Deklaratív programozás

A logikai és funkcionális programozás alapjai

Csató Lehel

Matematika-Informatika Tanszék
Babeş–Bolyai Tudományegyetem, Kolozsvár

2022 ősz

Az Előadások Témái

- Imperatív/deklaratív programok, Prolog alapok
- Listák, listakezelés
- Aritmetikai műveletek és operátorok
- Vágások használata, negáció
- Gyűjtő-predikátumok Prolog-ban, összegzések
- Vég-rekurzió optimalizálás
- Dinamikus predikátumkezelés; input-output prolog-ban
- Kifejezések dekompozíciója
- Rendezések, kereső algoritmusok
- Funkcionális programozás bevezető, történelem
- A Haskell programnyelv elemei, példaprogramok
- Algebrai típusok, típusosztályok
- Magasabb-rendű függvények, Map-Reduce, rendezések
- Input-output
- *(opc) Algebrai struktúrák 2: Monádok stb*
- Ismétlések és kérdések megvitatása

link: Canvas learning management system (LMS)

Régi honlap: `http://www.cs.ubbcluj.ro/~csatol/log_funk`

Vizsga (félév elején) – lásd SYLLABUS!

Labor (40%) + Parciális (25%) + Kollokvium (35%)

Laborgyakorlatok:

1. 2 Prolog feladatcsoport **20%**
2. 2 Haskell feladatcsoport **20%**
3. $\aleph_0$ feladat **első n beküldőnek**

Prolog, IQSYS, deklaratív programozás

Interjú Szeredi Péterrel[1]

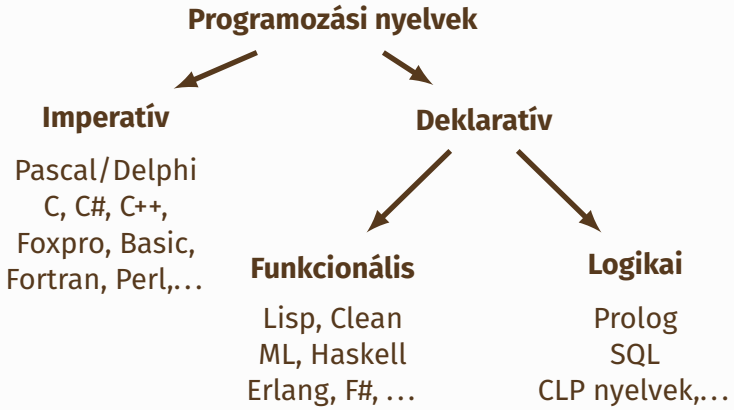
Úgy érzem, ma a logikai programozás is az általánosan elfogadott programozási paradigmák egyike.

A deklaratív programozás általában is halad szép lassan előre, mert bizonyos területeken az átláthatóság, a biztonság különösen fontos.

*Nagy cégek, mint pl. az Ericsson, a Motorola, vagy a Google (**map-reduce**) használnak deklaratív nyelveket.*

[1] http://www.agent.ai/main.php?articleID=796

Deklaratív vs. Imperatív



Deklaratív vs. Imperatív

- **Imperatív nyelvek:**
  - felszólító módú utasítás; változók használata;
  - például:

```
1 int fakt(int n) {
    int f=1;
    while(n>1) f *= n--;
    return f;
  }
```

- **Deklaratív nyelvek:**
  - kijelentő módú egyenletek, állítások;
  - változó: ismeretlen, de értéke *rögzített*;
  - Például (Clean / Haskell):

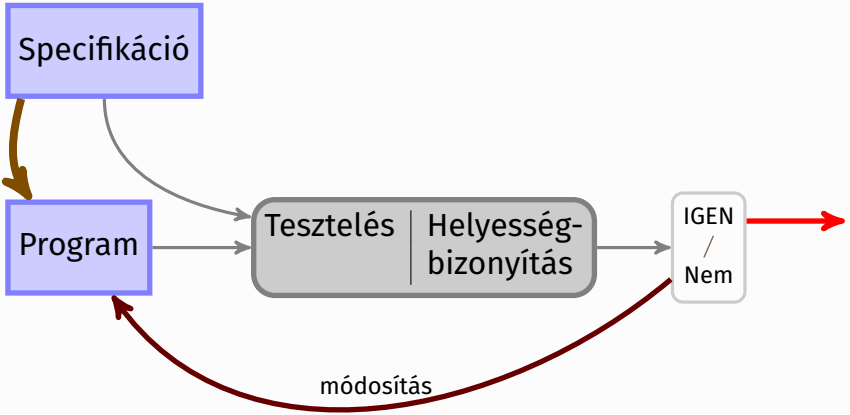
```
fakt 0 = 1
fakt n | n>0 = n * fakt (n-1)
```

Deklaratív nyelvek jelszavai:

- Egyszeres értékadás:
  - az értékadások sorrendje nem változtat az eredményen;
  - lehetővé teszi a párhuzamos végrehajtást.
- **A mit (what)** és nem a hogyan-t kell leírni:
  - nem fontosak a ciklusok vagy a változók deklarálása;
  - fontos, hogy a specifikáció szerint működjön a program.

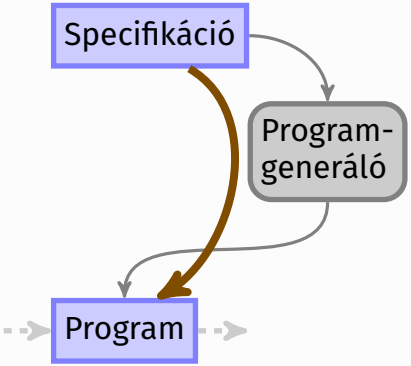
- Új, magas-szintű programozási elemek elsajátítása:
  - mintaillesztés,
  - visszalépéses keresés használata,
  - rekurzió (vég-rekurzió – tail-recursion).
- Más – új – gondolkodásmód:
  - önálló programrészek írása,
  - a kód és a jelentés összevethető: **program-verifikáció**.
- Új alkalmazási területek:
  - szimbolikus számítások,
  - következtetési módszerekre épülő megoldások,
  - nagyfokú biztonságot igénylő rendszerek,

Fejlesztési ciklus imperatív / deklaratív nyelveknél



Rendszerek fejlesztése hagyományos – imperatív programozás – esetén.

Fejlesztési ciklus imperatív / deklaratív nyelveknél



Rendszerek fejlesztése a deklaratív programozás segítségével.

Logikai programozás – általánosan

Alapelv:

- A program elemei logikai állításoknak felelnek meg.
  pl.

```
nagyszulo(U,N) :- szulo(U,Sz),szulo(Sz,N).
```

  Matematikai jelölés:

$$\forall \mathrm{U}\ \forall \mathrm{N}\ \exists \mathrm{Sz}\ \ \big(\mathrm{nsz}(\mathrm{U},\mathrm{N}) \leftarrow \mathrm{sz}(\mathrm{U},\mathrm{Sz}) \wedge \mathrm{sz}(\mathrm{Sz},\mathrm{N})\big)$$

- A program futása $\equiv$ dedukció (tételbizonyítás).

Logikai programozás első megvalósítása: Prolog

- A logikai állítások egyszerűek – eljárásdefiníciók;
- Tételbizonyítás, mint: mintaillesztés + visszalépéses keresés;
- Prolog = RDBMS + rekurzió + adatstruktúrák[2];

[2] Hanák, Szeredi, Benkő jegyzetek

Prolog - előnyök / hátrányok

Előnyök:

- tömör kód;
- „automatikus” visszalépéses keresés;
- logikai változó – meghatározatlan adatok kezelése,

Hátrányok:

- nehézkes tanulás – „tapasztalt programozóknak”.
- rögzített, rugalmatlan vezérlési mechanizmus.
- gyenge általánosítási képesség – ha a pontos következtetési szabályok nem alkalmazhatók.

Tovább:

- CLP - constraint logic programming.
- típusok – Mercury
- rugalmas vezérlés, párhuzamosság.

Prolog – Irodalom

Magyar nyelven:

- **Futó Iván (szerk):** Mesterséges Intelligencia
  (9.2. fejezet, Szeredi Péter).
- **Ásványi Tibor:** A logikai programozás és a Prolog.
  https://aszt.inf.elte.hu/~asvanyi/pl/jegyzetek/pl.pdf.gz
  https://canvas2.cs.ubbcluj.ro/files/214426/download?download_frd=1
- **Szeredi Péter, Benkő Tamás:** Deklaratív programozás.
- **Csató Lehel, Egri Edit:** A logikai és a funkcionális programozás alapjai.
  https://canvas2.cs.ubbcluj.ro/files/214419/download?download_frd=1

A link-ek megtalálhatóak:

Canvas oldalon

Prolog nyelv:

- Tények:

```
ferfi(adam).
no(eva).
gyereke(iszak,adam).
gyereke(iszak,eva).
```

  (hasonló az SQL táblához)

- Szabályok:

```
apja(X,Y) :- gyereke(Y,X),ferfi(X).
```

  `:-` következtetés; (baloldal igaz, ha a jobb teljesül)

  `,` „és”; (konjunkció)

  - A szabályok a tények közötti kapcsolatokat rögzítik.

Prolog – atomok, változók

Atomok: kisbetűvel kezdődnek vagy „'" közé zárt karakterek:

`adam`, `xYZ`, `a_123`, `'Another pint please'`

Változók: nagybetűvel vagy aláhúzásjellel kezdődnek:

`X`, `Utod`, `_`, `_ez_sem`

„Don't care" változó `_`

- minden változó, melynek első betűje `_`;
- lehet értéke, azonban ezt nem használjuk;

Példa:
Van-e szülője `adam`-nak.

```
gyereke(adam,_).
false
```

Az _ változóra **nem tudunk hivatkozni**!
A `gyereke(adam,_0s).` érthetőbb.

Atomok: kisbetűvel kezdődnek vagy „'" közé zárt karakterek:

`adam`, `xYZ`, `a_123`, `'Another pint please'`

Változók: nagybetűvel vagy aláhúzásjellel kezdődnek:

`X`, `Utod`, `_`, `_ez_sem`

„Don't care" változó `_`

- minden változó, melynek első betűje `_`;
- lehet értéke, azonban ezt nem használjuk;

Példa:
Van-e szülője `adam`-nak.

```
gyereke(adam,_).
false
```

Az `_` változóra **nem tudunk hivatkozni**!
A `gyereke(adam,_Os).` érthetőbb.

Lekérdezések:

```
?- gyereke(izsak,adam).
Yes
```

- a tudásbázis alapján a rendszer „következtet”.

```
?- gyereke(izsak,X).
adam;
eva;
false.
```

- Keressük azon X változókat, melyre igaz a **gyereke** reláció.
- „;” = vagy – diszjunkció
- Illesztünk egy megoldást, majd a következőt, ...majd ... **false**.

Prolog – Lekérdezések

A lekérdezés, a `?-` hasonló az SQL nyelv `SELECT` parancsához. Például:

```
?- gyereke(izsák,X).
```

Ugyanaz, mint

```
SELECT Szulo FROM gyereke WHERE gyerek="izsák";
```

Új táblák létrehozása szabályokkal történik.
Gyakori a tranzitív lezárás (rekurzió):

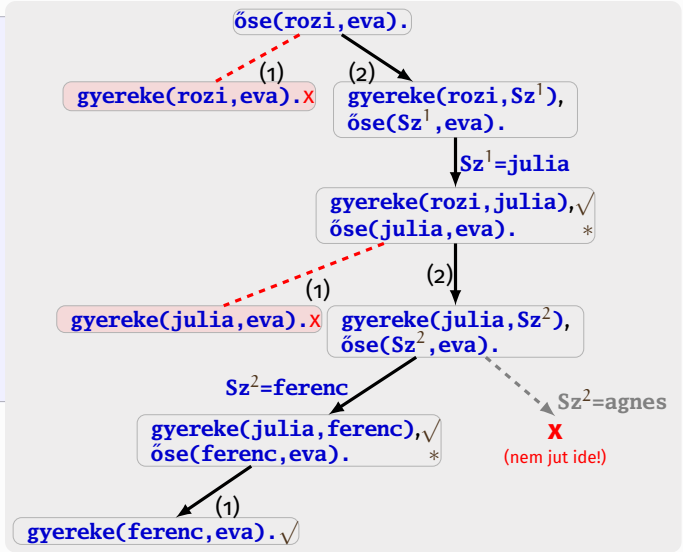
```
ose(Utod, Os) :-
    gyereke(Utod,Os).
ose(Utod, Os) :-
    gyereke(Utod,Utodszulo),
    ose(Utodszulo,Os).
```

Illusztráció – rekurzió

```
gyereke(ferenc,adam).
gyereke(ferenc,eva).
gyereke(marika,adam).
gyereke(marika,eva).
gyereke(peter,marika).
gyereke(peter,jozsef).
gyereke(julia,ferenc).
gyereke(julia,agnes).
gyereke(endre,julia).
gyereke(rozi,julia).
gyereke(endre,pali).
gyereke(sara,endre).
gyereke(sara,zita).
gyereke(zsolt,endre).
```

Illusztráció – rekurzió

```
1  gyereke(ferenc,adam).
   gyereke(ferenc,eva).
   gyereke(marika,adam).
   gyereke(marika,eva).
   gyereke(peter,marika).
6  gyereke(peter,jozsef).
   gyereke(julia,ferenc).
   gyereke(julia,agnes).
   gyereke(endre,julia).
   gyereke(rozi,julia).
11 gyereke(endre,pali).
   gyereke(sara,endre).
   gyereke(sara,zita).
   gyereke(zsolt,endre).
```



```
őse(Utod, Os) :-
      gyereke(Utod,Os).
őse(Utod, Os) :-
      gyereke(Utod,Utodszulo),
      őse(Utodszulo,Os).
```

- Többször is definiálhatunk egy predikátumot;
- **Igaz,** ha teljesül az első **vagy** a második, vagy ...
- Ez a diszjunkció jelölése a logikai programozásban.

```
őse(Utod, Os) :-
      gyereke(Utod,Os).
őse(Utod, Os) :-
      gyereke(Utod,Utodszulo),
      őse(Utodszulo,Os).
```

- Többször is definiálhatunk egy predikátumot;
- **Igaz,** ha teljesül az első **vagy** a második, vagy ...
- Ez a diszjunkció jelölése a logikai programozásban.

Példa:

```
ember(X) :- ( ferfi(X) ; no(X) ).
```

```
őse(Utod, Os) :-
      gyereke(Utod,Os).
őse(Utod, Os) :-
      gyereke(Utod,Utodszulo),
      őse(Utodszulo,Os).
```

- Többször is definiálhatunk egy predikátumot;
- **Igaz,** ha teljesül az első **vagy** a második, vagy ...
- Ez a diszjunkció jelölése a logikai programozásban.

Példa:

```
ember(X) :- ( ferfi(X) ; no(X) ).

ember(X) :- ferfi(X).
ember(X) :- no(X).
```

```
őse(Utod, Os) :-
     gyereke(Utod,Os).
őse(Utod, Os) :-
     gyereke(Utod,Utodszulo),
     őse(Utodszulo,Os).
```

Példa:

```
ember(X) :- ( ferfi(X) ; no(X) ).

ember(X):-
   ferfi(X).
ember(X):-
   no(X).
```

Átláthatóbb kód – könnyebb tervezés.

Axiomatikus mód

A Prolog-ban fel tudjuk építeni a természetes számokat.

- Természetes szám a nulla;
- Egy szám természetes szám, ha valamely más természetes szám után következik.

A **Prolog** kód:

```
1 tSzam(z).
  tSzam(s(X)):-
      tSzam(X).
```

A természetes számok axiómái:
A `z` a nullást, az `s(·)` a „következő” függvényt jelöli.

Írjuk le a `?- tSzam(X).` lekérdezés összes válaszát.

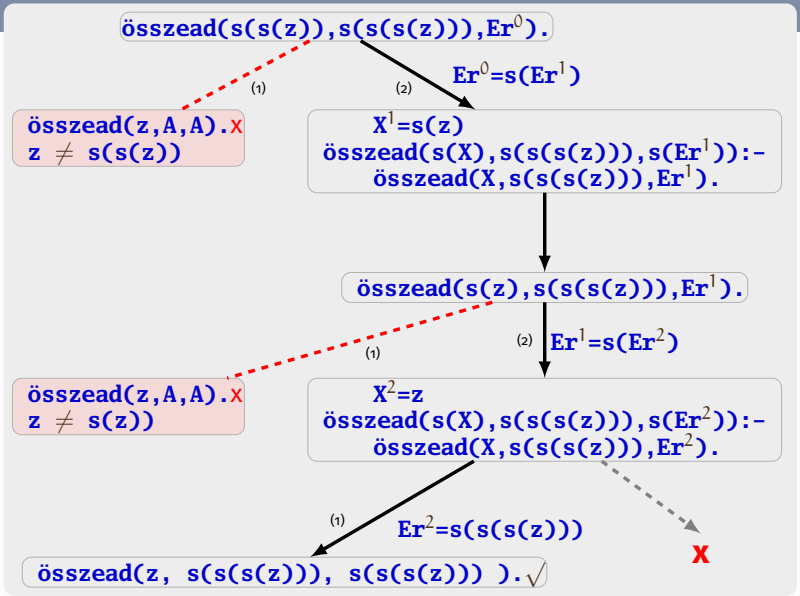
A páros számokat meghatározó predikátum:

```
paros(z).
paros(s(s(X))):-
    paros(X).
```

Az összeadás kódja:

```
osszead(z,Eredm,Eredm).
osszead(s(X),Kif,s(Eredm)):-
    osszead(X,Kif,Eredm).
```

A kettő ábrázolása: `s(s(z))`, a háromé `s(s(s(z)))`.
Számítsuk ki a $2 + 3$ kifejezés értékét a fenti ábrázolás szerint.



- **Írjuk le a $2 + X = 5$ egyenlet Prolog kódját.**
- Az ötből ki kell vonnunk kettőt, az eredmény három.
- A Prolog rendszer levezeti az eredményt.
- `?- osszead(2,X,5).` a Prolog kód, az eredmény `s(s(s(z)))`, ami a három reprezentációja.
- A Prolog predikátumok multifunkcionálisak: a paraméterek helye szerint az `összead` predikátummal kivonni is tudunk.
- Mivel nem hatékony a rendszer, a Prolog **nem az axiomatikus** műveleteket használja.

A természetes számokat „axiomatizáló” és visszaalakító predikátumok.

Aritmetikai műveleteket is használnak.

```
% transzf(N,AxN) ax.
% formába alakitás
transzf(0,z).
transzf(X,s(Z)):-
    X>0,
    X1 is X-1,
    transzf(X1,Z).

% iTranszf(AxN,N) ax.
% formából alakítás
iTranszf(z,0).
iTranszf(s(X),A):-
    iTranszf(X,A1),
    A is A1+1.
```

Teszt:

```
?- transzf(1233,A1), % és
 transzf(1511,A2), % és
 összead(A1,X,A2),% és
 iTranszf(X,N3).
```

```
X = 278.
```

Prolog összefoglaló

- A Prolog tudásbázist kezel;

  tények és szabályok

- Lekérdezések a tudásbázisból való információk kinyerésének módjai;

  ?- lekérdezés.

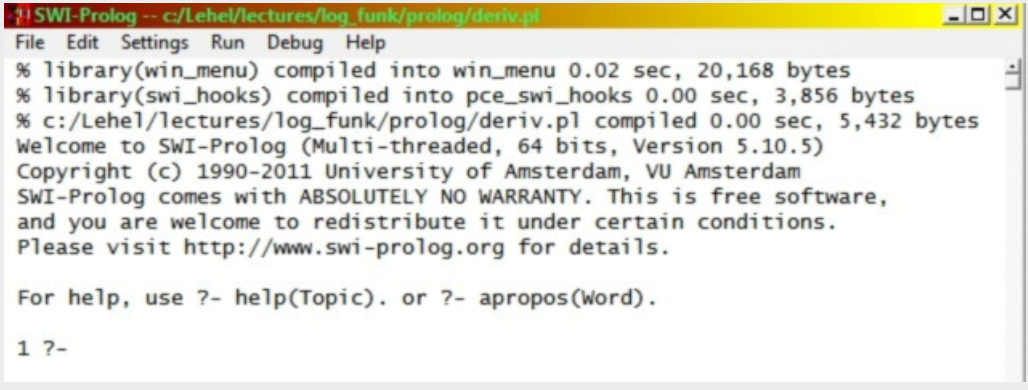
- A program, mivel megegyezik a specifikációval, **helyes**.

- Hatékony programok írása lehetséges a Prolog szintaxisának az ismeretében.

Az SWI-prolog rendszer I

- Ingyenes Prolog interpreter, nyitott forráskóddal;
- Létezik 32 és 64 bites változata a Windows operációs rendszeren;
- Elérhető különböző Linux-okra (Ubuntu: swipl);
- Nagyon hasznos a beépített EMACS és debug.
- Használjuk a **help** parancsot.

