	11	m2		
[3, 4], [1, 2]	2	2	2	m2		
[1, 2, 3, 4]	1	1	5	m3		
[5], [1, 2, 3, 4]	5	5	100	m3		
[6], [5], [1, 2, 3, 4]	6	6	6	104	m2	
[5, 6], [1, 2, 3, 4]	3	3	7	110	m3	
[7], [5, 6], [1, 2, 3, 4]	7	7	7	110	m3	
[8], [7], [5, 6], [1, 2, 3, 4]	8	8	8	111	m2	
[7, 8], [5, 6], [1, 2, 3, 4]	4	4	2	m2		
[5, 6, 7, 8], [1, 2, 3, 4]	2	2	1	1	m2	
[1, 2, 3, 4, 5, 6, 7, 8]	1	1	9	9	1000	m3
[9], [1, 2, 3, 4, 5, 6, 7, 8]	0	0	0	0	m4	
[1, 2, 3, 4, 5, 6, 7, 8, 9]						

m1: Az argumentumként átadott listának egyetlen eleme van (maga is lista), ezért az argumentumot mergepairs második kléza változtatás nélkül visszaadja az öt hívó sorting-nak.

m2: n páros, ez azt jelzi, hogy az argumentumként átadott lista első két eleme nem egyforma hosszú, ezért az argumentumot mergepairs első kléza változtatás nélkül visszaadja az öt hívó sorting-nak.

m3: n=0, az összes listák listáját olyan listává kell összeruttatni, amelynek egyetlen lista az eleme.

Deklaratív Programozás, BME, 2001 tavaszi félév

25. előadás (funkcionális programozás)

Simarendezés

- Az applikatív simarendezés (*smooth sort*) algoritmus $O(k \cdot kefe)$ alulról felfelé haladó rendezéséhez hasonló, de nem egyetlen listákat, hanem növekvő *futamokat* állít elő.
 - Ha a futamok száma n -tól független, azaz a lista majdnem rendezve van, akkor az algoritmus végrehajtási ideje $O(n)$, és a legrosszabb esetben is legfeljebb csak $O(n \cdot \log n)$.
- ```
(* nextrun : int list * int list -> int list * int list
 nextrun (run, xs) = ... *)
fun nextrun (run, x::xs) =
 if x < hd run then (rev run, x::xs) else nextrun(x::run, xs)
 | nextrun (run, []) = (rev run, []);
```
- nextrun eredménye egy pár, ennek
    - első tagja a futam (egy növekvő számsorozat),
    - a második tagja pedig a rendezendő lista maradéka.

Deklaratív Programozás, BMEF, 2001 tavaszi félév

25. előadás (funkcionális programozás)

Listák rendezése 488

## A futási idők összehasonlítása

```
fun futIdo2 (sort, sortFn) (xs, kind) =
 let val starttime = Timer.startCPUTimer()
 val zs = sort xs
 val {usr=tim,...} = Timer.checkCPUTimer starttime
 in "Int sort with " ^ sortFn ^ ", length = " ^ Int.toString(length xs) ^
 " (" ^ kind ^ ")", time = " ^ Time.fmt 2 tim ^ " sec\n"
 end;

val t101 = futIdo2 (msort, "msort")
 (Random.rangelist (1, 100000) (100000, Random.newGen ())), "Random");
val t102 = futIdo2 (bmsort, "bmsort")
 (Random.rangelist (1, 100000) (100000, Random.newGen ())), "Random");
val t103 = futIdo2 (msort, "msort")
 (Random.rangelist (1, 100000) (100000, Random.newGen ())), "Random");

Int.sort with msort, length = 100000 (Random), time = 10.96 sec
Int.sort with bmsort, length = 100000 (Random), time = 7.69 sec
Int.sort with msort, length = 100000 (Random), time = 7.70 sec
Int.sort with quicksort2, Int.compare, length = 100000 (Random), time = 11.98 sec
Int.sort with List.sort.sort, Int.compare, length = 100000 (Random), time = 14.17 sec
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

25. előadás (funkcionális programozás)

## Simarendezés (folyt.)

- A futam csökkentő sorrendben bővül, kilépéskor a futamot meg kell fordítani. `msorting` a futamokat ismételtlen előállítja és összefuttatja:
 

```
(* msorting : int list * int list list * int -> int list
 msorting (xs, lss, k) = ... *)
fun msorting (x::xs, lss, k) =
 let val (run, tail) = nextrun([k], xs)
 in
 msorting(tail, mergepairs(run::lss, k+1), k+1)
 end
 | msorting ([], lss, k) = hd(mergepairs(lss, 0));
```
- `(* msort : int list -> int list`  
`msort xs = az xs elemeinek a <= reláció szerint rendezett listája *)`  
`fun msort xs = msorting(xs, [], 0);`
- A simarendezés egy változata sort néven megtalálható a `List.sort` könyvtárban.

Deklaratív Programozás, BMEF, 2001 tavaszi félév

25. előadás (funkcionális programozás)

## LISTÁK HASZNÁLATA

***n* vezér a sakktáblán**

- Hányféleképpen rakható *n* vezér a sakktáblára úgy, hogy ne üssék egymást?

A vezéreket tartalmazó mezők sorának számát A sorvektort (egy egyre bővülő) listával az egyes oszlopokon belüli egy *n* hosszú sorvektor valósítjuk meg. Egy listához balról könnyű új adott oszlophoz rendelt mezőjébe írni  $s < n$  elemeket fűzni, a táblát és a vezérek helyzetét száma adja meg. Példa  $n = 4$  esetén: létre listát hosszengélye mentén tikr-özzük.

```

+---+---+---+---+
| | | | |
+---+---+---+---+
0 -> n-1
+---+---+---+---+
0 | | | | |
+---+---+---+---+
+---+---+---+---+
V | | | | |
+---+---+---+---+
n-1 | | | | |
+---+---+---+---+

```

```

...+---+---+---+---+
| | | | |
...+---+---+---+---+
n-1 <--- 0
...+---+---+---+---+
0 | | | | |
...+---+---+---+---+
+---+---+---+---+
V | | | | |
...+---+---+---+---+
n-1 | | | | |
...+---+---+---+---+

```

Deklaratív programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

***n* vezér a sakktáblán (folyt.)**

- Ha a 2-es oszlopba és az  $s=1$ -es sorba akarjuk lerakni az új vezért, akkor az *x*-szel jelölt mezőket kell megvizsgálnunk. Az eddig létrehozott listának (sorvektoroknak) két eleme van, ahol a lista fejének az indexe 0. A listafej értéke nem lehet  $s-1$ , sem  $s+1$ . A lista rekurzív algoritmussal dolgozható fel.

```

...+---+---+---+---+
| | | | |
...+---+---+---+---+
s | | | | |
...+---+---+---+---+
n-1 <--- 0
...+---+---+---+---+
0 | | | | |
...+---+---+---+---+
+---+---+---+---+
V | | | | |
...+---+---+---+---+
n-1 | | | | |
...+---+---+---+---+

```

Deklaratív programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

***n* vezér a sakktáblán (folyt.)**

- Azt, hogy az új vezért ütt-e a már táblára rakott másik vezér, a sorvektor vizsgálataival dönthetjük el, amely tehát azt adja meg, hogy a listaelemek indexe által meghatározott oszlopban és a listaelem értéke által meghatározott sorban vezér van.

- Az új vezér sorának száma, azaz az új listaelem értéke nem fordulhat elő a lista már felépített részében.
- Az új vezér átlós irányban sem lehet egy vonalban más vezérral a táblán. Ez azt jelenti, hogy ha a sorvektort jelentő lista elejére az *s* sorindexet akarjuk rakni, akkor az *i*-edik elemének az értéke, ha van ilyen eleme, nem lehet  $s-(i+1)$ , ill.  $s+(i+1)$ .
- A következő példa segít megvilágítani az esetet.

Deklaratív programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

***n* vezér a sakktáblán (folyt.)**

- „Útésben van”-vizsgálat