Prolog **parancsablak**:

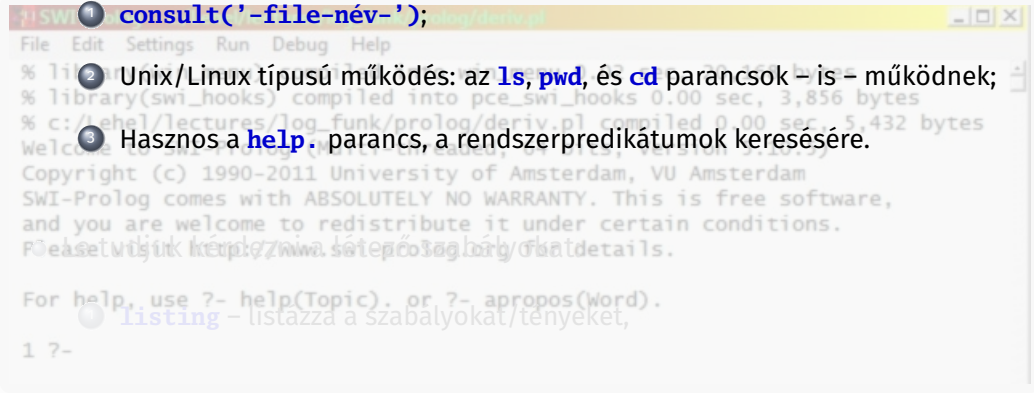


Az SWI-prolog parancs-ablak II

Prolog **parancsablak**:

- Lehetőség a tények/szabályok beolvasására

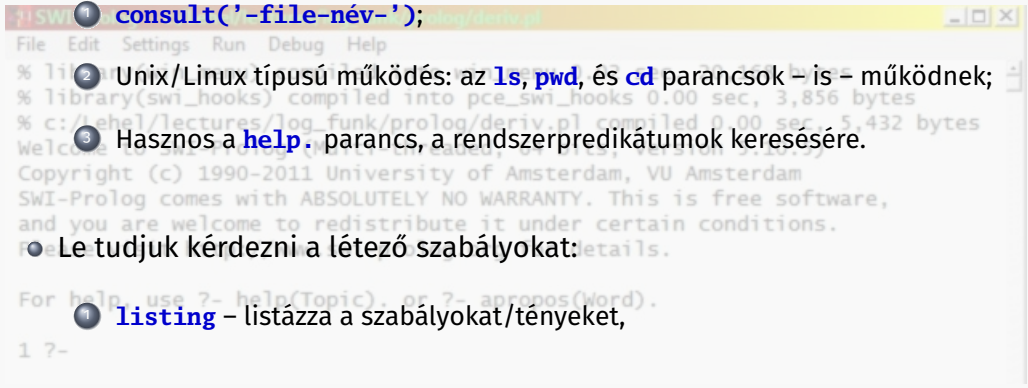
1. `consult('-file-név-');`
2. Unix/Linux típusú működés: az `ls`, `pwd`, és `cd` parancsok – is – működnek;
3. Hasznos a `help.` parancs, a rendszerpredikátumok keresésére.



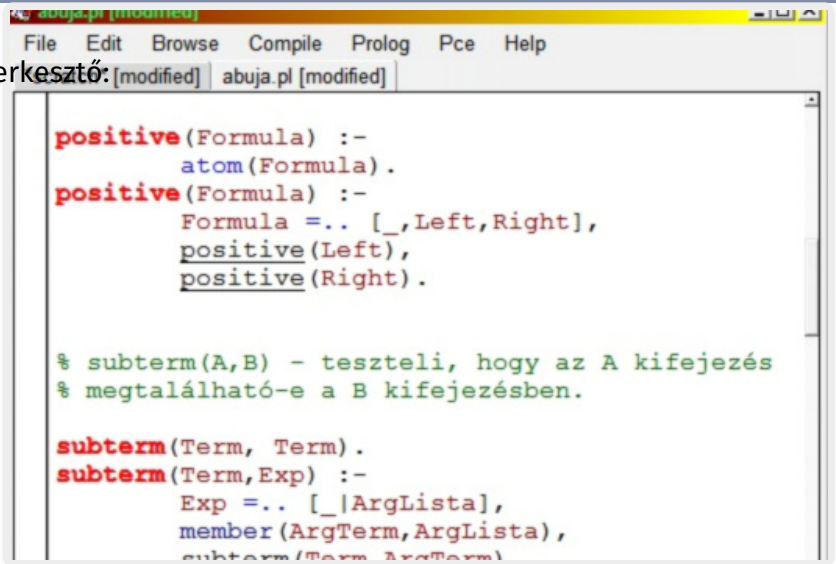
Az SWI-prolog parancs-ablak II

Prolog **parancsablak**:

- Lehetőség a tények/szabályok beolvasására
  1. `consult('-file-név-');`
  2. Unix/Linux típusú működés: az `ls`, `pwd`, és `cd` parancsok – is – működnek;
  3. Hasznos a `help.` parancs, a rendszerpredikátumok keresésére.
- Le tudjuk kérdezni a létező szabályokat:
  1. `listing` – listázza a szabályokat/tényeket,

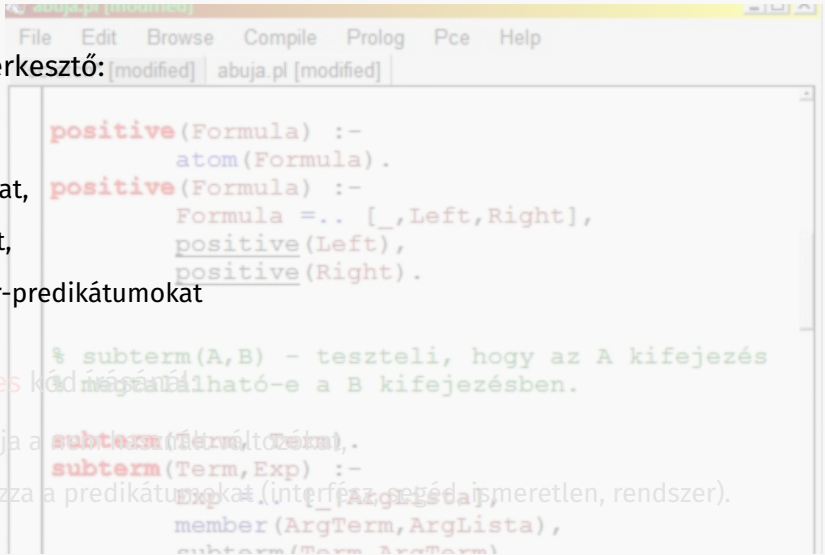


Emacs szövegszerkesztő:



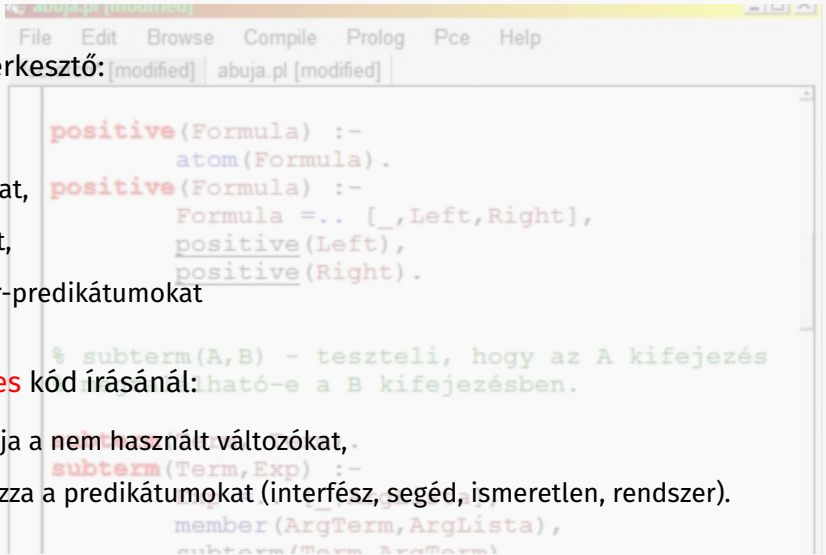
Emacs szövegszerkesztő:

- Kiemeli a
  1. változókat,
  2. rekurziót,
  3. rendszer-predikátumokat
- Segít a helyes kód írásánál
  1. azonosítja a változókat,
  2. osztályozza a predikátumokat (interfész, segéd, ismeretlen, rendszer).

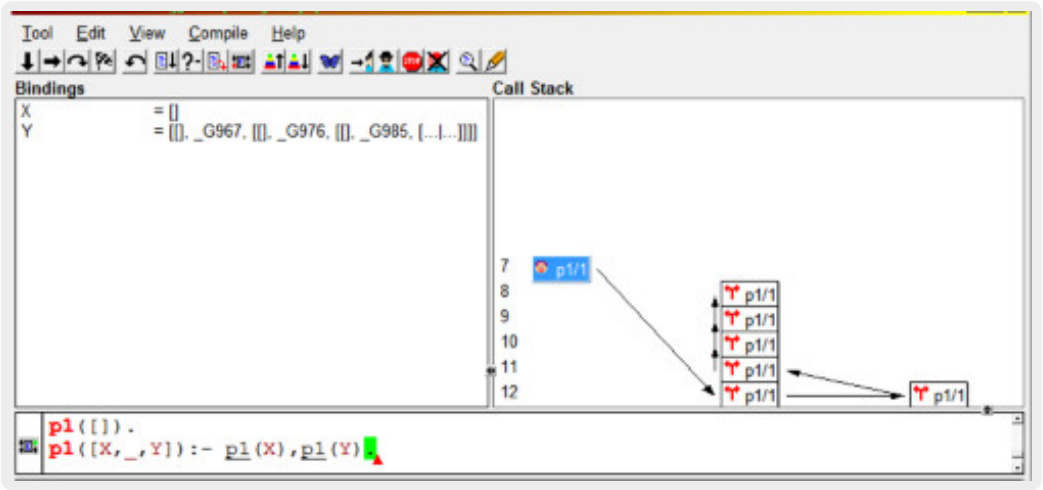


Emacs szövegszerkesztő:

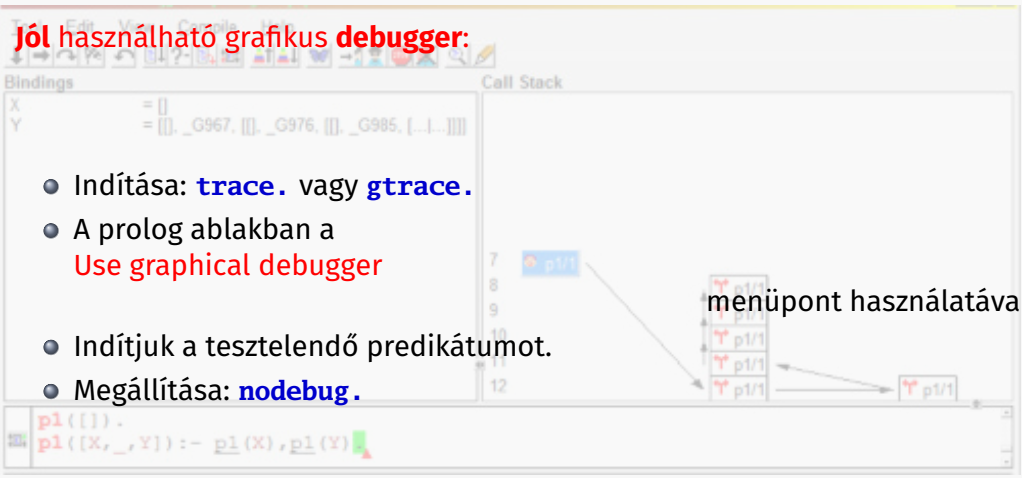
- Kiemeli a
  1. változókat,
  2. rekurziót,
  3. rendszer-predikátumokat
- Segít a helyes kód írásánál:
  1. azonosítja a nem használt változókat,
  2. osztályozza a predikátumokat (interfész, segéd, ismeretlen, rendszer).



Az SWI-prolog IV



Jól használható grafikus **debugger**:



- Indítása: `trace.` vagy `gtrace.`
- A prolog ablakban a
  Use graphical debugger
  menüpont használatával.
- Indítjuk a tesztelendő predikátumot.
- Megállítása: `nodebug.`

- `consult('filenév.pl')` - egy file-ban található predikátumok beolvasása.
- `var/1`, `atom/1`, `atomic/1`, `number/1` – tesztelő predikátumok, melyek igazak, ha argumentumuk egy változó, egy atom (nem összetett), stb.
- `string`, `float`, `nonvar` – szintén típusokat tesztelnek.
- `is_list`, `length(List,N`, `sort(BeL,KiL)`.
- `write`, `writeln`, `nl`, `format` kiíratási predikátumok.

- `consult('filenév.pl')` - egy file-ban található predikátumok beolvasása.
- `var/1`, `atom/1`, `atomic/1`, `number/1` – tesztelő predikátumok, melyek igazak, ha argumentumuk egy változó, egy atom (nem összetett), stb.
- `string`, `float`, `nonvar` – szintén típusokat tesztelnek.
- `is_list`, `length(List,N`, `sort(BeL,KiL)`.
- `write`, `writeln`, `nl`, `format` kiíratási predikátumok.

Logikai feladat

Jegyek

Öt diák vizsgázott három tárgyból. A jegyek az `[a,b,c,d,e]` lista elemei (`a` legjobb).

Tudjuk, hogy mindegyik diáknak más-más jegye volt a különböző tárgyakból, illetve azt is tudjuk, hogy a tárgyak osztályzatai különbözőek voltak a diákok között.

Ismerve, hogy

- **András** jobb eredményt ért el, mint **Brigi** fizikából, és jobb volt, mint **Nándor** matekből;
- **Vanda** volt az egyetlen lány, aki **c** jegyet kapott, és neki nem lett egy **a** jegye sem;
- Aki matekből **e**-t kapott, annak a kémia jegye **b**, illetve fizikából nem kapott **c**-t;
- **Pali** fizika jegye **d** volt; a legnagyobb jegye pedig **c**;
- Akinek matekből **b**-je volt, annak fizikából nem **e**-je volt;
- **Brigi** legjobb jegyét kémiából kapta, de a matek jegye gyengébb volt, mint a **Pali**é,

keressük meg a diákok jegyeit.

Az Előadások Témái

- Imperatív/deklaratív programok, Prolog alapok
- Listák, listakezelés
- Aritmetikai műveletek és operátorok
- Vágások használata, negáció
- Gyűjtő-predikátumok Prolog-ban, összegzések
- Vég-rekurzió optimalizálás
- Dinamikus predikátumkezelés; input-output prolog-ban
- Kifejezések dekompozíciója
- Rendezések, kereső algoritmusok
- Funkcionális programozás bevezető, történelem
- A Haskell programnyelv elemei, példaprogramok
- Algebrai típusok, típusosztályok
- Magasabb-rendű függvények, Map-Reduce, rendezések
- Input-output
- *(opc) Algebrai struktúrák 2: Monádok stb*
- Ismétlések és kérdések megvitatása

link: Canvas learning management system (LMS)

Régi honlap: `http://www.cs.ubbcluj.ro/~csatol/log_funk`

Vizsga (félév elején) – lásd SYLLABUS!

Labor (40%) + Parciális (25%) + Kollokvium (35%)

Laborgyakorlatok:

1. 2 Prolog feladatcsoport **20%**
2. 2 Haskell feladatcsoport **20%**
3. $\aleph_0$ feladat **első n beküldőnek**

Listák Prolog-ban

- Generikus adatstruktúra Prolog implementációja.
- Heterogén: minden elemének lehet más a típusa.
- Például:

```
[Z, bla, [], h(U,atom), 42, [a,b,c] ]
```

- Az `[a, b, c]` lista belső reprezentációja:

```
.(a, .(b, .(c, [])))
```

- A pont – `./2` – két-argumentumú predikátum, melyet egymásba ágyazhatunk.
- Az üres lista **speciális** atom, a `NULL` vagy `nil`-hez hasonlóan.

```
[]
```

Az SWI-Prolog 7-es verziójától a „.” fent van tartva a record típus mező-szelektornak.

Listák dekompozíciója

- Felbontjuk a listát:

```
Lista = [Fej| Maradék]
```

- Többféle dekompozíció lehetséges:

```
[a,b,c]=[a|[b,c]]=[a,b|[c]]=[a,b,c|[]]
```

- Például:

```
?- [1, 2, 3] = [Fej| Tobbi].
Fej = 1
Tobbi = [2,3]
```

- Másik példa:

```
[a] = [Fej| Tobbi].
Fej = a
Tobbi = []
```

Mindig az üres lista zár egy listát!

Listák dekompozíciója II

A `Lista = [Fej| Maradék]` minta

Szekvenciális / rekurzív adatfeldolgozásnál nagyon hasznos:

1. Feldolgozzuk a lista `Fej`-ét;
2. Rekurzívan hívjuk a predikátumot a `Maradék`-részre;
3. Megállunk, amikor a listánk kiürült.

Listák összefűzése

I

Példa a rekurzív predikátumra:

```
fuz([], Lista, Lista).

fuz([Fej| Marad], L2, [Fej| Marad2]):-
    fuz(Marad, L2, Marad2).
```

- A `fuz(L1,L2,Hosszú)` predikátum az `L1` és `L2` listák összefűzését illeszti a `Hosszú` listához.[3]
- Például:

```
?- fuz([a,b], [1,2,3], Eredm).
Eredm = [a,b,1,2,3]
```

[3] Kerültem a bemeneti/kimeneti minősítést. Miért?

Listák összefűzése II

És:

```
fuz(Eleje, Vege, [1,2]).
```

Válasz(ok):

```
Eleje=[]   Vege=[1,2]
Eleje=[1]   Vege=[2]
Eleje=[1,2]   Vege=[]
```

A Prolog esetén nincs bemenő/kimenő változó

Az illesztés által a változók különböző szerepeket vehetnek fel: lehetnek kontextustól függően bemenő vagy kimenő változók.

Lista elemeinek a lekérdezése

```
1 member(A, [A|_]).

  member(A, [_| Marad]) :-
      member(A, Marad).
```

- Egy lista elemeit sorra illeszti az **A** változóhoz:

```
member(Z,[a,b]).
Z=a;
Z=b;
false
```

- Ha **A** atom, a válasz logikai:

```
member(beka,[kecske,beka]).
true
member(macska,[kecske, beka]).
false
```

- Elágazásokat generál a levezetési fában

member alkalmazás

```
1 ?- member(Elem,[egy,kettő,három,négy]),
     write(Elem),
     fail.
```

Feladat: írjuk fel a levezetési fát.

- Az `X` rendre felveszi a lista elemeit,
- A `write(X)` – mellékhatásként – megjeleníti a lista elemeit.
- A `fail` predikátum mindig meghiúsul,
- A lekérdezés tehát sosem teljesül, azonban mellékhatásként kiíratjuk a lista összes elemét.

Lista hossza

Lista hosszát meghatározó predikátum:

```
hossz([], 0).

hossz([_| Marad], N) :-
    hossz(Marad, N1),
    N is N1 + 1.
```

- megfigyelhetjük a nem_érdekel_változó használatát.
- az **is** aritmetikai megfeleltetést végez (később bővebben).
- A predikátum-törzsben a sorrend fontos:
  - ha felcseréljük, nem tudunk értéket adni az **N**-nek;
  - a futás leáll hibával;
  - az **is** imperatív utasítás és a rendszert gyorsítja.

Elemek kiválasztása

Szeretnénk egy olyan predikátumot, mely egy listát felbont egy elemre és maradékra.

```
select(Fej, [Fej| Marad], Marad).

select(Mas, [Fej| Marad], [Fej| SMarad]) :-
    select(Mas, Marad, SMarad).
```

Kérdés: mi lesz a `select(pava,Madarak,[tyuk,liba,kakas,kacsa]).` lekérdezés eredménye?

Hány megoldás van?

Predikátum

Tudjuk, hogy a **select(E,L,R)** egy elemet választ az **L**-ből, a maradékot az **R**-ben tárolja.

Mire „jó” – mit számol:

```
1
f([], []).

f(L, [E | P]) :-
    select(E, L, R),
6   f(R, P).
```

```
permutation([], []).
permutation(List, [E | PermR]):-
    select(E , List, Rest),
    permutation(Rest, PermR).
```

- Egy permutációt talált meg/illeszt.
- a visszalépésnek hasznos felhasználása.
- összes permutáció: `?-findall(P,permutation([1,2,3],P),Lista).`

A Prolog-ban a lista hossza kiszámítható rekurzívan:

```
% lista hosszának számítása
l_hossz([_|Mar],N+1) :-
    l_hossz(Mar,N).

l_hossz([],0).
```

Az `l_hossz([a,b,c],N).` kérdésre a rendszer válasza `N=0+1+1+1`.

Annak ellenére, hogy a válasz helyes, szeretnénk azt rövid formában látni.

A rendszernek tudnia kell, hogy `0+1+1+1=3`.

Aritmetikai műveletek II

- A **0+1+1+1=3** azt jelenti, hogy a zérus utáni harmadik elem a természetes számok között (ld. első előadás);
- Nem hatékony a számok fenti ábrázolása – az **s(s(s(z)))** sem;

Például
Az összeadás Prolog kódja helyes, de nem hatékony:

```
osszead(z,Eredm,Eredm).
osszead(s(X),Kif,s(Eredm)):-
    osszead(X,Kif,Eredm).
```

Aritmetikai műveletek

is

- Az aritmetikai műveletek a Prolog rövidítései a gyorsabb számításokhoz;
- Az `is` operátor:
  1. a jobb oldali kifejezést **numerikusan** kiértékeli;
  2. egyesíti a választ a bal oldalon található kifejezéssel.
  3. a bal oldalon vagy változó van, vagy numerikus érték.

Példák:

```
% az X szabad változóra
?- X is 0+1+1+1.
X = 3.

% megegyeznek az értékek
?- 3 is 0+1+1+1.
true.
```

```
% kifejezés van a bal oldalon
?- 1+X is 0+1+1+1.
false.

% az X nem mindig szabad
?- X is 5, X is 1+1.
false.
```

Aritmetikai műveletek

is

- Az aritmetikai műveletek a Prolog rövidítései a gyorsabb számításokhoz;
- Az **is** operátor:
  1. a jobb oldali kifejezést **numerikusan** kiértékeli;
  2. egyesíti a választ a bal oldalon található kifejezéssel.
  3. a bal oldalon vagy változó van, vagy numerikus érték.

Példák:

```
% az X szabad változóra
?- X is 0+1+1+1.
X = 3.

% megegyeznek az értékek
?- 3 is 0+1+1+1.
true.
```

```
% kifejezés van a bal oldalon
?- 1+X is 0+1+1+1.
false.

% az X nem mindig szabad
?- X is 5, X is 1+1.
false.
```

Aritmetikai segédfüggvények:

- Használhatók az `is` jobb oldalán;
- `abs/1, sqrt/1, sign/1`;
- `round/1, ceiling/1, floor/1`;
- `max/2, min/2`;
- `sin/1,cos/1,tan/1,asin/1,acos/1,atan/1`;
- `log/1,log10/1,exp/1`;
- `pi/0,e/0,epsilon/0` – *konstansok*;
- `random/1` – véletlenszám generátor;

Lásd még az online `help`-et

- A rendszer gyorsítása a cél;
- Leszűkítése a deklaratív jellegnek;
- A Prolog-ban a kifejezések kifejezési fákként vannak ábrázolva;
- Az **is** a kifejezéseket értékeli ki – egyszerűsíti le.
- Használja az **=:=** – mindkét oldalt egyszerűsíti majd ellenőrzi az egyenlőséget:

```
3+4 == 4+3          false
3+4 =:= 4+3          true
```

Operátorok

- A programozói munka könnyítése;
- Egy vagy két-argumentumú függvények;
- `infix`, `prefix` vagy `postfix` típusúak;
- A összehasonlító műveletek: `<`, `=<`, `>`, `>=`, `=:=`, `=\=`, ...
- Az aritmetikai műveletek: `+`, `-`, `*`, `/`, `//` egésszel történő osztás, `(.)` a precedencia felülírása, ...
- Saját operátorokat definiálhatunk.

Lásd még az online **help**-et

Operátorok II

Saját operátorok definiálása: létező predikátumokat alakíthatunk át:

- Létezik a `ferfi/1` és a `no/1` predikátum;
- Az `op/3` predikátummal kijelentjük az operátorokat;
- A Prolog kód beszédesebb.

```
% operátorok kijelentése
:- op(300,xf,ferfi).
:- op(300,xf,no).
```

```
adam ferfi.
eva no.
ferenc ferfi.
julia no.
...
```

- A `current_op(Prec,Asszoc,Op)` predikátummal megkapjuk az `Op` operátor precedencia-szintjét és az asszociativitási mintát;
- A precedencia-szint 1 és 1200 között van;
- `Assoc=xfy` – `f` operátor, `x` és `y` operandusok.

Kiírathatjuk az összes operátort:

```
current_op(Pr,As,Op),
format('~d - prec: ~d, assoc: ~d\n',[Op,Pr,As]),
fail.
```

Megváltoztathatjuk az operátorok precedenciáját:
Változtassuk meg a `+` és a `*` operátorok precedenciáit úgy, hogy az `X is 4+5+6*7` Prolog kifejezés értéke `105` legyen.

Feladat: írjunk predikátumot, mely egy polinomot értékel:
poliErt(Poli,L_in,L_out)

A **Poli**=[[0,5],[1,-3],[4,1]] a $p(x) = 5 - 3x + x^4$ polinom.

A **L_in**=[-2,-1,0,1,2] listára a [27,9,5,3,15] a válasz.

Lépések:

1. A listát értékelő **lPoli**/2 hívja az elemet értékelő **egyPoli**-t;
2. A polinomot a $p(x) = a_0 + x \cdot p'(x)$ alakra hozzuk, ahol $p'(x)$ a csökkentett fokszámú polinom.
3. A **csokkent**/2 csökkenti egy polinom fokszámát;
4. rekurzió a polinom és az elemlista szerint is.

```
lPoli(Poli,[Fej|Mar],[VFej|VMar]):-
    egyPoli(Poli,Fej,VFej),
    lPoli(Poli,Mar,VMar),
    !.
lPoli(_,[],[]).

egyPoli([[0,A]|Mar],X,Ert):-
    csokkent(Mar,MarCs),
    egyPoli(MarCs,X,MErt),
    Ert is A + X * MErt,
    !.
egyPoli([[P,A]|Mar],X,Ert):-
    P>0,
    csokkent([[P,A]|Mar],Csokk),
    egyPoli(Csokk,X,Ert2),
    Ert is X*Ert2,
    !.
egyPoli([],_,0).
```

Feladat: írjunk predikátumot, mely egy polinomot értékel:
poliErt(Poli,L_in,L_out)

A **Poli**=[[0,5],[1,-3],[4,1]] a $p(x) = 5 - 3x + x^4$ polinom.

A **L_in**=[-2,-1,0,1,2] listára a [27,9,5,3,15] a válasz.

Lépések:

1. A listát értékelő **lPoli**/2 hívja az elemet értékelő **egyPoli**-t;
2. A polinomot a $p(x) = a_0 + x \cdot p'(x)$ alakra hozzuk, ahol $p'(x)$ a csökkentett fokszámú polinom.
3. A **csokkent**/2 csökkenti egy polinom fokszámát;
4. rekurzió a polinom és az elemlista szerint is.

```
lPoli(Poli,[Fej|Mar],[VFej|VMar]):-
    egyPoli(Poli,Fej,VFej),
    lPoli(Poli,Mar,VMar),
    !.
lPoli(_,[],[]).

egyPoli([[0,A]|Mar],X,Ert):-
    csokkent(Mar,MarCs),
    egyPoli(MarCs,X,MErt),
    Ert is A + X * MErt,
    !.
egyPoli([[P,A]|Mar],X,Ert):-
    P>0,
    csokkent([[P,A]|Mar],Csokk),
    egyPoli(Csokk,X,Ert2),
    Ert is X*Ert2,
    !.
egyPoli([],_,0).
```

Feladat: írjunk predikátumot, mely egy listának törli minden k-adik elemét: `torol(2,[a,b,c,d,e],ListaKi)` válasz: `ListaKi=[a,c,e]`

A feladatot a következőképpen végezzük el:

- A listából kivesszük az első $K-1$ elemet:

```
elsoK1(K,Lista,[],Lista):-
    K<2.
elsoK1(K,[Fej|Mar],[Fej|Mar2],Mar3):-
    K > 1,
    K1 is K-1,
    elsoK1(K1,Mar,Mar2,Mar3).
```

- A maradékra meghívjuk rekurzívan a kódot:

```
torol(K,Lista,Eredm):-
    elsoK1(K,Lista,Elsok,[_F | Marad]),
    torol(K,Marad,Marad2),
    append(Elsok,Marad2,Eredm).
torol(_K,[],[]).
```

Eredmény: `torol(2,[1,2,3,4],V).`-re `V=[1,3]`, de `torol(2,[1,2,3],V).`-re a válasz `false`.

Megoldás: vágások, illetve az üres lista külön kezelése:

```
torol(_K,[],[]):-
    !.

torol(K,Lista,Eredm):-
    elsoK1(K,Lista,Elsok,[_F | Marad]),
    torol(K,Marad,Marad2),
    append(Elsok,Marad2,Eredm),
    !.

torol(K,Lista,Elsok):-
    elsoK1(K,Lista,Elsok,[ ]).
```

Az Előadások Témái

- Imperatív/deklaratív programok, Prolog alapok
- Listák, listakezelés
- Aritmetikai műveletek és operátorok
- Vágások használata, negáció
- Gyűjtő-predikátumok Prolog-ban, összegzések
- Vég-rekurzió optimalizálás
- Dinamikus predikátumkezelés; input-output prolog-ban
- Kifejezések dekompozíciója
- Rendezések, kereső algoritmusok
- Funkcionális programozás bevezető, történelem
- A Haskell programnyelv elemei, példaprogramok
- Algebrai típusok, típusosztályok
- Magasabb-rendű függvények, Map-Reduce, rendezések
- Input-output
- *(opc) Algebrai struktúrák 2: Monádok stb*
- Ismétlések és kérdések megvitatása

Adminisztra ... ... trívia

link: Canvas learning management system (LMS)

Régi honlap: `http://www.cs.ubbcluj.ro/~csatol/log_funk`

Vizsga (félév elején) – lásd SYLLABUS!

Labor (40%) + Parciális (25%) + Kollokvium (35%)

Laborgyakorlatok:

1. 2 Prolog feladatcsoport **20%**
2. 2 Haskell feladatcsoport **20%**
3. $\aleph_0$ feladat **első n beküldőnek**

Feladat: állítsuk elő egy listának minden elemét egyszer tartalmazó változatát.

```
remD([], []).

remD( [Fej| Mar], Res ) :-
    member(Fej, Mar ),
    remD(Mar, Res ).

remD([Fej| Mar], [Fej| Res] ) :-
    remD(Mar, Res ).
```

A backtracking okán minden ág végrehajtódik.

Feladat: állítsuk elő egy listának minden elemét egyszer tartalmazó változatát.

```
remD([], []).

remD( [Fej| Mar], Res ) :-
    member(Fej, Mar ),
    remD(Mar, Res ).

remD([Fej| Mar], [Fej| Res] ) :-
    remD(Mar, Res ).
```

A backtracking okán minden ág végrehajtódik.

Eredmény:

```
remD([1,1,2,2],L).
L=[1,2]
L=[1,2,2]
L=[1,1,2,2]
```

Megoldás a dedukciós fa vágása.

Feladat: állítsuk elő egy listának minden elemét egyszer tartalmazó változatát.

```
remD([], []).

remD( [Fej| Mar], Res ) :-
    member(Fej, Mar ),
    remD(Mar, Res ).

remD([Fej| Mar], [Fej| Res] ) :-
    remD(Mar, Res ).
```

A backtracking okán minden ág végrehajtódik.

Eredmény:

```
remD([1,1,2,2],L).
L=[1,2]
L=[1,2,2]
L=[1,1,2,2]
```

Megoldás: a dedukciós fa vágása.

Feladat: állítsuk elő egy listának minden elemét egyszer tartalmazó változatát.

```
remD([], []).

remD( [Fej| Mar], Res ) :-
    member(Fej, Mar ),
    remD(Mar, Res ).

remD([Fej| Mar], [Fej| Res] ) :-
    remD(Mar, Res ).
```

A backtracking okán minden ág végrehajtódik.

Eredmény:

```
remD([1,1,2,2],L).
L=[1,2]
L=[1,2,2]
L=[1,1,2,2]
```

Megoldás: a dedukciós fa vágása.

Vágások a remD egyértelművé tételéhez

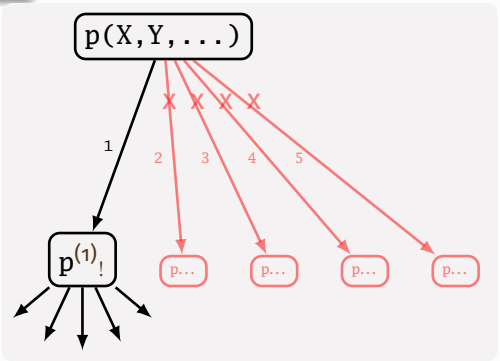
```
remD([], []).

remD([F| Tail], Res):-
    member(F, Tail),
    !,
    remD(Tail, Res).

remD([F| Tail], [F| Res]):-
    remD(Tail, Res).
```

- Ha a Prolog túljut a felkiáltójel előtti – member – kódon, akkor a következő ágakat nem hajtja végre.
- A vágás az adott szintre vonatkozik.
- Egyértelműsíti a megoldást.

Vágások a levezetési fában

Vágások

Megszüntetik a vágásjel – ! – után szereplő elágazásokat a levezetési fában.



Vágások a levezetési fában II

Tekintsük a következő kódot:

```
1 m(X,Y,X):-
    X<Y,
    !.

  m(_,Y,Y).
```

Vágások a levezetési fában II

Tekintsük a következő kódot:

```
m(X,Y,X):-
  X<Y,
  !.

m(_,Y,Y).
```

1. Mit „számít ki” a predikátum?
2. Az első esetben – ha `X<Y` – a harmadik argumentum megegyezik az `X` értékével;
3. Ha nem, akkor a harmadik argumentum az `Y` értékével egyezik. $\Rightarrow$ minimumot számol.

Mi történik a következő lekérdezéssel: `m(2,5,5)`.

Vágások a levezetési fában II

Tekintsük a következő kódot:

```
m(X,Y,X):-
  X<Y,
  !.

m(_,Y,Y).
```

1. Mit „számít ki” a predikátum?
2. Az első esetben – ha `X<Y` – a harmadik argumentum megegyezik az `X` értékével;
3. Ha nem, akkor a harmadik argumentum az `Y` értékével egyezik. $\Rightarrow$ minimumot számol.

Mi történik a következő lekérdezéssel: `m(2,5,5).`

Vágások a levezetési fában II

Tekintsük a következő kódot:

```
m(X,Y,X):-
  X<Y,
  !.

m(_,Y,Y).
```

1. Mit „számít ki” a predikátum?
2. Az első esetben – ha `X<Y` – a harmadik argumentum megegyezik az `X` értékével;
3. Ha nem, akkor a harmadik argumentum az `Y` értékével egyezik. $\Rightarrow$ minimumot számol.

Mi történik a következő lekérdezéssel: `m(2,5,5)`.

Vágások a levezetési fában III

`m(2,5,5)` kérdésre a rendszer válasza **`True`**.

- a vágások nem segítenek akkor, ha minden érték ismert;
- a rendszer illeszti a második esetet, ugyanis nem ért el a vágásig;
- A vágásokat jobb csak konstruktív levezetéseknél használni.

Vágások a levezetési fában IV

- Módozatok a javításra:
  - felírjuk a feltétel komplementerét (nem hasznos, csökken a kód minősége):

```
m(X,Y,X):-
  X<Y.
m(X,Y,Y):-
  X=>Y.
```

  - teszteljük azt, hogy a változónk ne legyen kötött (nem tudjuk tesztelésre használni):

```
1 m(X,Y,X):-
    X<Y,
    !.
  m(X,Y,Z):-
    var(Z),
6   Z=Y.
```

- Általános megoldás nincs – a programozó kell használja a neki megfelelőt.

Ulle Endriss jegyzete.

Adjunk egy elemet egy halmazhoz:

```
ad(E,Halmaz,Halmaz):-
  member(E,Halmaz),
  !.

ad(E,Halmaz,[E|Halmaz]).
```

- Hozzáadja az `E`-t, ha nem tagja már a `Halmaz`nak.
- Ezért kerül a hozzáadó rész a `member` és `!` után.
- Szintén nem teljes: igaz a `ad(alma,[korte,alma],[alma,korte,alma])` predikátum.

Vágások

Összefoglaló

Ulle Endriss jegyzete.

- A vágások megkönnyítik a programozó munkáját;
- A beépített visszalépéses – backtracking – algoritmust nagyon fel tudják gyorsítani.
- A programozó csak a **MIT**-tel kell törődjön, a feladat hogyan-ját a rendszer végzi.

Írjuk meg a halmazba egy elemet beszúró predikátumot.

```
ad(E,Halmaz,[E|Halmaz]):-
   \+ member(E,Halmaz),
   !.

ad(_,Halmaz,Halmaz).
```

- a `\+` a tagadást jelöli, ugyanaz, mint a `neg/1`;
- Egy megvalósítás:

```
neg(Kif):-
   call(Kif),
   !,fail.

neg(_).
```

Példa:
Keressük az ős–öket – azon személyeket, akiknek nincs szülőjük.

Egy megoldás:

```
os(V) :-
    \+ gyereke(V,_).
```

- `os(adam)`, `os(zita)` igaz.
- Azonban:
  - `os(istike)` is igaz;
  - `os(Valaki)` – fail.

Miért?

```
gyereke(ferenc,adam).
gyereke(ferenc,eva).
gyereke(marika,adam).
gyereke(marika,eva).
gyereke(peter,marika).
gyereke(peter,jozsef).
gyereke(julia,ferenc).
gyereke(julia,agnes).
gyereke(endre,julia).
gyereke(rozi,julia).
gyereke(endre,pali).
gyereke(sara,endre).
gyereke(sara,zita).
gyereke(zsolt,endre).

% utód = tranzitív lezárt
utod(Utod, Os) :-
    gyereke(Utod,Os).
utod(Utod, Os) :-
    gyereke(Utod,Utodszulo),
```

Negáció Prolog-ban III

- `os(istike)` igaz, mert `istike` nincs a családfában;
  - Nagyon sok atom létezik tehát, mely teljesíti a feltételeket.
  - A rendszer pozitív választ ad;
  - Nem tudunk semmit `istike` személyéről;
  - A tagadás komplementer halmazművelet.

Komplementer

A tények/szabályok negációja nagyon sok eredményt jelent.

Negáció Prolog-ban IV

- `os(Valaki)` eredménye fail.
- Nincs egy `Valaki` sem, aki ős lenne?
- Túl sok lehetséges eset van;
- A rendszer nem válaszol.

`fail` Prolog-ban

A Prolog-ban a `fail` eredmény azt jelenti, hogy a rendszer nem tud válaszolni.

Nem jelenti azt, hogy a kijelentés nem igaz.

A programozónak kell a helyességről gondoskodni.

Negáció Prolog-ban V

- A `member(E,[2,3,4])` lekérdezés eredményeként az `E` felveszi rendre a listában szereplő értékeket.
- Milyen értékeket kellene az `E` változónak felvenni ahhoz, hogy a `\+member(E,[2,3,4])` igaz legyen?
  - Válasz: nagyon sok lehetséges válasz van;
  - A rendszer válasza: `false`.

Az `os(Valaki)` predikátum egy specifikációja:

```
os(Valaki):-
    gyereke(_,Valaki),
    \+ gyereke(Valaki,_).
```

Prolog negáció

- A zárt világ feltételezés az alapja;
- A komplementer eseménytér nagysága miatt nagyon szűk az érvényessége;
- A false nem azt jelenti, hogy a megoldás hamis, hanem azt, hogy nem bizonyítható.
- A programozónak kell gondoskodni arról, hogy a futásnál minden változója kötött legyen.

Gyűjtőpredikátumok általánosan I

- Megvalósítják „az összes olyan” típusú „lekérdezést” Prolog-ban.

az adatbázis-terminológia szándékos.

- A „lekérdezés” folyamata:
  - „Származtatunk” – szabályokkal – új relációkat;
  - „Lekérdezzük” – **`findall`** – az eseteket;
  - Összegezzük az eredményt majd megjelenítjük.
- Hasonlóság az SQL-lel:
  - Az adatokban tárolt információk megjelenítése, összegzések létrehozása.
- Különbözőség az SQL-től:
  - Fa, illetve gráfstruktúrákat is fel tudunk dolgozni;
  - Új szabályok kijelentésével rekurzív és cirkuláris dependenciákat is tudunk használni.

Gyűjtőpredikátumok általánosan II

Gyüjtőpredikátumok:

- **`findall(Elem,CélPred,Lista)`** - az összes **`Célpred`**-et teljesítő **`Elem`**-et visszatéríti a **`Lista`** listába;
- **`bagof(Elem,CélPred,Lista)`** - a **`findall`**-hoz hasonlóan működik, de a **`CélPred`** változó argumentumai szerint csoportosít – klaszterez.
- **`setof(Elem,CélPred,Lista)`** - működése megegyezik a **`bagof`**-éval, azonban a **`Listá`**ból a duplikátumokat kiszűri.

A findall gyűjtőpredikátum I

Példa: permutáció

```
perm([],[]).

perm(Lista,[Elso|PMar]) :-
    select(Elso,Lista,Mar),
    perm(Mar,PMar).
```

- A levezetési fa összes megoldását ki tudjuk íratni a „vagy” – „;” operátor – segítségével;
- Azonban szeretnénk a megoldásokat tárolni;
- A Prolog a gyűjtőpredikátumok segítségével valósítja meg a tárolást.

A **findall** gyűjtőpredikátum I

Példa: permutáció

```
perm([],[]).

perm(Lista,[Elso|PMar]) :-
    select(Elso,Lista,Mar),
    perm(Mar,PMar).
```

```
?- perm([duna,tisza,maros],PL).
PL = [duna, tisza, maros];
PL = [duna, maros, tisza];
PL = [tisza, duna, maros];
PL = [tisza, maros, duna];
PL = [maros, duna, tisza];
PL = [maros, tisza, duna];
false.
```

- A levezetési fa összes megoldását ki tudjuk íratni a „vagy” – „;” operátor – segítségével;
- Azonban szeretnénk a megoldásokat tárolni;
- A Prolog a gyűjtőpredikátumok segítségével valósítja meg a tárolást.

A `findall` gyűjtőpredikátum II

- Szeretnénk egy lista összes permutációját tárolni.
- Eszköz a `findall`/3 predikátum:

```
findall(Egy,perm(Egy,[duna,tisza,maros]),Lista).
```

- A `findall(Elem,pred(...),Lista)` predikátum:
  - a `pred(...)` teljes levezetési fáját bejárja és
  - az `Elem` atom (vagy minta) összes előfordulását a
  - `Lista`ba menti, mely a harmadik argumentum.

Például `findall(Z,between(1,25,Z),L)` generálja az 1 és 25 közötti számok listáját.

A **bagof** gyűjtőpredikátum

I

- A **findall(Os,utod(julia,Os),Osok)** eredménye **Osok=[ferenc,agnes,adam,eva]**
- A **findall(Os,utod(_,Os),Osok)** listába tesz minden őst (többszörösen).

```
gyereke(julia,agnes).
gyereke(endre,julia).
gyereke(rozi,julia).
gyereke(endre,pali).
gyereke(sara,endre).
gyereke(sara,zita).
gyereke(zsolt,endre).

% utód = tranzitív lezárt
utod(Utod, Os) :-
    gyereke(Utod,Os).
utod(Utod, Os) :-
    gyereke(Utod,Utodszulo),
    utod(Utodszulo,Os).
```

A bagof gyűjtőpredikátum I

- A `findall(Os,utod(julia,Os),Osok)` eredménye `Osok=[ferenc,agnes,adam,eva]`
- A `findall(Os,utod(_,Os),Osok)` listába tesz minden őst (többszörösen).

```
gyereke(julia,agnes).
gyereke(endre,julia).
```

- A `bagof`/3 klaszterezi az eredményeket.
- A fennmaradó szabad változókat illeszti és kiszámolja az azokhoz tartozó eseteket,
- Visszalépésnél veszi a következőt,
- A `bagof(Os,ose(zsolt,Os),Osok)`==`findall`/3 (duplikátumok szűrése után).

A **bagof** gyűjtőpredikátum II

```
gyereke(ferenc,adam).
gyereke(ferenc,eva).
gyereke(marika,adam).
gyereke(marika,eva).
gyereke(peter,marika).
gyereke(peter,jozsef).
gyereke(julia,ferenc).
gyereke(julia,agnes).
gyereke(endre,julia).
gyereke(rozi,julia).
gyereke(endre,pali).
gyereke(sara,endre).
gyereke(sara,zita).
gyereke(zsolt,endre).

% utód = tranzitív lezárt
utod(Utod, Os) :-
    gyereke(Utod,Os).
utod(Utod, Os) :-
    gyereke(Utod,Utodszulo),
    utod(Utodszulo,Os).
```

A bagof gyűjtőpredikátum II

- A bagof(Os,ose(zsolt,Os),Osok)==findall/3 (duplikátumok szűrése után),
- A bagof(Os,ose(Utod,Os),Osok) esetén de Osok lista készül minden Utódra:

```
Utod = ferenc,
Osok = [adam, eva];
Utod = julia,
Osok = [ferenc, agnes, adam, eva];
Utod = marika,
Osok = [adam, eva];
...
```

- Hatékony összegző.
- fail, ha nem kielégíthető.