```

(* utesbenVan : int list -> bool
 utesbenVan zs = igaz, ha a (hd zs) vezér nincs útésben
 egyetlen (tl zs)-beli vezérrrel sem
*)
fun utesbenVan [] = false
 | utesbenVan (z::zs) =
 let fun uv _ _ [] = false
 | uv s1 s2 (r::rs) =
 z = r orlse s1 = r orlse s2 = r orlse
 uv (s1-1) (s2+1) rs
 in
 uv (z-1) (z+1) zs
 end;

```

Deklaratív programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

***n* vezér a sakktáblán (folyt.)**

- Egy megoldás előállítása

```
exception Zsakutca;

(* vezerek0 : int -> int list
 vezerek0 n = a feladvány egy megoldása n vezér esetén
 *)
fun vezerek0 n =
 let
 fun vez0 z zs =
 if z = 0 andalso utesbenVan zs orelse z = n then
 raise Zsakutca
 else if length zs = n then rev zs
 else vez0 0 (z::zs) handle Zsakutca => vez0 (z+1) zs
 in
 vez0 0 []
 end;
 end;
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Listák használata 507

***n* vezér a sakktáblán (folyt.)**

- Több megoldás előállítása listák listájával

```
fun vezerek n =
 let fun vez0 z zs =
 if z = 0 andalso utesbenVan zs orelse z = n then []
 else if length zs = n then [rev zs]
 else vez0 0 (z::zs) @ vez0 (z+1) zs
 in vez0 0 [] end;
 in vez0 0 [] [] end;
```

• Több megoldás előállítása listák listájával, akkumulátor alkalmazásával

```
fun vezerek n =
 let fun vez0 z zs ws =
 if z = 0 andalso utesbenVan zs orelse z = n then ws
 else if length zs = n then rev zs :: ws
 else vez0 0 (z::zs) (vez0 (z+1) zs ws)
 in vez0 0 [] [] end;
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

***n* vezér a sakktáblán (folyt.)**

- Több megoldás előállítása visszalépéssel

```
(* vezerek : int -> int list list
 vezerek n = a feladvány összes megoldásának listája n vezér esetén
 *)
fun vezerek n =
 let
 fun vez0 z zs =
 if z = 0 andalso utesbenVan zs orelse z = n
 then raise Zsakutca
 else if length zs = n then [rev zs]
 else (vez0 0 (z::zs) handle Zsakutca => []) @
 (vez0 (z+1) zs handle Zsakutca => [])
 in
 vez0 0 []
 end;
 end;
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

## BINÁRIS FÁK

## Bináris keresőfák

- Rendszerint adott kulcsú elemet keresünk, ehhez értékeket kell összehasonlítanunk egymással: a keresett kulcsnak *egyenlőségi típusúnak* kell lennie.
  - A példákban a string típust használjuk.
  - A függvények *kivételek* jeleznek, ha a keresett kulcsú elem nincs a keresőfában.
- ```
exception Bsearch of string;
```
- Szébb lenne, ha *generikus függvényeket* íránk; ezt gyakorlati feladatnak hagyjuk.