A `setof` gyűjtőpredikátum

`setof`/3

- Használata azonos a `bagof`/3 predikátummal,
- Az eredményeket mindig rendezi a `sort`/2 predikátum segítségével.
- Nem fut le – `fail` –, ha a célpredikátum nem kielégíthető.

`Utod^ose(Utod,Os)`

A predikátumban kijelentjük, hogy az `Utod` változó nem érdekel.

Adottak a következő predikátumok:

- `products( Id, Name, Subcat, Id_subcat, Cat, Id_cat, supp_id,prod_src_id,d_prod_from)`
- `costs(Id, timd_from,promo_id,channel_id, unit_cost,unit_price)`
- `sales(prod_id,cust_id,time_id,channel_id, promo_id,quant_sold,amount_sold)`

Feladat: Számítsuk ki bevételt adott időperiódusban, termékosztályok szerinti bontásban.

Lépések:

- a termékek szerinti bevétel kiszámítása;
- termékek bevételének összegzése osztályok szerint.

A **sales**/7 predikátumból a termékekhez tartozó jövedelmek meghatározása (tároljuk a termékosztályt is):

```
%-- SALES szurese
t_ss(ProdId,Menny,Ar):-
    sales(ProdId,_,_,_,_,Menny,Ar).

%-- a ProdId termek teljes bevetele
termOsszeg(ProdId,TBev):-
   bagof(M/Ar,t_ss(ProdId,M,Ar),MAr_L),
   feldolgoz(MAr_L,0,TBev).
%-- a termékek szerint osztályoz
feldolgoz([Menny/Ar| Marad], Temp, TAr):-
   Temp1 is Temp + Ar * Menny,
   feldolgoz(Marad, Temp1, TAr),
   !.
feldolgoz([], TAr, TAr).
```

A **termOsszeg**(**ProdId**,**TBev**) a **ProdId** termékhez tartozó teljes bevételt számítja ki (kódolja az osztályt).

A **findall** egy hosszú listába menti a **CatId**/**ProdId**/**TAr** értékeket, melyeket az **osszegOszt**/3 predikátum összegez csoportok szerint:

```
osztalyBevetel(TermL, OsztL) :-
   findall(Id/Bev, termOsszeg(Id,Bev), TermL),
   osszegOszt(TermL, [], OsztalyL).

osszegOszt([CatId/_/Bev | Mar], Temp, OsztalyL):-
   select(CatId/RegiBev, Temp, TMar),
   UjBev is Bev + RegiBev,
   osszegOszt(Mar, [CatId/UjBev | TMar], OsztalyL),
   !.
osszegOszt([CatId/_/Bev | Mar], Temp, OsztalyL):-
   osszegOszt(Mar, [CatId/Bev | Temp], OsztalyL),
   !.
osszegOszt([], OsztLista, OsztLista).
```

Megjelenítő kód:

```
% megjelenites
kiirat:-
  osztalyBevetel(TermekL,OsztalyL),
  kiiratTermek(TermekL),
  nl,writeln('Osztalyok:'),
  kiiratOsztaly(OsztalyL).

kiiratTermek([_/ProdId/Menny|MaradL]):-
  products(ProdId,Product,_,_,_,_,_,_),
  KMenny is round(Menny),
  format('A "~w" termek forgalma: ~d\n',[Product,KMenny]),
  kiiratTermek(MaradL),
  !.
kiiratTermek([]):-
  nl.

kiiratOsztaly([CatId/Menny|MaradL]):-
  products(_,_,_,_,Osztaly,CatId,_,_),
  KMenny is round(Menny),
  format('A "~w" osztaly forgalma: ~d\n',[Osztaly,KMenny]),
  kiiratOsztaly(MaradL),
  !.
kiiratOsztaly([]):-
```

Eredmény:

```
A "d_adventures_with_numbers" termék forgalma: 175564
A "d_extension_cable" termék forgalma: 60713
A "d_xtend_memory" termék forgalma: 366858

Osztályok:
Az "electronics" osztály forgalma: 14597516
A "photo" osztály forgalma: 17961866

A "peripherals_and_accessories" osztály forgalma: 31163988
A "software_other" osztály forgalma: 13834856
A "hardware" osztály forgalma: 20647606
```

Gyűjtőpredikátumok

Összefoglaló

- A **findall**/3 összegyűjti egy levezetési fában az összes alternatív választ;
- A **bagof**/3 a részlegesen parametrizált célpredikátumok esetén a megoldásokat csoportosan téríti vissza;
- A **setof**/3 a visszatérített listákból szűri a duplikátumokat, az eredmény halmazként is használható.
- Amennyiben egy változó nem érdekel, azt explicit jelölnünk kell: a **bagof**(**Elem**,**Temp**^**pred**(**Temp**,**Elem**),**Lista**) predikátumban a **Temp** „nem számít”.

Pénzváltó

I

Feladat: Adott N RON-t hányféleképpen tudunk kifizetni, ha végtelen mennyiségű érme áll rendelkezésre adott $[E1, E2, \ldots, Ek]$ érmékből.

Pénzváltó

Feladat: Adott N RON-t hányféleképpen tudunk kifizetni, ha végtelen mennyiségű érme áll rendelkezésre adott [E1,E2,...,Ek] érmékből.

```
ermek(N,[E],[K]):-
    K is floor(N/E),
    N =:= K*E.

ermek(N,[E|EMar],[K|KMar]):-
    Max is floor(N/E),
    between(0,Max,K),
    N1 is N - E*K,
    ermek(N1,EMar,KMar).
```

Megj: figyeljük meg a generatív megoldásmenetet!

Pénzváltó

I

Feladat: Adott N RON-t hányféleképpen tudunk kifizetni, ha végtelen mennyiségű érme áll rendelkezésre adott [E1,E2,...,Ek] érmékből.

```
ermek(N,[E],[K]):-
    K is floor(N/E),
    N =:= K*E.

ermek(N,[E|EMar],[K|KMar]):-
    Max is floor(N/E),
    between(0,Max,K),
    N1 is N - E*K,
    ermek(N1,EMar,KMar).
```

Megj: figyeljük meg a generatív megoldásmenetet!

Lekérdezés:

```
findall(L,ermek(142,[1,2,5,10,50,100],L),R),length(R,N).
```

Pénzváltó

I

Feladat: Adott N RON-t hányféleképpen tudunk kifizetni, ha végtelen mennyiségű érme áll rendelkezésre adott [E1,E2,...,Ek] érmékből.

```
ermek(N,[E],[K]):-
    K is floor(N/E),
    N =:= K*E.

ermek(N,[E|EMar],[K|KMar]):-
    Max is floor(N/E),
    between(0,Max,K),
    N1 is N - E*K,
    ermek(N1,EMar,KMar).
```

Megj: figyeljük meg a generatív megoldásmenetet!

Lekérdezés:
`findall(L,ermek(142,[1,2,5,10,50,100],L),R),length(R,N).` Válasz: 7890.

Utolsó hívás optimalizáció I

- A Prolog rekurzív hívásokat támogat;
- A rövid és könnyen érthető kódban általában sok a rekurzió;
- Rekurzióval gyakran találkozunk az axiomatikus fogalmazásnál is.
- A listák összefűzésénél axiómák = Prolog kód:

```
% append(ElsőL, MásodikL, EredL)

append([],L,L).

append([F|Mar],MasL,[F|AppMar]):-
    append(Mar,MasL,AppMar).
```

Második példa a `select`:

```
% select(Elem, Lista, MaradékLista )

select(X, [X| Mar], Mar ).

select(X, [Fej| Mar], [Fej| SelMar] ):-
   select(X, Mar, SelMar).
```

A példák jellemzői:

- Hangsúlyosan rekurzívak;
- A rekurzív hívás után nincs végrehajtandó kód;

Utolsó hívás optimalizáció III

Az utolsó hívás optimalizáció esetében a predikátumok futás előtt automatikusan átalakíthatóak:

- A függvényhívások elhagyhatók, ezáltal nem kell explicit helyet foglalni a lokális paramétereknek és a függvény argumentumainak;
- Az átalakítást a rendszer végzi el, kizárva a hibákat;
- A megfelelő – tail-recursive – formába nekünk kell hozni a predikátumot.

Utolsó hívás optimalizáció IV

Példák:

```
% hossz(Lista, Hossz)

hossz([],0).

hossz([_|Mar],H) :-
   hossz(Mar,HM),
   H is 1 + HM.
```

- A hossz definíciója nem teszi lehetővé az utolsó hívás optimalizációt;
- A segédváltozóval kiegészített hossz_ predikátum már optimalizálható.

Példák:

```
% hossz(Lista, Hossz)

hossz([],0).

hossz([_|Mar],H) :-
   hossz(Mar,HM),
   H is 1 + HM.
```

```
% hossz(Lista, Hossz)
hossz(Lista,H) :-
   hossz_(Lista,0,H).

hossz_([],H,H).
hossz_([_|Mar],Temp,Hossz):-
   T1 is Temp+1,
   hossz_(Mar,T1,Hossz).
```

- A hossz első definíciója nem teszi lehetővé az utolsó hívás optimalizációt;
- A segédváltozóval kiegészített **hossz_** predikátum már optimalizálható.

Írjunk optimalizálható Prolog kódot:

1. Számítsuk ki egy számlista átlagát.
2. Számítsuk ki egy lista legnagyobb elemét: a **maxim(Lista,MEl)** esetén a Lista legnagyobb értéke az **MEl**.
3. Számítsuk ki egy lista két legkisebb elemét.
4. Számítsuk ki egy szám faktoriálisát.

A tesztelésre használjuk a **randd**(2000000,50000000,**L**), **maxim**(**L**,**MAX**). kódot.

```
randd(0,_,[]) :-
    !.
randd(N,D,[R|Mar]) :-
    R is 1+random(D),
    N1 is N-1,
    randd(N1,D,Mar).
```

Az Előadások Témái

- Imperatív/deklaratív programok, Prolog alapok
- Listák, listakezelés
- Aritmetikai műveletek és operátorok
- Vágások használata, negáció
- Gyűjtő-predikátumok Prolog-ban, összegzések
- Vég-rekurzió optimalizálás
- Dinamikus predikátumkezelés; input-output prolog-ban
- Kifejezések dekompozíciója
- Rendezések, kereső algoritmusok
- Funkcionális programozás bevezető, történelem
- A Haskell programnyelv elemei, példaprogramok
- Algebrai típusok, típusosztályok
- Magasabb-rendű függvények, Map-Reduce, rendezések
- Input-output
- *(opc) Algebrai struktúrák 2: Monádok stb*
- Ismétlések és kérdések megvitatása

Adminisztra ... ... trívia

link: Canvas learning management system (LMS)

Régi honlap: `http://www.cs.ubbcluj.ro/~csatol/log_funk`

Vizsga (félév elején) – lásd SYLLABUS!

Labor (40%) + Parciális (25%) + Kollokvium (35%)

Laborgyakorlatok:

1. 2 Prolog feladatcsoport **20%**
2. 2 Haskell feladatcsoport **20%**
3. $\aleph_0$ feladat **első n beküldőnek**

Redundáns kódok

Feladat:
Írjuk meg a `Fibo(N,FN)` predikátumot, mely az `N`-edik Fibonacci számot feleltei meg az `FN`-nek.

```
fibo(0,1).
fibo(1,1).
fibo(N,FN) :-
    N > 1,
    N1 is N - 1, fibo(N1,FN1),
    N2 is N - 2, fibo(N2,FN2),
    FN is FN1 + FN2.
```

A predikátum nem hatékony

A `fibo(7,F5)` kiértékeléséhez meg kell találnia az `{F4,F3}` értékeket, ehhez pedig az `{F3,F2,F2,F1}` értékeket.

Az assert használata

- A **fibo/2** predikátum nem „emlékszik” arra, hogy milyen értékeket számolt már ki,
- A predikátumot módosítjuk a következők szerint:
  - Jelezzük a Prolog-nak, hogy a **fibo/2** dinamikus predikátum;
  - Vágásokat vezetünk be az első két kijelentésbe;
  - Miután kiszámolunk egy értéket, ezt predikátummá alakítjuk és megjegyezzük.
- Következmény: a már kiszámított értékeket nem fogja újraszámolni.

A dinamikus fibo/2

Az új predikátum:

```
:- dynamic(fib/2).

% fibo(N,NEDIK) az asserta kodolassal
fibo(0,1) :- !.
fibo(1,1) :- !.
fibo(N,Ert):-
    N>1,
    N1 is N-1,
    fibo(N1,Ert1),
    N2 is N-2,
    fibo(N2,Ert2),
    Ert is Ert1+Ert2,
    asserta(fibo(N,Ert):-!).
```

A `listing(fibo)` predikátummal megtekinthetjük a prolog reprezentációt.

A dinamikus fibo/2

A `fibo(22,F22)` lekérdezés után a `listing(fibo)` kódja:

```
:- dynamic fibo/2.

fibo(22, 28657) :- !.
fibo(21, 17711) :- !.
fibo(20, 10946) :- !.
fibo(19, 6765) :- !.
fibo(18, 4181) :- !.
fibo(17, 2584) :- !.
fibo(16, 1597) :- !.
fibo(15, 987) :- !.
fibo(14, 610) :- !.
fibo(13, 377) :- !.
fibo(12, 233) :- !.
fibo(11, 144) :- !.
fibo(10, 89) :- !.
fibo(9, 55) :- !.
```

```
fibo(8, 34) :- !.
fibo(7, 21) :- !.
fibo(6, 13) :- !.
fibo(5, 8) :- !.
fibo(4, 5) :- !.
fibo(3, 3) :- !.
fibo(2, 2) :- !.
fibo(0, 1) :- !.
fibo(1, 1) :- !.
fibo(A, D) :-
  A>1,
  B is A+ -1,
  fibo(B, E),
  C is A+ -2,
  fibo(C, F),
  D is E+F,
  asserta((fibo(A, D):-!)).
```

- Az **asserta** vagy **assertz** parancsokat használhatjuk új tények vagy szabályok bevitelére;
- Az új tényeket fel lehet használni a rendszer további működésénél;
- A predikátumokat a **retract** paranccsal töröljük a rendszerből;
- A **retractall** predikátum egy mintát teljesítő *összes* szabályt töröl (például az `retractall(fibo(_,_))` minden predikátumot töröl).

Input-output műveletek

- A Prolog konzol-alapú nyelv:
  - a parancs-ablak szolgál a rendszer be- és kimeneteként;
  - bemenet: `read` és `read_term`;
  - kimenet: `write`, `writeln`, `nl`;
  - formázott kimenet a `format(String,ParList)` paranccsal (`help format` a további információkért);
- Tudjuk tesztelni a predikátumokat a konzol-ban.

read – write

Példa read-write-ra:

```
?- read(X),writeln(X),fail.
|: valtozo.
valtozo
false.
```

Predikátum-szintű beolvasások

A **read** predikátum az argumentumot illeszti a beolvasott eredményhez, lehetővé téve argumentumok ellenőrzését:

```
?- read([X|Mar]).
|: [1,2,3,4,5].
X = 1,
Mar = [2, 3, 4, 5].
```

```
1 ?- read([X|Mar]).
|: mas.
false.
```

(ügyeljünk a pontra a sorok végén)

„Atomi” beolvasó műveletek

- `get_char(C)` – beolvas egy karaktert a bemenetről és illeszti a C-ban adott bemenő mintával.
- `get_byte(C)` – egy bitet, a `get_char` hoz hasonlóan.
- `get_code(C)` – egy számot illeszt a „`C`”-hez.
- `peek_code(C)` – megnézi a következő egységet az input-ból, anélkül, hogy kivenné (lásd még `peek_byte(C)`, `peek_char(C)`).
- speciális kódok:
  - `end_of_file` – atom, mely a file-végén van.
  - `end_of_line` – atom a sorvég tesztelésre.

Kétargumentumos beolvasó predikátumok

A kétargumentumos változatnál az első argumentum a bemenő file vagy stream.

„Atomi” kiíró műveletek

- `put_char(C)` – kiír egy karaktert a kimenetre.
- `put_byte(C)` – kiír egy bitet a konzol-ra.
- `put_code(C)` – egy karakter-kódot ír ki.
- `flush_output` – a felgyűlt cache-ben levő karaktereket kiírja, kiürítve a cache-t.
- a speciális kódok – a beolvasásnál ismertetett kódok – használhatók.

Kétargumentumos kiíró predikátumok

A kétargumentumos változatnál az első argumentum a file vagy stream.

Edinburgh IO

- **see**(**Atom**) – megnyitja az Atom-ban található file-ot olvasásra – stream-ként viselkedik;
- minden beolvasási művelet át lesz irányítva;
- **seen** – bezárja a megnyitott stream-et és a bemenetet a konzol-ról várja;
- **tell**(**Atom**) – megnyitja írásra az Atom-ot;
- minden kiírási művelet át lesz irányítva;
- **told** – bezárja a megnyitott stream-et és a kimenetet a konzol-ra irányítja át;

Mini-interpreter

I

```
olm = 9
bl = @add olm 2
leq0 = @add bl 4
re0 = @add bl 9
ytc8 = @add re0 6
lpe = 2
st2 = 10
ili3 = @add ytc8 6
olb3 = 3
qiu9 = 3
ne8 = 3
ecf6 = @add ytc8 1
by8 = 8
nhf4 = 10
ajv = @add ytc8 7
nco = @add olm 4
tr3 = 9
zdn = @add leq0 5
tw1 = @add nhf4 3
hw5 = 3
ll0 = @add ytc8 10
mil = 10
opj = 5
lj = @add ytc8 5
mj4 = 3
xs5 = 10
wsi4 = 5
zn = @add tr3 5
cli2 = @add hw5 6
eas = 10
ej = 2
egv = @add mil 10
mu6 = 4
xzz = 3
tt3 = 9
zia8 = 7
gfc = 7
gnz = @add xzz 1
ao = @add lj 8
adt = @add eas 9
gqu = @add gnz 9
fo = @add eas 7
sy8 = @add opj 8
ps8 = 8
sl8 = 4
bkr = @add sl8 3
zbw = 10
nf4 = @add xzz 7
dnk = 10
mis0 = @add zia8 4
```

```
pqv4 = 2
ju6 = 6
we = @add fo 5
up = 1
rqn6 = 2
ys = 5
kzm = @add gfc 8
wjx = @add ao 3
plz = @add wjx 9
arc8 = @add dnk 3
nb6 = 1
qc = 3
dvc4 = 5
ksn3 = 4
xvm5 = @add dnk 3
rj = 2
gb = 6
hy0 = 3
tn8 = 1
zs6 = @add dvc4 8
bhg = @add qc 3
ifm1 = @add bhg 2
jx = 8
gf7 = @add rj 2
swt5 = 9
xhr = @add ksn3 8
qbz = 8
re8 = 5
bsj = 8
ko0 = @add nb6 8
jh4 = @add qc 3
eyb = 8
on1 = 2
jlz7 = @add gf7 3
lj = 6
ck1 = @add swt5 9
wjc = @add xhr 9
gc = 6
gn6 = @add jx 6
sz = 7
xkk6 = 2
xe = 3
xjw7 = @add ck1 10
hv8 = 2
dp1 = @add gc 10
ecw = 3
rd9 = @add sz 3
yh9 = 6
esr = @add ecw 4
xqc = 4
```

```
@writeln xqc
va9 = @add xkk6 5
ng = @add ck1 6
kgy = @add wjc 2
ezx4 = 9
ao8 = @add gn6 6
at8 = 8
aq2 = 6
hcr = @add yh9 5
kd = 8
kq3 = @add at8 10
ae = @add hcr 5
lvi = 5
khs = @add kq3 4
nt0 = 7
mwf6 = @add kgy 7
ppt = 6
iet6 = 10
nn = 9
iwr5 = @add aq2 3
uzm = @add ae 10
io5 = @add mwf6 1
vop = @add kd 7
lir6 = @add mwf6 4
me2 = @add aq2 2
bvr2 = @add nt0 9
fyh = @add lir6 8
ptp0 = @add ppt 7
qwf2 = 9
ldq9 = 5
pzd = @add lvi 5
xvm1 = 1
xb = 5
fjc9 = 4
sx = @add fyh 6
mg5 = 8
tr = 10
hg2 = 7
pm = 2
bm4 = @add ldq9 2
cot5 = 10
uj = 10
fd = 7
zbx = @add me2 8
nll4 = 4
ew = @add xb 2
ref = @add fd 6
rgs6 = @add nll4 8
el6 = 2
ndp = @add sel6 9
```

```
bwr2 = 8
at = @add bm4 2
mlv = 4
dec1 = @add ref 3
won = 5
zi0 = 6
kh = 4
jt7 = @add dec1 9
ca9 = @add ndp 6
fbq = @add at 9
rme = @add rgs6 4
ia = 6
lqq = 2
ni1 = @add mlv 8
wdh = @add ia 7
wp = 1
xcr = 9
ayc7 = @add zi0 9
yhd = 9
ai2 = 5
tov = 6
dej2 = @add wdh 10
ehs = @add wp 4
qtp8 = @add ai2 8
hws3 = 3
lhe = @add jt7 8
brnl = 2
qiw4 = @add ia 5
xb3 = @add ayc7 9
zy2 = 6
phn6 = @add dej2 2
ono9 = 5
xsb = 6
tb9 = @add lhe 8
rk = 8
en = @add lhe 8
opc = 8
vf = 6
wf = 5
ydb = @add wf 6
hz = 1
mgo = @add qiw4 1
joy6 = @add wf 1
pck = 10
cnn1 = 8
gtu = @add joy6 4
zwb = @add hz 1
cqe6 = 7
sc8 = 10
xd6 = 9
```

```
hfo0 = 8
@writeln hfo0
df = @add xd6 4
buh2 = 8
dn = 5
vzg = @add pck 9
jr = 4
us7 = @add sc8 8
khp = 6
cf0 = 10
nqd9 = @add cqe6 7
ohi = @add sc8 10
hi = 8
ur2 = @add dn 2
ai9 = 7
rdq0 = 7
nvh4 = @add hfo0 10
jp0 = 5
fq = @add hi 9
tdq = 3
pt = @add dn 9
eq = @add nqd9 6
bx = @add eq 3
zt = @add hi 6
tln = 5
cda = @add nqd9 10
yup = 10
aac = 4
fa = 7
mp3 = @add tdq 8
tj9 = @add cda 8
yyo = @add bx 8
ggr = 7
fa3 = 7
vt7 = 2
ib = 2
sol = @add mp3 7
dl = @add bx 3
qs9 = @add zt 2
yqx = @add tln 3
xny = 6
hpe2 = @add yyo 7
lf = @add bx 6
wty = 10
ve5 = @add qs9 10
tjv = @add ggr 5
wta5 = 10
cz9 = @add sol 2
xn = 8
njs8 = @add yqx 1
```

Mini-interpreter II

http://www.sze.hu/~gtakacs/progver/2012.html

Feladat: hajtsuk végre a programot.