Deklaratív programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Bináris fák 511

Bináris keresőfák: binisert

A binisert függvény egy új kulcsú elemet rak be egy rendezett bináris fába, ha még nincs benne:

```
(* binisert(f, (b,y)) = az új (b,y) kulcs-érték párral bővített f fa
   binisert : (string * 'a) tree * (string * 'a) -> (string * 'a) tree *)
fun binisert (N((a,x), t1, t2), (b,y)) =
  if b < a
  then N((a,x), binisert(t1, (b,y)), t2)
  else if a < b
  then N((a,x), t1, binisert(t2, (b,y)))
  else (* a=b *) raise Bsearch("INSERT: " ^ b)
| binisert (L, (b,y)) = N((b,y), L, L);
```

Deklaratív programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Bináris keresőfák: blockup

- A blockup függvény adott kulcshoz tartozó értéket ad vissza egy rendezett bináris fából:

```
(* blockup(f, b) = az f fában a b kulcshoz tartozó érték
   blockup : (string * 'a) tree * string -> 'a *)
fun blockup (N((a,x), t1, t2), b) =
  if b < a
  then blockup(t1,b)
  else if a < b
  then blockup(t2, b)
  else x
| blockup (L, b) = raise Bsearch("LOOKUP: " ^ b);
```

Deklaratív programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Bináris fák 512

Bináris keresőfák: bupdate

- A bupdate függvény meglévő kulcsú elembe új értéket ír be egy rendezett bináris fában:

```
(* bupdate(f, (b,y)) = az f fa, a b kulcshoz tartozó érték helyén az y értékkel
   bupdate : (string * 'a) tree * (string * 'a) -> (string * 'a) tree *)
fun bupdate (N((a,x), t1, t2), (b,y)) =
  if b < a
  then N((a,x), bupdate(t1, (b,y)), t2)
  else if a < b
  then N((a,x), t1, bupdate(t2, (b,y)))
  else (* a=b *) N((b,y), t1, t2)
| bupdate (L, (b,y)) = raise Bsearch("UPDATE: " ^ b);
```

Deklaratív programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Lusta lista

- Olyan lista, amelynek a farka függvény, ezáltal késleltetjük a kiértékelését.
- Ily módon *végtelen listákkal* hozhatunk létre.
- A lusta listának hátrányai, veszélyei is vannak, pl.
 - egy lusta lista bármely részét megjeleníthetjük, de sohasem az egészet;
 - két lusta lista elemeiből páronként képezhetünk egy harmadikat, de nem számíthatjuk ki egy lusta lista elemének az összegét, nem kereshetjük meg benne a legkisebbet, nem fordíthatjuk meg az elemek sorrendjét;
 - úgy kell rekurziót definiálnunk, hogy nincs alapeset;
 - egy program befejeződése helyett csak azt igazolhatjuk, hogy az eredmény tetszőleges véges része véges idő alatt előáll.
- A lusta listát sorozatnak (*sequence*) nevezzük, és a seq típusoperátort használjuk a létrehozására.


```
datatype 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

Deklaratív programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Listák rendszése 515

Lusta lista (folyt.)

- Egy sorozat fejét adja eredményül a head függvény; abortál, ha üres sorozatra alkalmazzuk.


```
(* head : 'a seq -> 'a *)
fun head (Cons(x, _)) = x;
```
- Egy sorozat farkát adja eredményül a tail függvény; abortál, ha üres sorozatra alkalmazzák.


```
(* tail : 'a seq -> 'a seq *)
fun tail (Cons(_, xf)) = xf();
```

A sorozat farka unit -> 'a seq típusú *függvény*, erre illesztjük az xf mintát tail fejében; tail törzseben xf-et a () argumentumra kell alkalmazni.

Deklaratív programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Listák rendszése 516

Lusta lista (folyt.)

- consq(x, xq) x-et berakja az xq sorozatba:


```
(* consq : 'a * 'a seq -> 'a seq *)
fun consq (x, xq) = Cons(x, fn () => xq);
```
- Ha a consq függvényt alkalmazzuk, mondjuk, az (x, E) argumentumra, az SML a consq(x, E) kifejezést *nem lustán* értékeli ki, hiszen alapvetően möhő kiértékelést.
 - Ha E kiértékelésének eredményét xq-val jelöljük, akkor consq(x, E) kiértékelése a fenti definíció szerint Cons(x, fn () => xq)-t eredményez.
 - A consq-beli fn () => xq függvény nem késlelteti a fark (a példában E) kiértékelését consq alkalmazásakor.
 - A lusta kiértékelés érdekében a híváskor is a Cons(x, fn () => E) alakot kell használnunk, consq(x,E) nem jó.
 - Az explicit fn () => E késlelteti a kiértékelést, és ezzel *szükség szerinti hivatkozást* valósít meg.

Deklaratív programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Lusta lista (folyt.)

- Példaként a korábban megismert `from` és `take` függvények lusta változatait mutatjuk be.
- A `fromq k` sorozat egészek k -tól induló végtelen sorozata.


```
(* fromq : int -> int seq *)
fun fromq k = Cons(k, fn () => fromq(k+1));
```
- `takeq(xq, n)` az `xq` sorozat első n eleméből képzett listát adja vissza:


```
(* takeq : 'a seq * int -> 'a list *)
fun takeq (xq, 0) = []
  | takeq (Cons(x, xf), n) = x :: takeq(xf(), n-1)
  | takeq (Nil, n) = [];
```
- Az '`a`' seq típus nem egészen lusta kiértékelési: egy nemüres sorozat fejét a **rendszer mindig feldolgozza**.

Deklaratív programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Listák rendeltése 518

Egyszerű függvények lusta listákra (folyt.)

- Az `append` függvény addig nem nyúl `yc`-hoz, amíg `xq` ki nem ürül – vagyis csak akkor nyúl hozzá, ha `xq` véges. Véges sorozatot `consq`-val készíthetünk.


```
(* appendq : 'a seq * 'a seq -> 'a seq *)
fun appendq (Cons(x, xf), yq) = Cons(x, fn () => appendq(xf(), yq))
  | appendq (Nil, yq) = yq;
```
- Most érthetjük meg, hogy miért kellett a típusdefinicióban a `Nil` konstruktoralándót definiálni.

Deklaratív programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Egyszerű függvények lusta listákra

- A kiszámíthatóság érdekében egy függvény eredményének tetszőleges véges része az argumentum véges részétől függhet csak.
- Amikor az eredményre szükség van, akkor ez az igény váltja ki az argumentum feldolgozását.
- Első példánkban egészeket egyesével emelünk négyzetre. Amikor szükség van rá, az eredmény farka (egy függvény) alkalmazza a `square` függvényt az argumentum farkára.


```
(* square : int seq -> int seq *)
fun square Nil: int seq = Nil
  | square (Cons(x, xf)) = Cons(x * x, fn () => square(xf()));
```
- Két lusta lista hasonlóan adható össze.


```
(* addq : (int seq * int seq) -> int seq *)
fun addq (Cons(x, xf), Cons(y, yf)) = Cons(x+y, fn () => addq(xf(), yf()))
  | addq _ : int seq = Nil;
```

Deklaratív programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Listák rendeltése 520

Magasabb rendű függvények lusta listákra

- A `map` lusta változata:


```
(* mapq : ('a -> 'b) -> 'a seq -> 'b seq *)
fun mapq f (Cons(x, xf)) = Cons(f x, fn () => mapq f(xf()))
  | mapq f Nil = Nil;
```
- A `filter` lusta változata:


```
(* filterq : ('a -> bool) -> 'a seq -> 'a seq *)
fun filterq p (Cons(x, xf)) = if p x
  then Cons(x, fn () => filterq p(xf()))
  else filterq p(xf())
  | filterq p Nil = Nil;
```
- `square` a korábban láthatónál sokkal egyszerűbben definiálható `mapq`-val:


```
val squareq = mapq (fn i => i * i);
```

Deklaratív programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Magasabb rendű függvények lusta listákra (folyt.)

- Olyan számsorozatot állítunk elő, amelyben 50-nél nagyobb, 7-esre végződő egészek vannak:


```
filterq (fn n => n mod 10 = 7) (fromq 50);
```
- Az iteratorteg függvény – a fromq egy általánosítása – a következő sorozatot állítja elő (vö. repeat-tel): $[x, f(x), f(f(x)), \dots, f^k(x), \dots]$.