A program szintaxisa:

- értékadás vagy kiíratás;
- értékadás az „=” jellel történik;
- nem deklaráljuk a változókat, melyek az értékadásnál jönnek létre;
- az értékadás jobb oldalán
  - egész szám; vagy
  - `@add`-del jelölt összeadása a két értéknek, mely vagy konstans vagy változó.
- a `@writeln` utáni változót kiírjuk a kimenetre.

Mini-interpreter III

Lépések:

- A program beolvasása és átalakítása Prolog formába:
  - A bemenő `stream`-ből olvasás;
  - Egy sor beolvasása, majd atomokká alakítása;
- A program „értelmezése”:
  - Egy állapottér definíciója;
  - A konzol kimenetnek az inicializálása;
  - A sorok értelmezése, mint:
    - Állapottér-módosító utasítások;
    - Konzol-kimenet módosító utasítások.

Mini-interpreter I

```
% interpret(FileName) - ertelmezi a FileName-ban megadott 'kodot'
interpret(FileName,Res) :-
    open(FileName, read, InStream),     % megnyitjuk a file-t
    parse_lines(InStream,[],[],_,Res), % feldolgozzuk
    close(InStream).                    % bezarjuk

% ertelmezi a kodot soronkent
parse_lines(InStream, D0, O0, DF, OF) :-
    read_line_to_string(InStream,Line),       % beolvasunk egy sort
    Line \= end_of_file, !,                   % ha FILE_END, kilépünk
    split_string(Line, " ", "", StringList),  % felbontjuk a string-et
    interpret_line(StringList, D0, O0, D1, O1),
    parse_lines(InStream, D1, O1, DF, OF).
% a kilépésnél -- ha LINE == end_of_file -- vége a feladatnak
parse_lines(_, D0, O0, D0, OF):-
    reverse(O0,OF).
```

Mini-interpreter I

```
% interpret(FileName) - ertelmezi a FileName file-ban levo programot
interpret(FileName,Res) :-
    open(FileName, read, InStream),          % megnyitjuk a file-t
    parse_lines(InStream,[],[],_,Res),       % ertelmezzuk a sorokat
    close(InStream).

% ertelmezi a kodot soronkent
parse_lines(InStream, D0, O0, DF, OF) :-
    read_line_to_string(InStream,Line),      % beolvas egy sort
    Line \= end_of_file, !,                  % ha FILE_END, kilepunk
    split_string(Line, " ", "", StringList), % felbontjuk a string-et
    interpret_line(StringList, D0, O0, D1, O1), % ertelmezzuk a sort
    parse_lines(InStream, D1, O1, DF, OF).   % rekurzio
% a kilépésnél -- ha LINE == end_of_file -- vége a feladatnak
parse_lines(_, D0, O0, D0, OF):-
    reverse(O0,OF).
```

- Az **interpret(FileName,Res)** a file-ból olvassa az utasítások listáját.
  - A **parse_lines(S,InD,InV,FinD,FinV)** értelmezi a sorokat;
- A **split_string**/4 a sztring-et felbontja.
- Az **interpret_line**/4 a sztring-listát értelmezi.

Mini-interpreter II

```
% Mintaillesztesek a lehetséges értékek sorok szerint
interpret_line([SX, "=", SY], D0, Out, D1, Out) :-
    name(AX, SX),                % string -> atom konverziok
    name(AY, SY),                % masik
    get_val(D0, AY, VY),         % megkeressuk az erteket
    put_var(D0, (AX, VY),D1),    % beillesztjuk a tudasbazisba
    !.                           % vagas
interpret_line([SX, "=", "@add", SY, SZ],D0,Out,D1,Out) :-
    name(AX,SX),
    name(AY,SY), get_val(D0,AY,VY),
    name(AZ,SZ), get_val(D0,AZ,VZ),
    VX is VY + VZ,
    put_var(D0, (AX,VX), D1),
    !.
interpret_line(["@writeln",SX],D0,Out,D0,[VX|Out]) :-
    name(AX,SX), get_val(D0, AX,VX).
```

Mini-interpreter II

```
% Mintaillesztesek a lehetséges értékek sorozatára
interpret_line([SX, "=", SY], D0, Out, D1, Out) :-
    name(AX, SX),                % string -> atom konverziok
    name(AY, SY),                % masik
    get_val(D0, AY, VY),         % megkeressuk
    put_var(D0, (AX, VY),D1),    % beillesztjuk a változóba
    !.                           % vagas
interpret_line([SX, "=", "@add", SY, SZ],D0,Out,D1,Out) :-
    name(AX,SX),
    name(AY,SY), get_val(D0,AY,VY),
    name(AZ,SZ), get_val(D0,AZ,VZ),
    VX is VY + VZ,
    put_var(D0, (AX,VX), D1),
    !.
interpret_line(["@writeln",SX],D0,Out,D0,[VX|Out]) :-
    name(AX,SX), get_val(D0, AX,VX).
```

- Az **interpret_line**/5 értelmezi – minták szerint – a sztring tartalmát.
- A **name**/2 megvalósítja a sztring <-> atom konverziót.
- A **get_val**/3 kiolvassa a változók listájából a változó értékét (amennyiben szükség).
- A **put_var**/3 menti a változó értékét.

A szótár-kezelő predikátumok:

```
% egy változó értékének a megállapítása
get_val(_,AY,AY):-
    number(AY),
    !.
get_val(Dict,AY,VY):-
    member( (AY,VY), Dict).

% a DICT szótárban tároljuk a változó értékét
put_var(Dict,(AX,VX),[(AX,VX)| Others]):-
    select( (AX,_), Dict, Others),
    !.
put_var(VarDict, (AX,VX), [(AX,VX)| VarDict]).
```

Interpreter - II

„Atomizált” megoldás

```
% "programot" ertelmez
beolvas_utasitas(LUt):-
   see('mitirki.txt'),
   listaba_olvas(LUt),
   seen.

% STREAM-bol listaba olvas soronkent
listaba_olvas([Utasitas| Rest]):-
   egy_sor(S),spaces,
   S \== [],!,
   sorbol_atomlista(S,Utasitas),
   listaba_olvas(Rest).
listaba_olvas([]).

% spaces -- torli az ures karaktereket
% a stream-bol
spaces:-
   peek_char(C),
   (C='\r'; C='\n'; C='\t'),
   get_char(_C2),
   spaces,!.
spaces.

% Egy sort olvas be STRING-kent
egy_sor([C|Rest]):-
   get_char(C),
   C \== end_of_file,
   C \== '\n', C \=='\r',
   egy_sor(Rest).
egy_sor([]).

% sorbol_atomlista(Str,L_Expr)
sorbol_atomlista([],[]):-
   !.
sorbol_atomlista(Str,[Atom|MarE]):-
   kov_atom(Str,Atom,MarS),
   !,
   sorbol_atomlista(MarS,MarE).
```

```
% kov_atom(Str,AccExpr,Ans)
kov_atom(Str,Atom,Mar):-
   append(Coll,[' '|Mar],Str),
   !,
   name(Atom,Coll).
kov_atom(Str,Atom,[]):-
        name(Atom,Str).

% WRAPPER a TAIL-REC uzemmodhoz
feldolgoz(LUt,Ki):-
   feldolgoz(LUt,[],[],KiRev),
   reverse(KiRev,Ki).

%% feldolgoz(Ut,AllTer,AccKi,Ki).
% a megallasi feltetel
feldolgoz([], _ ,Ki,Ki).

% egy utasitas feldolgozasa
feldolgoz([Ut|MarUt],All,EddKi,Ki):-
   ertelmez(Ut,All,UjAll,EddKi,UjKi),
   feldolgoz(MarUt,UjAll,UjKi,Ki).

% Valt vagy SZAM erteke
getErt(Valt,_,Valt):-
   number(Valt),
   !.
% VALT az allapotterbol
getErt(Val,All,Ert):-
   member([Val,Ert],All),
   !.

% Ertelmezo
% WRITELN
ertelmez(['@writeln',V],A,A,Ki,[E|Ki]):-
   member([V,E],A),
   !.
```

```
% Ertekadas:
% Ha a V mar deklaralt
ertelmez([V,'='|J_O],All,[[V,E]|MA],Ki,Ki):-
   select([V| _ ],All,MA), % a regi ...
   ertekel(J_O,All,E),
   !.

% Ha a V nem deklaralt
ertelmez([V,'='|J_O],_All,[[V,E]|MA],Ki,Ki):-
   ertekel(J_O,MA,E).


% ertekel(Kif,Allapotok,Ertek).
ertekel([Valt],All,Ert):-
   getErt(Valt,All,Ert),
   !.
ertekel(['@add',Val1,Val2],All,Ert):-
   getErt(Val1,All,Ert1),
   getErt(Val2,All,Ert2),
   Ert is Ert1+Ert2,
   !.
```

Input-output Prolog-ban

A fentiek mellett lehet használni a következő predikátumokat is:

1. `read_line_to_codes(user_input,Codes)` – a `user_input` stream-ből beolvassa a következő sort és a `Codes` változóban tárolja;
2. `atom_codes(Atom,Codes)` atommá alakítja a `Codes` lista elemeit;
3. `atomic_list_concat(Lista,' ',Atom)` felbont/összefűz egy listát Atommá.

Példa:

```
?- read_line_to_codes(user_input,Cs),
   atom_codes(A,Cs),
   atomic_list_concat(L,' ',A).
|: tesztelem a sztringet.
Cs = [116, 101, 115, 122, 116, 101, 108, 101, 109|...],
A = 'tesztelem a sztringet.',
L = [tesztelem, a, 'sztringet.'].
```

Figyeljük meg a pontot az utolsó atom végén!

Feladatok

- **Legkevesebb számú ugrás:** adott egy lista, melynek nemnegatív elemei az adott pozícióból megengedett ugrás hosszát jelentik (egyet mindig léphetünk). Találjuk meg a legrövidebb sorozatot.
- **Közelítő felezés – particionálás:** adott súlyok listáját osszuk úgy két részre úgy, hogy a keletkezett partíciók súlya közötti abszolút eltérés a minimális legyen.
- **Legnagyobb közös részsorozat:** adott két karaktersor esetén állapítsuk meg a legnagyobb közös részkaraktersort: azt a leghosszabb karaktersort, mely mindkét sorozatban szerepel.

http://www.quora.com/What-are-the-top-10-most-popular-dynamic-programming-problems-among-interviewers

Az Előadások Témái

- Imperatív/deklaratív programok, Prolog alapok
- Listák, listakezelés
- Aritmetikai műveletek és operátorok
- Vágások használata, negáció
- Gyűjtő-predikátumok Prolog-ban, összegzések
- Vég-rekurzió optimalizálás
- Dinamikus predikátumkezelés; input-output prolog-ban
- Kifejezések dekompozíciója
- Rendezések, kereső algoritmusok
- Funkcionális programozás bevezető, történelem
- A Haskell programnyelv elemei, példaprogramok
- Algebrai típusok, típusosztályok
- Magasabb-rendű függvények, Map-Reduce, rendezések
- Input-output
- *(opc) Algebrai struktúrák 2: Monádok stb*
- Ismétlések és kérdések megvitatása

Adminisztra ... ... trívia

link: Canvas learning management system (LMS)

Régi honlap: `http://www.cs.ubbcluj.ro/~csatol/log_funk`

Vizsga (félév elején) – lásd SYLLABUS!

Labor (40%) + Parciális (25%) + Kollokvium (35%)

Laborgyakorlatok:

1. 2 Prolog feladatcsoport **20%**
2. 2 Haskell feladatcsoport **20%**
3. $\aleph_0$ feladat **első n beküldőnek**

Predikátumok (de)kompozíciója

Szükségünk lehet

- új predikátumokra, melyeket tudunk futtatni vagy tárolni;
- elemezni predikátumokat;
- változtatni akarunk predikátumok argumentumain.

Eszköz:

(De)kompozíciós operátor `=..`

`gyereke(istvan, ferenc) =.. L.` konstrukcióra a rendszer válasza:
`L = [gyereke, istvan, ferenc].`

Példa (ism):
Predikátum, mely
egy-argumentumú
függvényeket „kiértékel”.

Az
`ert([sin,cos,inc],LBe,LLKi)`
predikátum
a harmadik argumentumban az
`LBe` bemenetnek megfelelő
`LLKi=[L1,L2,L3]` listák
vannak,
melyek a `sin,cos,inc`
függvényekhez tartoznak.

Predikátumok (de)kompozíciója II

Példa (ism):
Predikátum, mely egy-argumentumú függvényeket „kiértékel”.

Az **ert([sin,cos,inc],LBe,LLKi)** predikátum a harmadik argumentumban az **LBe** bemenetnek megfelelő **LLKi=[L1,L2,L3]** listák vannak, melyek a **sin,cos,inc** függvényekhez tartoznak.

```
ert([] ,_L , []).

ert([F | Mar], L ,[VL | VMar] ):-
   ertEgy(F, L, VL ),
   ert(Mar, L, VMar).


% ertEgy - egy függvényt értékel

ertEgy(_F, [], []).

ertEgy( F, [E | Mar], [VE | VMar] ):-
   Kif =.. [F, E, VE],
   call(Kif),
   ertEgy(F, Mar, VMar).
```

Predikátumok (de)kompozíciója III

Más predikátumok:

- `arg(?A,+Term,?N)` – a második argumentumban szereplő kifejezésből illeszti az `A` változót a kifejezés `N`-edik argumentumával.
- `functor(?Term,?Name,?Arity)` – visszatéríti az első argumentum-beli kifejezés nevét illetve az argumentumainak a számát.

Példa:

```
?- arg(A,haromszog(12,24,13),V).
A = 1,
V = 12 ;
A = 2,
V = 24 ;
A = 3,
V = 13.
```

Implementáljuk az `arg` predikátumot a dekompozíciós operátor segítségével

Rendezések

Feladat

Egy lista elemeit tegyük növekvő sorrendbe.

Írjunk hatékony kódokat a következő rendező algoritmusokra:

- beszúrásos rendezés;
- buborék-rendezés;
- összefésüléses rendezés;
- QuickSort.

Feltételezzük, hogy a „<” és a „>” relációk definiáltak a lista elemeire.

Vizsgáljuk meg az algoritmusok hatékonyságát.

Beszúrásos rendezés

- Rendezett listába szúrjuk be a bemenő lista elemeit.
- A második argumentum a rendezett lista, kezdetben `[]`.
- A `beszur(E,L,UjL)` a rendezett `L`-ba helyezi az `E` elemet.

```
isort( BeL, KiL ) :-
  isort( BeL, [], KiL).

% isort/3 három arg.
isort( [Fej| Mar], Acc, KiL):-
   beszur( Fej, Acc, Acc1),
   isort( Mar, Acc1, KiL).
isort( [], KiL, KiL).

% beszuras
beszur( E, [Fej| Mar], [E, Fej| Mar]):-
   E < Fej,
   !.
beszur( E, [Fej| Mar], [Fej| BMar]):-
   beszur(E,Mar,BMar).
beszur(E,[],[E]).
```

Beszúrásos rendezés

- Rendezett listába szúrjuk be a bemenő lista elemeit.
- A második argumentum a rendezett lista, kezdetben `[]`.
- A `beszur(E,L,UjL)` a rendezett `L`-ba helyezi az `E` elemet.

```
isort( BeL, KiL ) :-
  isort( BeL, [], KiL).

% isort/3 három arg.
isort( [Fej| Mar], Acc, KiL):-
   beszur( Fej, Acc, Acc1),
   isort( Mar, Acc1, KiL).
isort( [], KiL, KiL).

% beszuras
beszur( E, [Fej| Mar], [E, Fej| Mar]):-
   E < Fej,
   !.
beszur( E, [Fej| Mar], [Fej| BMar]):-
   beszur(E,Mar,BMar).
beszur(E,[],[E]).
```

Az algoritmus bonyolultsága

Átlagban a lista feléig hasonlítunk elemeket.
Egy **N** hosszúságú listára: $\sum_{k=1}^{N} k/2 = N(N+1)/4$. A bonyolultság négyzetes.

Buborék-rendezés

- A lista elemeit felcseréli ha nincsenek megfelelő sorrendben.
- Ha csere volt, akkor a lista nem teljesen rendezett, még egyszer kell tehát rendezni.

```
buborek( BeL, RendL) :-
   cserel( BeL, Acc),
   !,
   buborek( Acc, RendL).
buborek( RendL, RendL).

% cserélő predikátum
cserel( [A, B| Mar], [B, A| Mar]):-
    B<A,
    !.
cserel( [A| Mar], [A| CsMar]):-
    cserel( Mar, CsMar).
```

Tulajdonságok:

- A **cserel/2** csak akkor terminál, ha lehet cserélni elemeket;
- Ha a **cserel/2** terminált, akkor a **BeL** lista nem volt rendezett;
- Ha nem lehet cserélni elemeket, akkor a lista rendezett;
- A megoldás ekkor a **RendL** lista.
- Az algoritmus átlagideje **négyzetes**;
- Jó lenne úgy változtatni az algoritmuson, hogy minden rossz sorrendben levő elemet cseréljen.

H.F.

A rendezés:

- Kettéosztjuk a listát;
- Rendezzük rekurzívan;
- Összefésüljük a rendezett részlistákat.

```
frend( [], []).
frend([X], [X]):-
    !.
frend( BeL, RendL) :-
    oszt( BeL, L1, L2),
    frend( L1, RL1),
    frend( L2, RL2),
    osszef( RL1, RL2, RendL).

% felosztás
oszt( [],  [], []).
oszt( [A],[A], []):-
    !.
oszt( [A, B| M], [A| M1], [B| M2]):-
    oszt( M, M1, M2).
```

Tulajdonság:

- Az algoritmus $N \log N$ bonyolultságú;

Összefésüléses rendezés II

- A fésülés során a duplikátumokat kiszűrjük.
- Az utolsó esete az **osszef** predikátumnak.

```
% összefésül két listát
osszef( [], KiL, KiL).
osszef( KiL, [], KiL):-
    !.
osszef( [F1| M1], [F2| M2], [F1| M]):-
    F1 < F2,
    osszef( M1, [F2| M2], M),
    !.
osszef( [F1| M1], [F2| M2], [F2| M]):-
    F1 > F2,
    osszef( [F1| M1], M2, M),
    !.
osszef( [F| M1], [F| M2], [F| M]):-
    osszef( M1, M2, M).
```

QuickSort

- Kettéosztjuk a listát az első – **Pv** – elemnél kisebb és nagyobb értékekre.
- Rendezzük rekurzívan a két részlistát;
- Összefűzzük a rendezett részlistákat.

Tulajdonság:

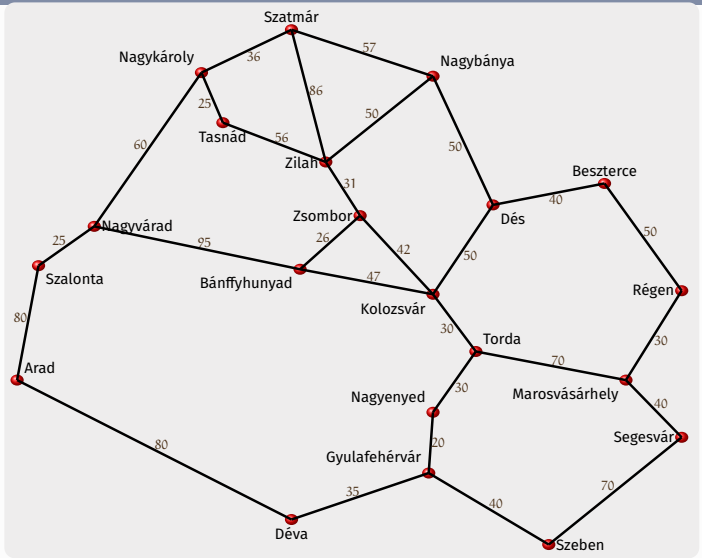
- Az algoritmus átlagos bonyolultsága
  $N \log N$;

```
1  qsort( [],  []).
   qsort( [X], [X]):-
       !.
   qsort( [Piv |Mar], RendL) :-
       q_oszt( Piv, Mar, KL, NL),
6      qsort( KL, RKL),
       qsort( NL, RNL),
       append(RKL, [Piv| RNL], RendL).

   q_oszt( Pv, [E| Mar], [E| M1], M2):-
11     E < Pv,
       !,
       q_oszt(Pv, Mar, M1, M2).

   q_oszt(Pv, [E| Mar], M1, [E| M2]):-
16     E > Pv,
       !,
       q_oszt(Pv, Mar, M1, M2).

   q_oszt(Pv, [ _| Mar], M1, M2):-
21     q_oszt(Pv, Mar, M1, M2).

   q_oszt(_,[],[],[]).
```

Rendezés összefoglaló

- A beszúrásos-, illetve buborék-rendezés nem hatékony;
- Ajánlott az algoritmusok bonyolultságának a vizsgálata;
- A Prolog **`sort/2`** algoritmusa az összefésüléses rendezést használja;
- Az **`msort/2`** a duplikátumokat nem szűri;
- Használhatjuk a **`keysort/2`** predikátumot, mely **`Kulcs-Érték`** elemű listát rendez;
- A **`predsort(Pred,BeL,RendL)`** a háromargumentumú **`Pred`** predikátumot hívja meg, melynek első argumentuma a `<`, `>`, vagy `=` operátorok valamelyike.

Használjuk a **`help.`** parancsot



- Kereső algoritmusok:
  - egy állapottérben működnek;
  - állapotok között utakat specifikálnak;
  - keresnek optimális utat.
- Több típusú:
  - Mélységi keresés;
  - Szélességi keresés;
  - Bonyolultabb kereső algoritmusok.
- Nagyon fontos az állapottér-reprezentáció.
- A klasszikus mesterséges intelligenciában alkalmazzák.