```
(* iteratorteg : ('a -> 'a) -> 'a -> 'a seq *)
fun iteratorteg f x = Cons(x, fn () => iteratorteg f (f x));
```
- fromq-t iteratorteg-val így definiálhatjuk:


```
(* fromq : int -> int seq *)
val fromq = iteratorteg (fn i => i+1);
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Álvéletlen számok (folyt.)

- Ha a nextrandom-ot 1.0 és 2147483647.0 közötti seed-re alkalmazzuk, ugyanebbe a tartományba eső más értéket állít elő az a * seed mod m művelettel. (A valós számokat a tájcsorolás elkerülésére használjuk.)
- A lusta lista előállítására iteratorteg-t nextrandom-ra és seed valós számmá alakított kezdőértékére alkalmazzuk. mapq gondoskodik arról, hogy a lusta listában minden értéket elosszunk m-mel, és így randseq 0.0-nál nem kisebb és 1.0-nél kisebb értékeket adjon eredményül. Látható, hogy a lusta lista a megvalósítás részletét szépen elrejtí a felhasználó elől.
- Az előállított álvéletlen-számok 0.0-nál nem kisebb és 1.0-nél kisebb valós számok; mapq-val alakíthatjuk át őket 0 és 1 közötti egészekké:


```
mapq (floor o sec1 10.0 op*) (randseq 1);
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Álvéletlen számok

- Hagymányos álvéletlenszám-generátorok: olyan eljárások, amelyek egy *frissíthető változóban* tárolják a *seed* (mag) értéket – ebből állítják elő egy következő hívásnál a következő álvéletlen számot.
- Lusta listaként megvalósítva: a következő álvéletlen szám csak szükség esetén áll elő.


```
(* randseq : int -> real seq *)
local val a = 16807.0 and m = 2147483647.0
  (* nextrandom : real -> real
   *)
  fun nextrandom seed =
    let val t = a * seed
        in t - real(floor(t/m)) * m
        end
    in
      fun randseq s = mapq (secr op/ m) (iteratorteg nextrandom (real s))
    end;
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Prímszámok előállítása eratoszteniési szűréssel

- Az algoritmus:
 1. Vegyük az egészek 2-vel kezdődő sorozatát: (2, 3, 4, 5, 6, 7, ...).
 2. Töröljük az összes 2-vel osztható számot: (3, 5, 7, 9, 11, ...).
 3. Töröljük az összes 3-mal osztható számot: (5, 7, 11, 13, 17, 19, ...).
 4. Töröljük az összes 5-tel osztható számot: (7, 11, 13, 17, 19, ...).
 5. Töröljük az összes ...
- A sorozat első eleme mindig a következő prím. A sorozatban azok a számok maradnak benne, amelyek az eddig előállított prímeikkel nem oszthatók.


```
(* sift : int -> int seq -> int seq *)
fun sift p = filterq (fn n => n mod p <> 0);
```
- A sift a p argumentum többszöröseit törli egy lusta listából.

Deklaratív Programozás, BMEF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Primszámok előállítása eratoszteniési szitával (folyt.)

- A sieve-nek már csak ismételtlen alkalmaznia kell sift-et a megfelelő lista listára. Mivel ez a lista sohasem üres, nem kell az üres lista listára illeszkedő változatot írunk.

```
(* sieve : int seq -> int seq *)
fun sieve (Cons (p, nfx)) = Cons (p, fn () => sieve (sift p (nf ())))
  | sieve Nil = Nil;
```

Négyzetgyökvonás Newton-Raphson módszerrel (folyt.)

- Ezzel

```
(* groot : real -> real *)
fun groot a = within 1E~6 (iterateq (nextapprox a) 1.0);
```

- A példában világosan különválasztjuk a leállásvizsgálatot (termination test) a következő jelölt előállításától.