Kereső algoritmusok típusai

- **Mélységi keresés:**
  - a Prolog alapértelmezett kereső algoritmusa.
- **Szélességi keresés:**
  - Prolog megvalósítása nehézkesebb;
- **A* algoritmus:**
  - Figyelembe veszi a becsült távolságot a céltól.

Megszorítások

- Nem tudjuk kiterjeszteni a keresési gráfot;
- Szabályunk van, melyek a lehetséges következő lépéseket adják meg;
- Minél kevesebb erőforrás felhasználásával szeretnénk célt érni.

Mélységi keresés

Tegyük fel, hogy:

- Találtunk egy ábrázolást a feladatunknak;
- Megírtuk az `egylepes(Most,Kov)` predikátumot:
  - felsorolja a `Most` állapotból lehetséges következő állapotokat;
  - a vagy-okat tehetjük külön predikátumba;
  - a `findall`-t tudnánk használni az összes következő állapot tárolására;
- Tudjuk tesztelni a célállapotot: `cel(All)` igaz, ha célban vagyunk;
- Célunk, hogy a kezdeti állapottól egy úton – állapot-listán – jussunk el a célig.

Mélységi keresés II

```
keres_celig(All, []):-
   cel(All).
keres_celig(All, [Kov| KLista]):-
   egylepes(All, Kov),
   keres_celig(Kov, KLista).
```

- Az algoritmus nem ismeri fel a köröket a gráfban $\Rightarrow$ nem terminál.
- Szükséges egy mélységi korlát bevezetése a **`keres_celig(Kezd,MaxM,Lista)`** formában.

Mélységi keresés II

```
keres_celig(All, []):-
    cel(All).
keres_celig(All, [Kov| KLista]):-
    egylepes(All, Kov),
    keres_celig(Kov, KLista).
```

- Az algoritmus nem ismeri fel a köröket a gráfban $\Rightarrow$ nem terminál.
- Szükséges egy mélységi korlát bevezetése a **keres_celig(Kezd,MaxM,Lista)** formában.

```
keres_celig(All, _MaxM, []):-
    cel(All).

keres_celig(All, MaxM, [Kov| KLista]):-
    egylepes(All, Kov),
    MaxM > 0,
    TM is MaxM - 1,
    keres_celig(Kov, TM, KLista).
```

Javított mélységi keresés

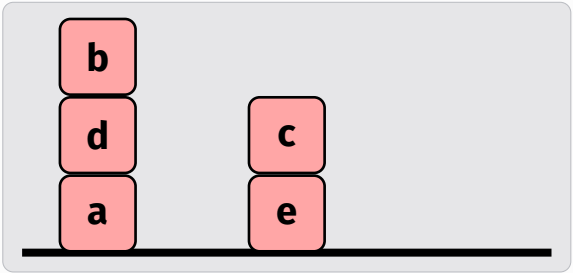
- Tároljuk a látogatott állapotokat;
- A már látott állapotokba nem lépünk;
- **keres_2(All,MaxM,Lista)** segédpredikátuma tárolja a látogatott állapotokat.

```
keres_2(All, MaxM, LAll):-
    keres_2(All, MaxM, [], LAll).

% keres_2 - négyargumentumos változat
keres_2(All, MaxM, Acc, RevL):-
    cel(All),
    reverse([All| Acc], RevL).

keres_2(All, MaxM, Acc, VL):-
    egylepes(All, Kov),
    \+ member(Kov, Acc),
    MaxM > 0,
    TM is MaxM - 1,
    keres_2(Kov, TM, [All| Acc], VL).
```

A két **keres_2** argumentumszáma különböző.

Alkalmazás (1)

Kockavilág I

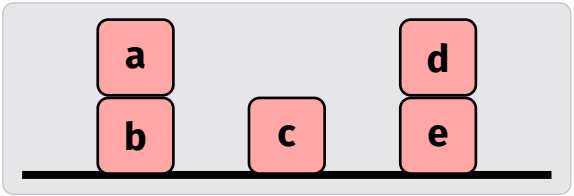


- Kezdőállapotból egy célállapotba eljutni.
- Csak a legfelső kockát mozgathatjuk – üres oszlopba vagy felülre.

Reprezentáció:

- Listák listája: az oszlopok fentről lefele;
- Annyi lista van, ahány oszlop;
- A fenti lista ábrázolása: `[[b,d,a],[c,e],[],[],[]]`;

Célállapot: `[[a,b],[c],[d,e],[],[]]`



Egy lépés:

```
egylepes(BeAll, [Mar1, [Fent1|O2] | Regiek]):-
    select([Fent1|Mar1],BeAll,MaradAll),
    select(O2,MaradAll,Regiek).
```

- A célállapotban a listák sorrendje nem számít: permutálhatóak az oszlopok.

Célállapot tesztje:

```
cel(All):-
    celteszt(All,[[a,b],[c],[d,e],[],[]]).

celteszt( [Fej| Mar], CelL):-
    select(Fej, CelL, LMar), !,
    celteszt(Mar,LMar).
celteszt([],[]).
```

Ha legtöbb hét lépésre hívjuk meg:

`BeL=[[b,d,a],[c,e],[],[],[]],keres_2(BeL,7,LL),`

- Összesen 88605 megoldás lesz; példák:
- (1): `[ [[b,d,a],[c,e],[],[],[]], [[d,a],[b,c,e],[],[],[]], [[c,e],[b],[d,a],[],[]], [[e],[c],[b],[d,a],[]], [[a],[d,e],[c],[b],[]], [[],[a,b],[d,e],[c],[]]]`,
- (2): `[ [[b,d,a],[c,e],[],[],[]], [[e],[c],[b,d,a],[],[]], [[d,a],[b],[e],[c],[]], [[a],[d,e],[b],[c],[]], [[],[a,b],[d,e],[c],[]]]`

Alkalmazás (1)

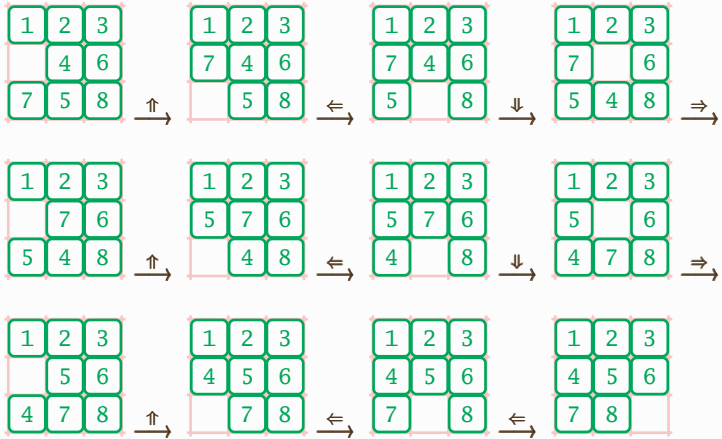
- Nem mindegy, hogy milyen algoritmust használunk;
- Fontos, hogy ki tudjuk választani a legrövidebb megoldást;
- A megoldások száma nagyon gyorsan nő:
  - $1, 2, 3 \Rightarrow 0$
  - $4 \Rightarrow 12$
  - $5 \Rightarrow 306$
  - $6 \Rightarrow 5,598$
  - $7 \Rightarrow 88,605$
  - $8 \Rightarrow 1,296,798$
  - $\ldots$

Programkód:

```
findall(K,(keres_2([[b,d,a],[c,e],[],[],[]],8,M),length(M,K)),T),length(T,N)
```

Alkalmazás (2)

Kirakós-puzzle I



- Ábrázolás listaként, a céllista az `[1,..,8,0]`;
- A lépések függnek a nullás pozíciójától;

Alkalmazás (2)

Kirakós-puzzle II

Megoldás:

```
puzzle( K0, Ut) :-
    length( K0, Nk),
    N is ceiling( sqrtNk),
    N * N =:= Nk,
    gen_cel( Nk, Cel),
    ut( N, K0, Cel, Ut).

% wrapper predikátum
ut( N, K0, Cel, Utak) :-
  ut(N, K0, [], [K0], Cel, Utak).
```

Célállapot:

```
gen_cel( N, Cel):-
    N1 is N - 1,
    gen_cel( N1, [0], Cel).
gen_cel( 0, Cel, Cel):-
    !.
gen_cel( K, L, Cel) :-
    K1 is K - 1,
    gen_cel( K1, [K| L],Cel).
```

```
% a Kezd es Cel  kozott keres utat.
% A TUt-ba gyujtjuk a lepeseket.
% Megallunk, ha Kezd==Cel.
ut(_,Cel,TUt,_,Cel,Ut):-
    reverse(TUt,Ut), !.
```

```
% Lepunk Kezd-bol es keressuk az
% utat a Cel-ba. Az AllList-ben
% a mar bejart allapotok vannak.
ut(N,Kezd,TUt,Volt,Cel,VU) :-
  egylepes(N,Kezd,IE,Koz),
  \+ member(Koz,Volt),
  length(Volt,Nb),Nb < N*N,
  ut(N,Koz,[IE|TUt],[Kezd|Volt],Cel,VU).
```

Alkalmazás (2)

Kirakós-puzzle II

Megoldás:

```
puzzle( K0, Ut) :-
    length( K0, Nk),
    N is ceiling( sqrtNk),
    N * N =:= Nk,
    gen_cel( Nk, Cel),
    ut( N, K0, Cel, Ut).

% wrapper predikátum
ut( N, K0, Cel, Utak) :-
  ut(N, K0, [], [K0], Cel, Utak).
```

```
% a Kezd es Cel  kozott keres utat.
% A TUt-ba gyujtjuk a lepeseket.
% Megallunk, ha Kezd==Cel.
ut(_,Cel,TUt,_,Cel,Ut):-
   reverse(TUt,Ut), !.
```

Célállapot:

```
gen_cel( N, Cel):-
    N1 is N - 1,
    gen_cel( N1, [0], Cel).
gen_cel( 0, Cel, Cel):-
    !.
gen_cel( K, L, Cel) :-
    K1 is K - 1,
    gen_cel( K1, [K| L],Cel).
```

```
% Lepunk Kezd-bol es keressuk az
% utat a Cel-ba. Az AllList-ben
% a mar bejart allapotok vannak.
ut(N,Kezd,TUt,Volt,Cel,VU) :-
  egylepes(N,Kezd,IE,Koz),
  \+ member(Koz,Volt),
  length(Volt,Nb),Nb < N*N,
  ut(N,Koz,[IE|TUt],[Kezd|Volt],Cel,VU).
```

- Fontosak a:
  - robotikában – célállapot elérésénél;
  - navigációs eszközökben;
  - mesterséges intelligenciában;
- Az algoritmusok nem hatékonyak;
  - szükséges egy-egy feladattípushoz adaptálni,
  - heurisztikák segíthetnek,

Feladatok:

- Osszunk fel egy téglalapot négyzetekre. A téglalapnak adottak a méretei. Találjuk meg az összes megoldást.
- Optimális egy megoldás, ha létezik egy bontási sorrend úgy, hogy kettőbe tudjuk vágni a téglalapokat egy egyenes vonallal. Találjunk optimális megoldást adott négyzetre bontásra.
- Adott egy lista, mely négyzetek oldalainak méretét tartalmazza. Helyezzük el a négyzeteket egy téglalap alakú mezőben. Keressük meg a minimális területű téglalapot.

- A hangsúly az relációk meghatározásán, a kapcsolatok felépítésén (a szabályok meghatározásán) van. A megjelenítés mellékes - ezért a konzol mód.
- Az SQL nyelvvel rokon, tehát potenciálisan hasznos a programozók számára.
- A keresési algoritmus (backtracking) beépített, nem kell ezt külön kódolni $\Rightarrow$ kevesebb hibalehetőség.
- Kevesebb kód, könnyebb hibakeresés.

Hátrány:
Zárt világ hipotézis: ha nincs válasz - vagy nemleges - az pusztán annyi, hogy nem bizonyítható a tudás/adatok alapján.

Az Előadások Témái

- Imperatív/deklaratív programok, Prolog alapok
- Listák, listakezelés
- Aritmetikai műveletek és operátorok
- Vágások használata, negáció
- Gyűjtő-predikátumok Prolog-ban, összegzések
- Vég-rekurzió optimalizálás
- Dinamikus predikátumkezelés; input-output prolog-ban
- Kifejezések dekompozíciója
- Rendezések, kereső algoritmusok
- Funkcionális programozás bevezető, történelem
- A Haskell programnyelv elemei, példaprogramok
- Algebrai típusok, típusosztályok
- Magasabb-rendű függvények, Map-Reduce, rendezések
- Input-output
- *(opc) Algebrai struktúrák 2: Monádok stb*
- Ismétlések és kérdések megvitatása

link: Canvas learning management system (LMS)

Régi honlap: `http://www.cs.ubbcluj.ro/~csatol/log_funk`

Vizsga (félév elején) – lásd SYLLABUS!

Labor (40%) + Parciális (25%) + Kollokvium (35%)

Laborgyakorlatok:

1. 2 Prolog feladatcsoport **20%**
2. 2 Haskell feladatcsoport **20%**
3. $\aleph_0$ feladat **első n beküldőnek**

„Miért fontos a funkcionális programozás”

John Hughes — Why functional programming matters

A cikk bevezetője (id)

- As software becomes more and more complex, it is more and more important to structure it well.
- Well-structured software is
  - easy to write and to debug, and
  - provides a collection of reusable modules that reduce future programming costs.
- two features of functional languages that can contribute to modularity:
  - higher-order functions and
  - lazy evaluation.

(id.vége)

A Haskell nyelvet használjuk

mert tartalmazza a magasabb-rendű függvények implementációját, a lusta kiértékelést, a típuslevezetést, a curry-zést.

A modern funkcionális nyelvek jellemzői

Funkcionális jellemzők, melyeket átvettek a modern programnyelvek:

- lambda-kifejezések használata: szeretnénk egy – egyszerű – műveletet elvégezni, a függvény neve nem annyira fontos.
- paraméterek típusainak a mellőzése: szeretnénk, ha a változók és függvényparaméterek típusait levezetné a rendszer.
- függvények parciális paraméterezése: definiáljunk függvényeket a korábban megírt függvényekből úgy, hogy néhány paraméter értékét rögzítjük (pl. C++ konstruktorok).

Programozási szokások, melyek alapvetőek a funkcionális (deklaratív) programozásnál:

- „immutable” – az objektumokat/változókat a létrehozás után ne változtassuk[4]; a deklaratív programnyelvek legfőbb jellemzője.
- „side-effect-free” a függvényeken belül ne változtassuk a globális változók értékeit.[5]
- „generic functions” adott függvényt minél általánosabban írjunk meg: a **Haskell** típuslevezetése „biztosítja” a függvények generikus jellegét.[6]

[4] Mi történik **Java** vagy **C++** objektumok „setter” metódusaival
[5] Legyenek globális változók, ha csak függvényeink vannak?
[6] Ehhez egy nagyon jól kigondolt függvényosztály szükséges!

Fősodrású – mainstream – programnyelvek

sok funkcionális jellemzőt implementáltak:

- A **java**, **C++**, **python** nyelvek mindegyike lehetővé teszi a lambda-függvények definícióját és használatát;
- A **C++** nyelvben az **auto** kulcsszó lehetővé teszi a típuslevezetést;
- Nagyon sok nyelvben – pl. **c++**, **julia** – léteznek definíciók, melyek immutábilis „változók” kijelentését teszik lehetővé;

Funkcionális programnyelvek

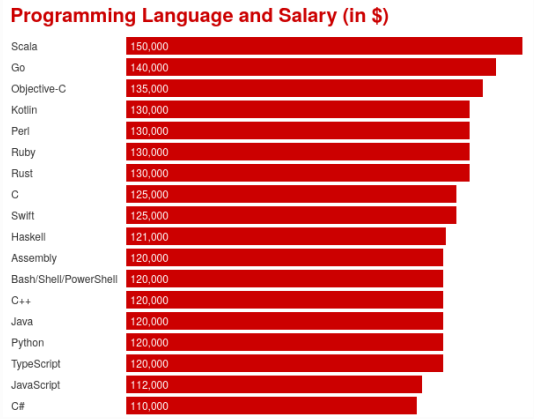
melyeket az iparban használnak:

- **Closure** – a LISP nyelv dialektusa („code as data”), mely **JVM**-re fordít, de a **ClosureScript** JS-re, illetve **.NET** rendszerre fordít;
- **Elm** – funkcionális nyelv, mely HTML/JS-re fordít. WEB-es frontend-ek implementálására használják;
- **Erlang** – az **Ericsson** által fejlesztett – minta-alapú – kevert funkcionális jellemzőkkel is rendelkező nyelv, melyet banki rendszerekben is használnak.
- **Elixir** – az Erlang VM-re fordító nyelv, mely WEB-re és beágyazott rendszerekre fordít.

Cégek melyek támogatják a funkcionális programozást: forrás: medium.com

- WhatsApp – nyelve az **Erlang**;
- FaceBook – **Haskell**-t használ a belső eszközök fejlesztésére;
- PInterest – az **Erlang**-ban megírt értesítő rendszere a kódbázisát 10.000-ről 1000 sorra csökkentette, **és felezte az erőforrás-igényt**.
- Bet365 – a javaról váltott **Erlang**-ra, mely által a párhuzamosítás sokkal könnyebbé vált.
- Google – a belső projektjeinek az implementálása **Haskell**ben történik – pl. *Ganeti*, mely a *Xen* és a *KVM* alapú virtualizációt manageli.
- Intel – a programrendszerek kutatócsoportja többszálú programozást támogató nyelvet implementál. ▸ forrás: Intel Labs
- NICTA – a **Haskell**-t használják egy *„mikroKernel”* helyességének az ellenőrzésére (2010-2016). pl. ACM Transactions on Computer Systems, 34/1, DOI: 10.1145/2893177

Az iparban a fizetések mértéke:



forrás: insigths

A funkcionális programmodell I

- Egy programozási stílus, egyszerűsíti a programkódot:
  - Levezeti a függvények típusát;
  - A while/for ciklusok helyett rekurzió;
- Nem tesz különbséget a függvények és az adatok között: minden függvény;
- Következmény: csak függvényeket adunk át paraméterként;

Alkalmazások:

- A modern programnyelvek funkcionálisak is: **C++**, **Java8**, **JavaScript**, **C#**, **Scala**, **Ruby**, ...
- A Map-Reduce módszer a nagy adatbázis – Big Data – feldolgozására.
- A **Haskell** egy tisztán funkcionális nyelv, ebben fogjuk írni a programokat.

Jellemzők:

- Lambda-függvények és lambda-kifejezések használata:
  - `(\ x -> x*x)`

  ha nem fontos a függvény neve.
- Függvények argumentumként történő átadása:
  - `map (\ -> x*x) [2,3,4,5]`

  kimenet: `[4,9,16,25]` – fontos a függvényargumentum.
- Függvények típusának levezetése:
  - `(sq x = x*x)`

  típusa: `sq :: Num a => a -> a`

a típusváltozó

A Haskell-nek nagyon rugalmas típusrendszere van, amelybe be kell illesszük saját típusunkat.

Funkcionális programozás történelem I

- Alapja a XX. század 30-as éveiben kidolgozott lambda-kalkulus, mely eredetileg a függvények kiszámíthatóságát tanulmányozta.
- A Church-Turing tézis szerint egy természetes számokon értelmezett függvény kiszámítható – a szó klasszikus értelmében –, amennyiben egy Turing-gép ki tudja számítani (1930–1952).
- A kiszámíthatóság fontos lépése a Curry-zés: tetszőlegesen nagy argumentumszámú függvény felírható egy-argumentumú függvényként:

$$f(x, y, z) = f_x(y, z) = (f_x)_y(z)$$

A Curry-zés következménye, hogy függvényeket tudunk származtatni:

```
inc = (+) 1
```

Funkcionális programozás történelem II

- Az első funkcionális programnyelv a Lisp (LISt Processing language), melyet John McCarthy (1927–2011) alkotott. A rekurzív függvények tanulmányozása volt a cél.
- A LISP nyelvnek (és utódainak – Scheme[7]) fontos szerepük volt a mesterséges intelligencia fejlődésében. Nem volt tisztán funkcionális nyelv (kezdetben volt `goto` utasítás is)...
- 1970-es évek végétől[8] hardver-kombináló programok írása SASL-ben (St. Andrews Static Language).
- A nyolcvanas évek közepétől fontos a a típus-absztrakció: egy listákat feldolgozó függvény *ne* legyen függő az elemeinek a típusától. Megjelentek a típusos funkcionális nyelvek, melyek szükségessé tették típushierarchiák fejlesztését.

Funkcionális programozás történelem III

- A 80-as évek közepétől megjelent a többszörös kijelentése a függvényeknek (és természetesen a mintaillesztés).
- A Miranda[9] funkcionális nyelv, mely megvalósította a lusta kiértékelést. Megjelennek az algebrai típusok, melyek típusváltozókat is tartalmaznak.
- Az **ML**[10] és a **Clean**[11] nyelveket a nyolcvanas évek végén fejlesztették.
- A **Clean** jellegzetessége az egyedi típusok – uniqueness typing – definíciója, mely a felhasználói bemenetet tudja kezelni.

Funkcionális programozás történelem IV

- A **Haskell** bizottság[12] jóváhagyta a Haskell-standardot, mely azóta is funkcionális.
- A Haskell-ben a típusosztály fontos szereplő, segítségével lehet generikus kódokat írni.
- A deklaratív paradigma szerint egy változó értéke nem változik – immutabilitás – ezt a Haskell a
  - *monádokkal* valósítja meg, melyek az
  - *akciókat* – a dinamikus komponenseket – kódolják.

[7] Nagyon sok változat létezik: MicroScheme, Scheme'48, YPsilon, Schemik, …
[8] D.A. Turner: Some History of Functional Programming Languages
[9] Sok változat, közismert a Miranda.org változat.
[10] A svédországi Chalmers egyetem fejlesztése 1984-ből
[11] A hollandiai Nijmegen fejlesztése, Rinus Plasmeijer és csapata (Horváth Zoltán doktori dolgozata) fejlesztették.
[12] Haskell Report – publikálva a *SIGPLAN Notices*-ban (Hudak et al. 1992).