Most az abszolút különbséget $(|x - y| < \epsilon)$ tesztelejük, de vizsgálhatnánk pl. a relatív különbséget $(|\frac{x}{y} - 1| < \epsilon)$ vagy az $\frac{|x-y|}{\frac{|x|+|y|}{2}} < \epsilon$ feltételt.

A feladat többi része független attól, hogy milyen leállásvizsgálatot alkalmazunk, és így is kell megfogalmazni a megoldást.

Négyzetgyökvonás Newton-Raphson módszerrel

- nextapprox x_k -ből x_{k+1} -et számítjuk ki az $x_{k+1} = \frac{x_k + x}{2}$ képlet alapján.

```
(* nextapprox : real -> real -> real *)
fun nextapprox a x = (a/x + x)/2.0;
```

- A befejeződés megállapítására egyszerű tesztet írunk:

```
(* within : real -> real seq -> real *)
fun within (eps: real) (Cons (x, xf)) =
  let val Cons (y, yf) = xf ()
  in
    if abs (x-y) <= eps then y else within eps (Cons (y, yf))
  end;
```

A (Cons (y, yf)) és az xf () lista lista ugyanaz: az else-ágban azért használjuk az első, mert xf () meghívása költségesebb.

Négyzetgyökvonás Newton-Raphson módszerrel (folyt.)

- Írjunk függvényt a következő jelölt előállítására, és rejtjük el a részleteket:

```
(* approxq : real -> real seq *)
fun approxq a =
  let (* nextapprox : real -> real
      *)
  in
    fun nextapprox x = (a/x + x) / 2.0
    in iterateq nextapprox 1.0
    end;
```

- Ezzel groot egy „tisztább” változata:

```
(* groot : real -> real *)
val groot = within 1E~6 o approxq;
```

Keresztszorzatokból álló lista

- Legyen xq és yp egy-egy sorozat. Képezzünk új sorozatot az (x_i, y_j) párokból, ahol $x_i \in xq$ és $y_j \in yp$!
- Először hagyományos listákra oldjuk meg a feladatot `map` és `pair` alkalmazásával.
- xs és ys egy-egy lista. Képezzünk listát az (x_i, y_j) párokból, ahol $x_i \in xs$ és $y_j \in ys$!
- `map-et`, `pair-t` és `List.concat`-ot alkalmazva juthatunk el a keresett függvényhez.


```
(* pair : 'a -> 'b -> ('a * 'b) *)
fun pair x y = (x, y);
```
- A `pair-t` a `map-pel` az `ys` lista elemeire alkalmazva olyan párokból álló listát kapunk eredményül, amelyben a párok első tagja a rögzített x érték, a második tagja pedig az `ys` egy-egy eleme.


```
map (pair x) ys
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Listák rendeltése 531

Keresztszorzatokból álló lista lista

- A páirs-hez hasonlóan állíthatjuk elő párok lista listájának lista listáját:


```
(* pairq : 'a seq -> 'b seq -> ('a * 'b) seq seq *)
fun pairq xq yq = mapq (fn x => mapq (pair x) yq) xq;
```
- Az eredmény véges része kiterítható `takeqq`-val, amely a bal felső saroktól számtól első m sorból és n oszlopból álló téglalapot jeleníti meg az `xq` lista listából:


```
(* 'a takeq : (int * int) * 'a seq seq -> 'a list list *)
fun takeq (m, n), xq = map (secl n takeq) (takeq(m, xq));
```
- Példa: olyan lista lista, amelyben a párok első tagja az egymás után következő egészek 30-tól kezdve, második tagja pedig a prímszámok 2-től kezdve:


```
- pairq (fromq 30) primes;
> val it = Cons (Cons ((30, 2), fn) : (int * int) seq seq
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Keresztszorzatokból álló lista (folyt.)