Az Előadások Témái

- Imperatív/deklaratív programok, Prolog alapok
- Listák, listakezelés
- Aritmetikai műveletek és operátorok
- Vágások használata, negáció
- Gyűjtő-predikátumok Prolog-ban, összegzések
- Vég-rekurzió optimalizálás
- Dinamikus predikátumkezelés; input-output prolog-ban
- Kifejezések dekompozíciója
- Rendezések, kereső algoritmusok
- Funkcionális programozás bevezető, történelem
- A Haskell programnyelv elemei, példaprogramok
- Algebrai típusok, típusosztályok
- Magasabb-rendű függvények, Map-Reduce, rendezések
- Input-output
- *(opc) Algebrai struktúrák 2: Monádok stb*
- Ismétlések és kérdések megvitatása

Adminisztra ... ... trívia

link: Canvas learning management system (LMS)

Régi honlap: `http://www.cs.ubbcluj.ro/~csatol/log_funk`

Vizsga (félév elején) – lásd SYLLABUS!

Labor (40%) + Parciális (25%) + Kollokvium (35%)

Laborgyakorlatok:

1. 2 Prolog feladatcsoport **20%**
2. 2 Haskell feladatcsoport **20%**
3. $\aleph_0$ feladat **első n beküldőnek**

Haskell jellemzők

- A Haskell (`haskell.org`) egy nyitott forráskódú platformfüggetlen rendszer.
- Két módban futtatható: kompilált – **`ghc`** – vagy interpretált – **`ghci`** – módokban.
- Az interpretált módot használjuk a tesztelés során (hatékony futtatást kompilálás után lehet elérni).
- A **`ghci`** egy *linux*-alapú rendszer:
  - **`:l <filenév>`** – egy file beolvasása (load);
  - **`:r`** – reload;
  - **`:h`** – help;
  - **`:q`** – kilép.
- A sikeres kompilálás után tudjuk futtatni az implementált függvényeket.
- A szerkesztéshez használjunk okos szerkesztőt, mely képes felismerni a **`Haskell`** szintaxist (notepad++, sublime-text).

Haskell jellemzők

I

- A funkcionális programmodell megvalósítója, de facto standard a 2000-es évek elejétől;
- Modern programozási paradigma: típusok, osztályok, függvények kijelentése;
- A program futása ekvivalens a függvénydefiníciók behelyettesítésével;
- A számításnak – és számíthatóságnak – egy matematikai modellje.

Haskell jellemzők II

A funkcionális programozás:

- deklaratív: a függvényeknek nincsenek mellékhatásai,
- A programozás specifikáció-közeli: általában a programok – modulok – lépésenkénti fejlesztése (felbontása),
- Fontos a hatékonyság és a kiszámíthatóság,
- Kiértékelési stratégiák: az egyszerűsíthető kifejezések egyszerűsítése.
  - **lusta kiértékelés**;
  - mohó kiértékelés (hagyományos);
  - párhuzamos kiértékelés.
- Bizonyított, hogy a lusta kiértékeléssel egy program (eredménye) mindig kiszámítható, amennyiben az létezik.

Haskell jellemzők III

Jellemzők:

- Lambda-függvények és *lambda-kifejezések* használata:
  - `(\ x -> x*x)`

  ha nem fontos a függvény neve.
- Függvények *argumentumként* történő átadása:
  - `map (\ -> x*x) [2,3,4,5]`

  kimenet: `[4,9,16,25]` – fontos a *függvényargumentum*.
- Függvények *típusának levezetése*:
  - `sq x = x*x`

  típusa: `sq :: Num a => a -> a`

Fontos: az `a` egy típusváltozó

A Haskell-nek nagyon rugalmas *típusrendszere* van, amelybe be kell illesszük saját típusunkat.

- Egyszerű – kijelentő – függvénydefiníciók:
  - `sq x = x * x`
  - `inc x = x + 1`
  - `sqinc x = sq (inc x)`
- Lehetőség matematikai függvénydefinícióra:
  - `sqinc = sq . inc`
    (a `.` a függvényösszetétel operátor, az aritmetikai operátorokhoz hasonlóan)
  - `fact n = prod [1..n]`
    (az `[1..n]` listakonstruktor, számokat tartalmazó listát állít elő)

A `ghci` interpreter

Az interaktív módban lehetőség van függvények definíciójára a `let twice f x = f (f x)` paranccsal.

Feltételes utasítások:

Az `if`-et kerüljük!

- Mintaillesztéssel:

```
my_app [] lis = lis
my_app (x:xs) lis = x : my_app xs lis
```

- Feltételekkel:

```
my_app lis1 lis2
    | lis1 == []  = lis2
    | otherwise  = x: my_app xs lis2
    where (x:xs) = lis1
```

Lokális (függvény)kijelentések:

- **where** – lokális függvénykijelentések.

```
oszt []  = ([], [])
oszt [x] = ([x],[])
oszt (x1:x2:xs) = (x1:xs1,x2:xs2)
    where (xs1,xs2) = oszt xs
```

Figyeljük meg:

- Az **xs1**, **xs2** változók csak az **oszt** függvényen belül léteznek.
- A változókat mintaillesztéssel kapjuk meg – az egyenlőség bal oldalán.

Listakezelés Haskell-ben

A listák alapstruktúrák:

- A listakonstruktor a **`(:)`** `1:2:3:4:5:[]`
- A listák végét a **`[]`** jelzi.
- Sokfajta listakonstruktor:
  - Felsorolás `[1,2,3,4,5]`
  - Felbontás: `(1:2:[3,4,5])`
  - Generálás: `[1..100]`
  - Generálás II: `[1,9..100]`
- Descartes szorzat (halmazelméleti jelölés):
  `[f x y | x<-lista1, y<-lista2, feltetel]`

Lista permutációja:

```
perm []     = [ [] ]
perm (x:xs)
  =[bsz x i tp | i<-[0..length xs], tp<-perm xs]
  where
    bsz e 0 xs = e:xs
    bsz e i (x:xs)
      | i>0 = x : bsz e (i-1) xs
```

Nyelvi elemek:

- Mintaillesztések: (1) üres lista tesztje; (5) argumentum értékének tesztje,
- Feltételes utasítás (7) – `i>0`,
- Lokális függvénykijelentés (5-7),
- Listakonstruktor (1) felsorolás; (3) `..`; (3) Zermelo-Fraenkel formalizmus.

Parciális függvénydefiníció

Függvények parciális definíciója:
A bemenetének egy részhalmazán érvényes.

```
hd (x:xs) = x

tl (x:xs) = xs

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

Parciális függvénykijelentések

A hibakezelés egy – költségmentes – eszköze: ha a függvény nincs definiálva bizonyos bemeneti értékekre (üres lista, negatív számok), akkor a rendszer leáll hibaüzenettel.

Az Előadások Témái

- Imperatív/deklaratív programok, Prolog alapok
- Listák, listakezelés
- Aritmetikai műveletek és operátorok
- Vágások használata, negáció
- Gyűjtő-predikátumok Prolog-ban, összegzések
- Vég-rekurzió optimalizálás
- Dinamikus predikátumkezelés; input-output prolog-ban
- Kifejezések dekompozíciója
- Rendezések, kereső algoritmusok
- Funkcionális programozás bevezető, történelem
- A Haskell programnyelv elemei, példaprogramok
- Algebrai típusok, típusosztályok
- Magasabb-rendű függvények, Map-Reduce, rendezések
- Input-output
- *(opc) Algebrai struktúrák 2: Monádok stb*
- Ismétlések és kérdések megvitatása

Adminisztra ... ... trívia

link: Canvas learning management system (LMS)

Régi honlap: `http://www.cs.ubbcluj.ro/~csatol/log_funk`

Vizsga (félév elején) – lásd SYLLABUS!

Labor (40%) + Parciális (25%) + Kollokvium (35%)

Laborgyakorlatok:

1. 2 Prolog feladatcsoport **20%**
2. 2 Haskell feladatcsoport **20%**
3. $\aleph_0$ feladat **első n beküldőnek**

Haskell alaptípusok

- **Alap- – beépített – típusok:**
  - Alaptípusok: `Int`, `Integer`, `Float`, `Double`, `Char`, `Bool`, ...
- **A függvényeknek is van típusa:**
  - Ennek a jelölése `Integer->Integer`
- **Új típusokat tudunk származtatni a már létező típusokból:**
  - Párosok (hármasok, ...) `( a,b )`
    Az `a`, `b` típusváltozót jelöl: `(Int,Char)`, `((Int,Bool),Int,Float)`.
- **A lista is alap-típus**, de a lista-elem típusának a függvénye:
  `[ Int ]`, `[ (Bool,String) ]`.

Haskell típusváltozók

Típusváltozó

- A polimorfizmus egyik hatékony eszköze (a **hd** minden listára lefut);
- Mindig kisbetűvel kezdődik;
- Nagyon hasznos a függvények/paraméterek típusának a levezetésénél;

- Alapstruktúra a lista, a típusa tartalomfüggő: `[a]`
  az **a** lehet alap- vagy származtatott típus

`[Int]`, `String=[Char]`, `[ [Float] ]`, ...

Haskell típusváltozók II

A típuslevezetéseknél használjuk a változókat.

Példák:

```
dList x = x ++ x

dList:: [a] -> [a]
```

```
double x = x + x

double:: Num a => a -> a
```

A fenti típuslevezetés illusztrálja a típusváltozók használatát:

- (bal) Ismert a `(+++)` operátor – függvény – szignatúrája, „kiszámítjuk” az általunk definiált függvény típusát.
- (jobb) Ismert a `(+)` operátorhoz szükséges előfeltételek; ezeket alkalmazzuk. (`Num` egy típusosztály)

Haskell típusdefiníciók

(Allen&Moronuki, 2015, pp. 454)

```
data Vehicle = Car Manufacturer Price
    -- (t)      (k1)     [1]       [2]
             | Plane Airline
    --         (k2)     [3]
5           deriving (Eq, Show)
```

A **Manufacturer**, a **Price**, az **Airline** korábban definiált típusok.

Ahol:

- **data** - típuskonstruktor
- (t) - típusnév;
- (k1),(k2) - alkalmazható konstruktorok;
- (i) - az új típus részei.

A **data** konstruktor esetében

- A típusnév függvények szignatúrájában fog szerepelni.
- A konstruktorok függvények:

```
Car:: Manufacturer -> Price -> Vehicle
Plane:: Airline -> Vehicle
```

- A **deriving Eq** – generálja az **Eq** típusosztály függvényeit.

Haskell „record”

(Allen&Moronuki, 2015, pp. 472)

```
data Automobile = Null
    |  Car { make :: String
           , model :: String
           , year :: Int
           } deriving (Eq, Show)
```

Ahol:

„|” - vagy jel - az adatkonstruktorok alternatíváit jelöli.

make,... - az attribútumok nevei.

A **make**, a **model**, a **year** adott típusú mezők, melyekre tudunk hivatkozni.

A nevesített mezők esetén:

A mezők függvények:

```
make:: Automobile -> String
model:: Automobile -> String
year:: Automobile -> Int
```

A rendszer generálja a `getter` metódusokat.

Típusdefiníciók

A `data` típuskonstruktor

- Lehet paraméterezett, pl.

  ```
  data Tree a
  ```

  egy „a” típusú elemeket tároló fa.

- Mindig kell, hogy tartalmazzon konstruktort, pl. a `Tree a` esetén a `Node` vagy a `Leaf` konstruktorokat.

A `type` szinoníma-definíció

Egy létező típuskonstrukciót helyettesít, pl.

```
type String=[Char]
```

Segíti a típusok generikus használatát.

Példa:

- A típuslevezetés eredménye (rendszer-következtetés):

```
double:: Num a => a -> a
double x = x + x
```

  - A `Num` egy típusosztály;
  - A '+' művelet ebben a típusosztályban van értelmezve;
  - A művelet jelenléte implikálja a típusmegszorítást.

`double:: Num a =>` ...

Típusosztályok II

- Lehetővé teszik a hatékony típusozást és típuslevezetést.
  A **double** függvény értelmezett, ha a + van értelmezve.

Típusosztály

Adott funkcionalitással rendelkező entitások:
függvényeket lehet futtatni hibamentesen.

- A **Num** típusosztály tartalmazza a numerikus műveletek sémáit:

```
class Num a where
  (+), (-), (*) :: a -> a -> a
  negate        :: a -> a
  abs           :: a -> a
  signum        :: a -> a
  fromInteger   :: Integer -> a
  x - y       = x + negate y   -- inline kód
  negate x    = 0 - x
```

  - A kivonás és a negálás műveletek implementáltak, a többit kell implementáljuk ahhoz, hogy a **Num** típusosztályba – is – tartozzon a típusunk.
  - Ha nincs szükségünk pl. a szorzásra, akkor azt ki tudjuk hagyni (parciális definíció).

- A típusainkra tudjuk futtatni a **double** kódját, ha
  - implementáltuk az összeadást és
  - instanciáltuk a **Num** típusosztályra:

Például:

```
instance Num Bool where
  True  + _ = True
  _ + b = b
  False * _ = False
  _ * b = b
```

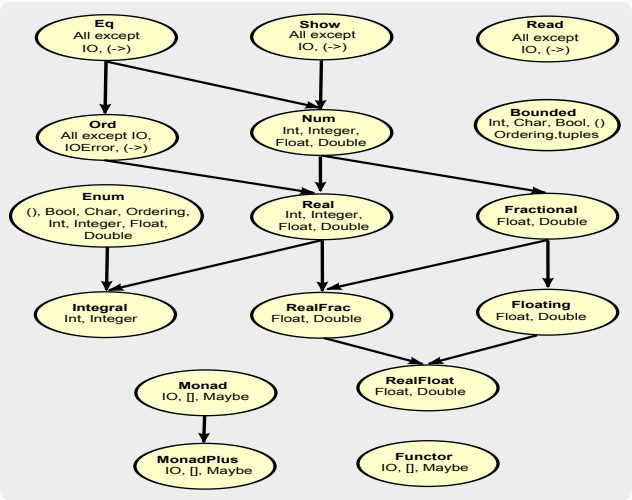
- az **instance** kulcsszó a bennfoglalást rögzíti;
- Eredmény, hogy tudjuk futtatni a **double**-t a **Bool** típusú változókra.

Típusosztályok V

- Főbb típusosztályok:
  - **Eq** típusosztály:
    ```
    class Eq a where
      (==), (/=) :: a -> a -> Bool
    ```
  - **Num** típusosztály, az aritmetikai műveletek kódja.
  - **Fractional** osztást biztosít.
  - **Show** típusosztály a konzol-ra történő kiíratást valósítja meg.
  - **Ord** típusosztály biztosítja az elemek összehasonlítását.

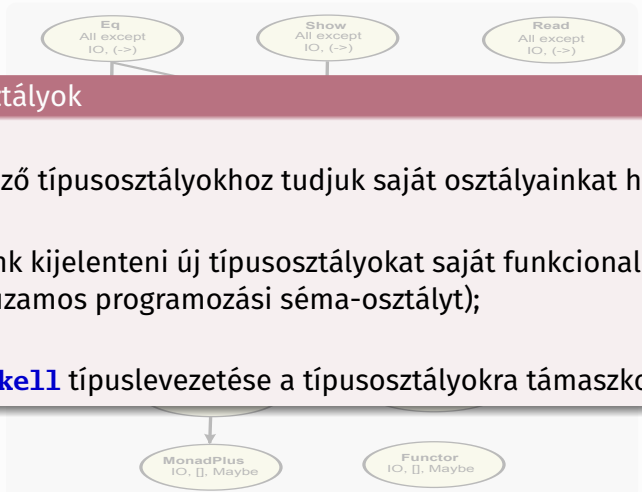
Haskell osztály-hierarchia

"Classes" by Dirk Hünniger - http://www.haskell.org/onlinereport Wikimedia Commons Lic.



Haskell osztály-hierarchia

"Classes" by Dirk Hünniger - http://www.haskell.org/onlinereport Wikimedia Commons Lic.



A típusosztályok

- A létező típusosztályokhoz tudjuk saját osztályainkat hozzáfűzni;
- Tudunk kijelenteni új típusosztályokat saját funkcionalitással (pl. párhuzamos programozási séma-osztályt);
- A **Haskell** típuslevezetése a típusosztályokra támaszkodik.

Algebrai típusok

- Saját típusokat tudunk kijelenteni:

```
data Napok = Hetfo |
             Kedd  |
             ...   |
             Vasarnap
```

  - A **data** kulcsszó új típust jelent;
  - típusnevek nagybetűsek, konstruktorokat definiálnak (felsorolás jelen esetben);

- Tudunk használni típusváltozókat. Pl:

```
data Tree a = Node (Tree a) a (Tree a)
            | Leaf
```

  - A típusdefiníció rekurzív;
  - Az **a** egy típusváltozó;
  - Fentebb egy bináris fának a definíciója.

Rekurzív típusok

- Típus-szinten definiáljuk a rekurziót;
- A rekurzív típusok kiváltják a pointer-eket;
- A kompilálás során a pointer-ek visszatérnek;
  azonban kódszinten nem kell kezelni,
  $\Rightarrow$ a megírt kód biztonságos.

Algebrai típusok használata

Minta-illesztéssel:

```
depth :: (Tree a) -> Int
depth Leaf = 0
depth (Node bal _ jobb)
   = 1 + max mbal mjobb
 where
   mbal = depth bal
   mjobb= depth jobb
```

- A `Leaf` és a `Node` . . . a konstruktor-kulcsok a minták;

A `case` utasítással:

```
depth :: (Tree a) -> Int
depth tree = case tree of
   Leaf -> 0
   (Node bal _ jobb) -> 1 + max mbal mjobb
  where
    mbal = depth bal
    mjobb= depth jobb
```

Példák algebrai típusokra I

- A `Maybe` típus:

```
data Maybe a = Just a | Nothing

-- használata
f x = case (maybef x) of
  Just ret -> g ret
  Nothing  -> hibakezel
```

  - A `Maybe` a típuskezelés egyik eszköze;
  - A `Nothing` ágra tehetünk hibakezelő függvényeket.

A `Maybe a` a típust egy elemmel terjeszti ki

Példák algebrai típusokra II

- Az `Ordering` típus:

```
data  Ordering
  =  LT | EQ | GT
  deriving
    (Eq, Ord, Bounded, Enum, Read, Show)
```

A `compare` függvény használja:

```
compare :: (Ord a) -> (Ord a) -> Ordering
compare x1 x2
   |   x1 == x2 = EQ
   |   x1 <= x2 = LT
   | otherwise  = GT
```

Példák algebrai típusokra III

- Az **Either** típus:

```
data Either a b =
   Left a  |
   Right b
```

A típusok „összege”

A **Left** konstruktorral az „a” típus, a **Right** konstruktorral a „b” típus elemeit tudjuk felsorolni.

Típusok kitekintő

- A típus(annotáció) a programok helyességvizsgálatának egyik eszköze.
- A „legkisebb” típus az `()` - ennek nincs értéke.
- Az alaptípusok és az azokon értelmezett operátorok – **`Int`**, **`Double`**, ...
- A származtatott típusok – pl. `[ (Int,[Bool]) ]`
- Az osztály függvénygyűjtemény; a típusokat „el kell látni” ezekkel.
- Az osztályok hierarchiát alkotnak.

Használjuk a típuskijelentéseket

A függvények típusai „jelzik” a szándékunkat:

```
double:: Num a => a -> a
```

Az Előadások Témái

- Imperatív/deklaratív programok, Prolog alapok
- Listák, listakezelés
- Aritmetikai műveletek és operátorok
- Vágások használata, negáció
- Gyűjtő-predikátumok Prolog-ban, összegzések
- Vég-rekurzió optimalizálás
- Dinamikus predikátumkezelés; input-output prolog-ban
- Kifejezések dekompozíciója
- Rendezések, kereső algoritmusok
- Funkcionális programozás bevezető, történelem
- A Haskell programnyelv elemei, példaprogramok
- Algebrai típusok, típusosztályok
- Magasabb-rendű függvények, Map-Reduce, rendezések
- Input-output
- *(opc) Algebrai struktúrák 2: Monádok stb*
- Ismétlések és kérdések megvitatása

Adminisztra ... ... trívia

link: Canvas learning management system (LMS)

Régi honlap: `http://www.cs.ubbcluj.ro/~csatol/log_funk`

Vizsga (félév elején) – lásd SYLLABUS!

Labor (40%) + Parciális (25%) + Kollokvium (35%)

Laborgyakorlatok:

1. 2 Prolog feladatcsoport **20%**
2. 2 Haskell feladatcsoport **20%**
3. $\aleph_0$ feladat **első n beküldőnek**

Listafeldolgozó műveletek / minták

- A listáknak – ill. a szekvenciális struktúráknak – a feldolgozása fontos tulajdonság;
- A Haskell-ben a listafeldolgozó műveleteket generikusan tudjuk kezelni;
- A listatranszformáció alapvető példája a függvények argumentumként történő kezelésének.
- Egy lista elemeit nem tudjuk cím szerint elérni; az index művelet rekurzív kódja (`infixl 9 !!`):

```
(!!) :: [a] -> Int -> a
[] !! _ = error "subscript"
(x:_) !! 0 = x
(_:xs)!! n = xs !! (n-1)
```

Lista-elemek szűrése – `filter`

A `filter`

- egy lista elemeit szűri egy predikátum szerint;
- lusta kiértékelésű;

```
filter:: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    |otherwise =     filter p xs
```

Listakonstruktor: `filter p lis = [x | x <- lis, p x]`

Például:

```
even x = x `mod` 2 == 0
evens  = filter even [2..]
```

Elemek átalakítása – map

A map

- egy lista elemein végez műveleteket
- lusta kiértékelésű

```
map :: (a -> b) -> [a] -> [b]
map   _    []    = []
map   f (x:xs) = f x : map f xs
```

Listakonstruktor: `map f lista = [f x | x <- lista]`

Listák iteratív „fogyasztása” – `fold(l/r)`

A `fold` függvény:

- Egy lista elemeit „göngyölíti” fel;
- A listakonstruktor műveletet – a `(:)`-t – helyettesíti egy bináris művelettel;
- Az üres-lista helyett egy kezdőelem lesz;