- Hogyan érhetjük el, hogy az x végigfusson az xs lista összes elemén? Az eddig szabad x -et kössük le egy függvény argumentumaként:


```
fn x => map (pair x) ys

majd alkalmazzuk újból a map-et erre a függvényre és  $xs$ -re:
map (fn x => map (pair x) ys) xs
```
- Listák listáját kapjuk eredményül, mert a belső `map` már listát adott vissza, amelyek minden eleméből újabb listát képeztünk a külső `map-pel`.
 `List.concat` elvégzi a szükséges simítást:


```
(* pairs : 'a list -> 'b list -> ('a * 'b) list *)
fun pairs xs ys = flat (map (fn x => map (pair x) ys) xs);
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Listák rendeltése 532

Keresztszorzatokból álló lista lista (folyt.)

- `takeq` `((3, 5), it)`:


```
> val it = [[(30, 2), \ldots, (30, 11)],
            [(31, 2), \ldots, (31, 11)],
            [(32, 2), \ldots, (32, 11)]] : (int * int) list list
```
- Ha ki akarjuk simítani a lista listát, egy `List.concat`-hoz hasonló, lista listákra alkalmazható függvényel nem megyünk semmire: ha `xq` végtelen, `appendq (xq, yq) = xq`. Azonban két lista lista elemei páronként egymásba ékelhetők:


```
(* interleaveq : 'a seq * 'a seq -> 'a seq *)
fun interleaveq (Nil, yq) = yq
  | interleaveq (Cons (x, xf), yq) = Cons(x, fn () => interleaveq(yq, xf()));
```
- `interleaveq` a rekurzív hívásban változtatja a két lista listát.


```
- takeq(10, interleaveq(fromq 0, fromq 50));
> val it = [0, 50, 1, 51, 2, 52, 3, 53, 4, 54] : int list
```

Deklaratív Programozás, BMEF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Keresztszorzatokból álló lista lista (folyt.)

- `enumerate`: lista listák listájából egyetlen lista listát állít elő. Legyen a kétszemes mélységű lista lista feje `xq` és a farka `xqf`; alkalmazzuk `enumerate`-et rekurzívan `xqf-re`, majd az eredményt ékeljük `xq`-ba:

```
(* enumerate : 'a seq seq -> 'a seq *)
fun enumerate Nil = Nil
  | enumerate (Cons (xq, xqf)) = interleaveq (xq, enumerate(xqf()));
```

Ez a „megoldás” nem jó, mert a „végtelen” lista lista miatt a rekurzió nem ér véget: az SML-ben, amely alapvetően möhög kiértékelésű, a rekurzív hívást késleltetni kell.

- Több esetet kell megkülönböztetnünk:

```
(* 'a enumerate : 'a seq seq -> 'a seq *)
fun enumerate Nil = Nil
  | enumerate (Cons (Nil, xqf)) = enumerate (xqf())
  | enumerate (Cons (Cons (x, xf), xqf)) =
    Cons(x, fn () => interleaveq(enumerate(xqf()), xf()));
```

Deklaratív Programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)

Keresztszorzatokból álló lista lista (folyt.)

- Ha a bemenő lista lista üres, készen vagyunk. Ha nem üres, meg kell vizsgálni a lista lista fejét: ha ez üres, akkor folytatni kell a rekurzív hívást, ha nem üres, akkor az explicit `fn () => ...` függvénydefiniációval *késleltetni* kell a rekurziót.

- Példa: pozitív egészekből álló párok egy lista listáját!

```
- val posintbq = pairbq (fromq 1) (fromq 1);
> val posintbq = Cons (Cons ((1, 1), fn):(int * int) seq seq
- takeq(15, enumerate posintbq);
> val it = [(1,1), (2,1), (1,2), (3,1), (1,3), (2,2),
(1,4), (4,1), (1,5), (2,3), (1,6), (3,2),
(1,7), (2,4), (1,8)]: (int * int) list
```

Deklaratív Programozás, BMF, 2001 tavaszi félév

26. előadás (funkcionális programozás)