```
foldr:: (a -> b -> b) -> b -> [a] -> b
foldr _  e [] = e
foldr op e (x:xs) = x `op` foldr op e xs
```

Például: `foldr (*) 1 [2..5] = 2*(3*(4*(5*1)))`

- A zárójelezés (asszociációs sorrend) jobbról balra.

Fordítva: `foldl (*) 1 [2..5] = (((1*2)*3)*4)*5`

Feladat: vezessük le a `foldl` függvényt, majd annak típusát!

fold(l/r) alkalmazások

A fold(l/r) általános listafeldolgozási programozási minta:

- Egy lista legkisebb elemének a meghatározása:

```
minl (x:xs) = foldl min x xs
```

A min két elem minimuma.

- A két legnagyobb elem meghatározása:

```
ketmax (x1:x2:xs) = foldl nagyok (maxX, minX) xs
    where
    maxX = max x1 x2
    minX = min x1 x2
    nagyok (x1,x2) xn
        | xn > x1 = (xn,x1)
        | xn > x2 = (x1,xn)
        |otherwise= (x1,x2)
```

fold(l/r) alkalmazások

- Egy lista elemeinek az összege:
  ```
  sum = foldl (+) 0
  ```
- Két lista összefűzése:
  ```
  l1 ++ l2 = foldr (:) l2 l1
  ```
- Lista megfordítása:
  ```
  rev = foldl (\ l x -> x:l) []
  ```
- Példa: lista rendezése:

```
sort = foldl i_sort []
   where
      i_sort [] e = [e]
      i_sort (x:xs) e
         | e > x = x : i_sort xs e
         |otherwise = e:x:xs
```

Figyeljük meg a parciális paraméterezést!

Ciklus-iterátor – **until**

- Az $x_{n+1} = f(x_n)$ iterációt valósítja meg.
- A **cond** feltétel teljesítése esetén leáll (konvergencia).
- A programkód:

```
until cond f x0
    |   cond x0 = x0
    | otherwise = until cond f (f x0)
```

Ciklus-iterátor – until II

1. alkalmazás: Newton algoritmusa egy szám négyzetgyökének a kiszámítására.

Az iteráció (iter):

$$x_{n+1} = \frac{\left(x_n + \frac{a}{x_n}\right)}{2}$$

A kód:

```
newton a = until conv iter a
  where
    conv x = abs (x*x-a) < 1e-10
    iter x = (x+a/x)/2
```


A programnyelvekben ez a kód szerepel!

Ciklus-iterátor – until III

2. alkalmazás: Az a^b érték meghatározása ($a > 0$ esetre).

Az $a^b = \exp(b \ln a)$, ezért implementálni kell az $\exp$ és a $\ln$ függvényeket.

A kód:

```
(~^~):: (Fractional t, Ord t) => t -> t -> t
a ~^~ b
  | a > 0 = expo (b * ln a)

expo :: (Ord a, Fractional a) => a -> a
expo x = z
  where
  (z,_,it) = until felt iter (0,1,0)
  felt (_,an,_) = an == 0
  iter (sn,an,n) = (sn+an, an*x/(n+1), n+1)
```


Figyeljük meg a származtatott adatstruktúrák használatát!

A `map-reduce` paradigma

- Sok számítás végzésének párhuzamosítására használják;
- A `map` művelet függvényalkalmazásának függetlenségét használja ki;
- A `fold` segítségével számítjuk ki az eredményt;
  `fold(r/l) = reduce`
- A párhuzamosítás automatizált: a rendszer végzi el a párhuzamosítást, nekünk „pusztán” a kódunkat kell megfelelő formába hozni.

A map-reduce paradigma

Példák I

- Szeretnénk egy hosszú lista elemei négyzetének átlagát meghatározni:
  `sqAvg lis = (sum (\ x-> x*x) lis) / (length  lis)`

Két listabejárás...

- Segédstruktúrával:

```
sqAvg lis = ss/nn
 where
  (ss,nn) =
    foldl (\(x,x')(y,y')->(x+y,x'+y')) (0,0)
    $
    map (\x -> (x*x,1)) lis
```

Egy listabejárás, tail-recursion ...

- A `map f [x1,x2,...]` $\equiv$ `[(T f x1),(T f x2),...]`,
  ahol `T` egy szál, mely az `f` függvényt futtatja.

- **Gyorsítás:** feldaraboljuk a listát, majd listák listájára alkalmazzuk a számolást:

```
darabol _ [] = []
darabol n lis = take n lis : darabol n (drop n lis)
```

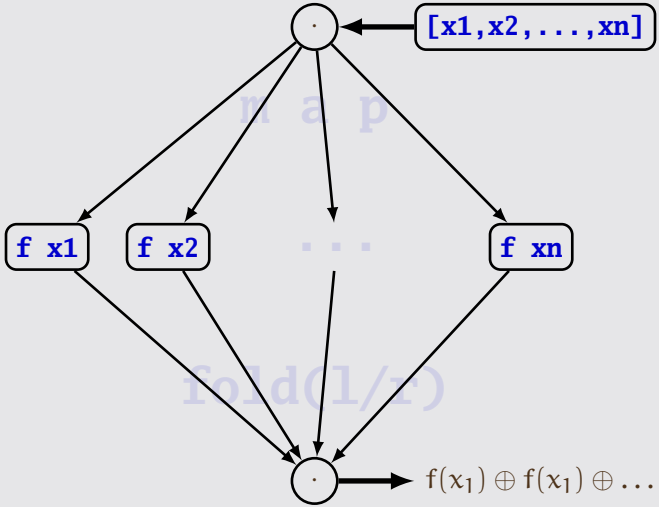
- Az új kód:

```
sqAvg2 n lis = ss/nn
 where
  (ss,nn) =
    compr     $       --  (ss,nn)  <- [(ss,nn)]
    map compr $       -- [(ss,nn)] <- [[(n,1)]]
    map parok $       -- [[(n,1)]] <- [  [n]  ]
    darabol n lis
  -- segédfüggvények
  compr= foldl (\(x,x')(y,y')->(x+y,x'+y')) (0,0)
  parok= map (\ x-> (x*x,1))
```

- (5-6) sor: a `map` művelet, a (4) sorban a `reduce`;

- A (7). sorban az előkészítés (pre-processing).

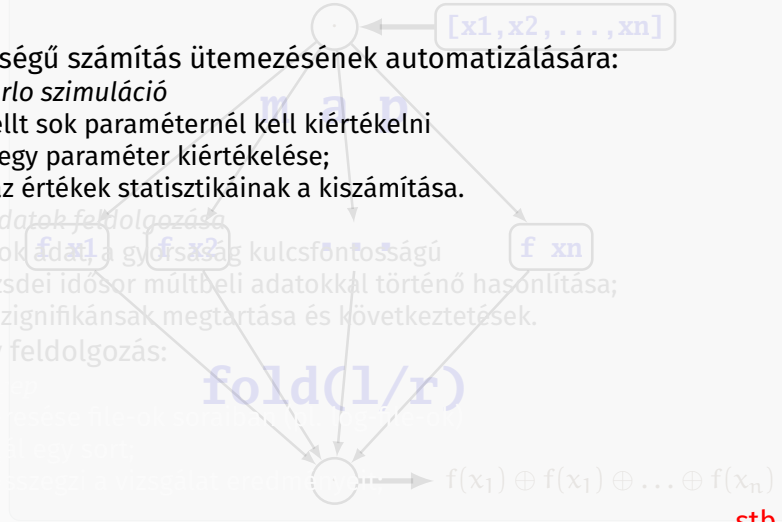
A map-reduce összefoglaló



A map-reduce alkalmazások

Alkalmazások:

- Nagy mennyiségű számítás ütemezésének automatizálására:
  - *Monte-Carlo szimuláció*
    Egy modellt sok paraméternél kell kiértékelni
    map: egy-egy paraméter kiértékelése;
    reduce: az értékek statisztikáinak a kiszámítása.
  - *Tőzsdei adatok feldolgozása*
    Nagyon sok adat, a gyorsaság kulcsfontosságú
    map: a tőzsdei idősor múltbeli adatokkal történő hasonlítása;
    reduce: szignifikánsak megtartása és következtetések.
- Adat-intenzív feldolgozás:

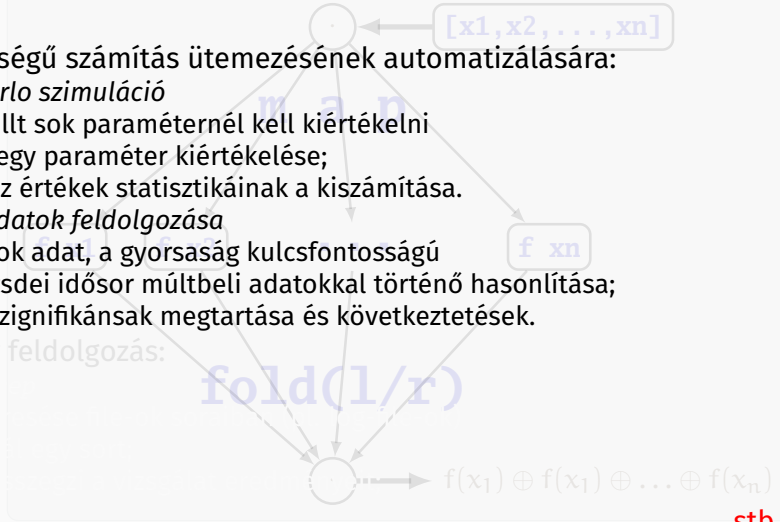


stb.

A map-reduce alkalmazások

Alkalmazások:

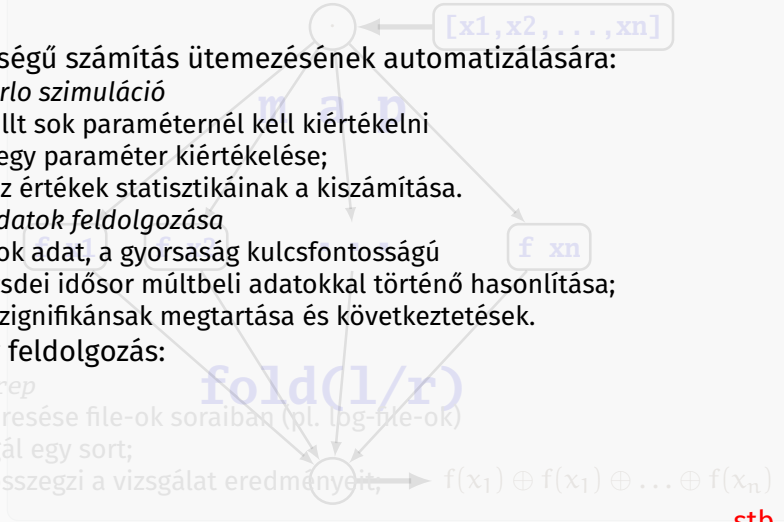
- Nagy mennyiségű számítás ütemezésének automatizálására:
  - *Monte-Carlo szimuláció*
    Egy modellt sok paraméternél kell kiértékelni
    map: egy-egy paraméter kiértékelése;
    reduce: az értékek statisztikáinak a kiszámítása.
  - *Tőzsdei adatok feldolgozása*
    Nagyon sok adat, a gyorsaság kulcsfontosságú
    map: a tőzsdei idősor múltbeli adatokkal történő hasonlítása;
    reduce: szignifikánsak megtartása és következtetések.
- Adat-intenzív feldolgozás:



A map-reduce alkalmazások

Alkalmazások:

- Nagy mennyiségű számítás ütemezésének automatizálására:
  - *Monte-Carlo szimuláció*
    Egy modellt sok paraméternél kell kiértékelni
    map: egy-egy paraméter kiértékelése;
    reduce: az értékek statisztikáinak a kiszámítása.
  - *Tőzsdei adatok feldolgozása*
    Nagyon sok adat, a gyorsaság kulcsfontosságú
    map: a tőzsdei idősor múltbeli adatokkal történő hasonlítása;
    reduce: szignifikánsak megtartása és következtetések.
- Adat-intenzív feldolgozás:
  - *Osztott grep*
    Minták keresése file-ok soraiban (pl. log-file-ok)
    map: vizsgál egy sort;
    reduce: összegzi a vizsgálat eredményeit;

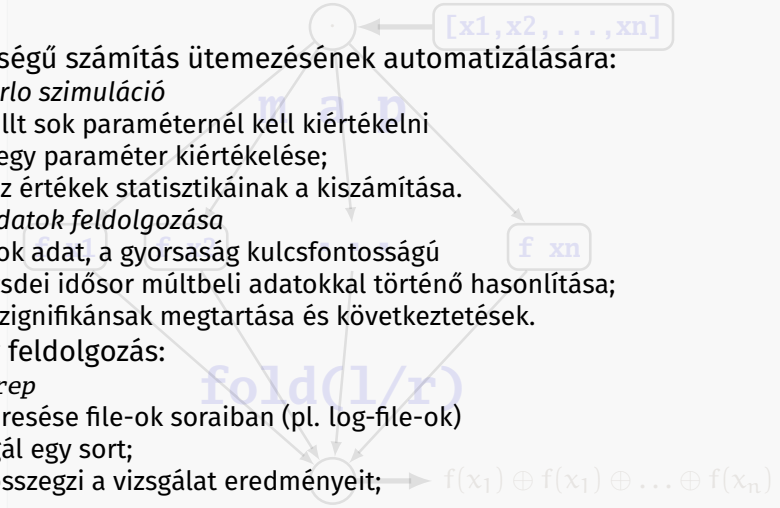


stb.

A map-reduce alkalmazások

Alkalmazások:

- Nagy mennyiségű számítás ütemezésének automatizálására:
  - *Monte-Carlo szimuláció*
    Egy modellt sok paraméternél kell kiértékelni
    map: egy-egy paraméter kiértékelése;
    reduce: az értékek statisztikáinak a kiszámítása.
  - *Tőzsdei adatok feldolgozása*
    Nagyon sok adat, a gyorsaság kulcsfontosságú
    map: a tőzsdei idősor múltbeli adatokkal történő hasonlítása;
    reduce: szignifikánsak megtartása és következtetések.
- Adat-intenzív feldolgozás:
  - *Osztott grep*
    Minták keresése file-ok soraiban (pl. log-file-ok)
    map: vizsgál egy sort;
    reduce: összegzi a vizsgálat eredményeit;



stb.

Rendező algoritmusok

- A funkcionális implementáció hasznos a futási idő megállapításánál;
- Lehet becsülni a bonyolultságot;
- Négyzetes futási idejű a **beszúrásos rendezés**;
- A **quicksort** átlagosan jól teljesít: $\mathcal{O}(N \log N)$;
- Az összefésüléses rendezés mindig $\mathcal{O}(N \log N)$ idejű;
- Nagy lista és véges – kevés – kulcs esetén lineáris idejű rendezés.

Beszúrásos rendezés

```
isort = foldl beszurrend []
  where beszurrend [] e = [e]
        beszurrend (x:xs) e
          | e>x = x : beszurrend xs e
          |otherwise= e:x:xs
```

Típus?

- A lista elemeit egyenként dolgozza fel; $\Leftarrow$ **fold**
- Az aktuális elemet **beszúrja** a rendezett részsorozatba;
- A kezdő részsorozat üres $\Leftarrow$ **[]**
- A számításigény:

$$N^{\texttt{foldl}} \cdot \left(\frac{N}{2}\right)^{\texttt{beszurrend}} = \mathcal{O}(N^2)$$

Gyorsrendezés – quicksort

```
qsort:: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = rkis ++ x:rnagy
  where
    rkis  = qsort [e | e<-xs, e<x]
    rnagy = qsort [e | e<-xs, e>x]
```

- A lista első elemét kiemeli; ⇐ (x:xs)
- Kiválassza a kisebbeket; ⇐ [e|e<-xs,e<x]
- Kiválassza a nagyobbakat; ⇐ [e|e<-xs,e>x]
- Rendezi a két részsorozatot: ⇐ rekurzió
- Az átlagos számításigény $T(N)$:

$$T(N) = \left(\frac{N}{2}\right)^{(++)} + T\left(\frac{N}{2}\right)^{rkis} + T\left(\frac{N}{2}\right)^{rnagy} = \cdots \mathcal{O}(N \cdot \log N)$$

Összefésüléses rendezés

```
f_rend:: Ord a => [a] -> [a]
f_rend [] = []
f_rend [x]= [x]
f_rend lis = fesul (frend lis1) (f_rend lis2)
  where
    (lis1,lis2) = oszt lis
```

- A listát kettéosztja; $\Leftarrow$ **oszt**
- Rendezi a két részsorozatot: $\Leftarrow$ rekurzió
- Összefésüli a két rendezett sorozatot; $\Leftarrow$ **fesul**
- Az átlagos számításigény $T(N)$:

$$T(N) = N^{\textbf{oszt}} + N^{\textbf{fesul}} + 2 \cdot T\left(\frac{N}{2}\right)^{\textbf{rek}} \cdots = \mathcal{O}(N \cdot \log N)$$

Összefésüléses rendezés II

Segédfüggvények:

```
oszt::[a] -> ([a],[a])
oszt [] = ([],[])
oszt [x]= ([x],[])
oszt (x1:x2:xs) = (x1:xs1,x2:xs2)
  where
    (xs1,xs2) = oszt xs


fesul:: Ord a => [a] -> [a] -> [a]
fesul [] lis = lis
fesul lis [] = lis
fesul (x1:xs1) (x2:xs2)
  | x1<x2  =  x1 : fesul xs1 (x2:xs2)
  |otherwise= x2 : fesul (x1:xs1) xs2
```

Az Előadások Témái

- Imperatív/deklaratív programok, Prolog alapok
- Listák, listakezelés
- Aritmetikai műveletek és operátorok
- Vágások használata, negáció
- Gyűjtő-predikátumok Prolog-ban, összegzések
- Vég-rekurzió optimalizálás
- Dinamikus predikátumkezelés; input-output prolog-ban
- Kifejezések dekompozíciója
- Rendezések, kereső algoritmusok
- Funkcionális programozás bevezető, történelem
- A Haskell programnyelv elemei, példaprogramok
- Algebrai típusok, típusosztályok
- Magasabb-rendű függvények, Map-Reduce, rendezések
- Input-output
- *(opc) Algebrai struktúrák 2: Monádok stb*
- Ismétlések és kérdések megvitatása

Adminisztra ... ... trívia

link: **Canvas learning management system (LMS)**

Régi honlap: `http://www.cs.ubbcluj.ro/~csatol/log_funk`

Vizsga (félév elején) – lásd SYLLABUS!

Labor (40%) + Parciális (25%) + Kollokvium (35%)

Laborgyakorlatok:

1. 2 Prolog feladatcsoport **20%**
2. 2 Haskell feladatcsoport **20%**
3. $\aleph_0$ feladat **első n beküldőnek**

Függvények, mellékhatások I

A függvények definíciója (matematika)

Egy függvény az x bemenetre egy y kimenetet „produkál”:

$$y = f\ x$$

- Egy bemenetre, mindig mindig ugyanaz a **kimenet**.
- Nem minden „informatikus” függvény ilyen; pl. a `readChar` függvénynek nincs bemenete és sok lehetséges kimenete van.
- Függvényeink mellékhatás-mentesek voltak.

Mellékhatásokat „kezelő” függvények:

- Általában a bemenetért / kimenetért felelős függvények ilyenek.
- Ha egy karaktert be akarunk olvasni:

```
getChar :: IO Char
```

- Ha egy karaktersort meg akarunk jeleníteni:

```
putStrLn :: String -> IO ()
```

Az IO típus/osztálykonstruktor

- Jelzi, hogy az argumentum „kintről” érkezik.
- A `Maybe`-hez hasonlóan, az érték be lett csomagolva;
- A lista egy-egy elemét kivesszük – a `!!` operátorral, illetve pl a `head` függvénnyel.

Impure

Az olyan programokat, melyek „bemeneteket” kezelnek, általában nem tiszta függvénynek nevezzük.

Haskell-ben:

```
getChar ::  IO Char
```

- Az `IO` algebrai típus, mely azt jelzi, hogy a függvény „tartalmaz” mellékhatást;
- Ezt a mellékhatást fel tudjuk dolgozni;

Hasonlóan a `Maybe` típushoz...

Függvények, mellékhatások IV

- Az **IO** típusmódosító bemenet-kimenet műveleteket jelent[13]
- Az **IO** osztály műveleteit is típusosztályok segítségével adjuk meg.
- Ez a típusosztály a monád.

A függvényeket „le kell futtatni”

- Tíz karakter beolvasását felsoroljuk egy listában:

```
take 10 $ repeat getChar
```

  azonban a lista függvényeit végre kell hajtani.
- Erre van a **do** szintaxis.

[13] A Haskell-beli **IO** függvényeket definiál.

A „do” jelölés

A do függvénnyel:

- IO műveleteket helyezünk egymás után;

  `do {a1; a2; ... an}`

- Az `a1; a2; ... an` IO-függvényeket szekvenciálisan végezzük el.
- Példa:

  ```
  do { putStr "Elso resz,"; putStr " masodik resz."}
  ```

  eredménye `"Elso resz, masodik resz."`.
- Egyszerűsített jelölést is használhatunk, ahol egymás alá írjuk a függvényeket (hasonlóan a **where** kulcsszóhoz).

A „do” jelölés II

Az egyszerűsített **do** parancs:

```
do
   action1
   action2
   action3

-- alternatívák

do
   x1 <- action4
   action5
   let x2 = function7
   action7
   return x3

-- fontos az ‘‘indent’’
```

Jellemzők:

- Mindegyik függvény kimenete **IO**-t tartalmaz.
- Például a
  `putChar :: Char -> IO ()`
  `putStrLn :: String -> IO ()`
- „IO-sító” függvény pl a (az IO egy monád)
  `return :: Monad m => a -> m a`
  A **do** testében: `return :: a -> IO a`
- A balranyíl - „<-” - kicsomagolja az **IO**-ból az értéket;
- A `let v = ...` konstrukció egy függvény értékét teszi elérhetővé;
- A **return** a kilépés a **do** blokk-ból – az **x3** lehet a definiált **x1**... változókat tartalmazó kifejezés is.

Példa:

```haskell
nameDo :: IO ()
nameDo = do
    putStr "First name: "
    fN  <-  getLine
    putStr "Last name: "
    lN  <-  getLine
    let full = fN ++ " " ++ lN
    putStrLn $ "Hello " ++ full
```

```haskell
nameLm :: IO ()
nameLm = putStr "First name: " >>
    getLine >>= \ fN ->
    putStr "Last name: " >>
    getLine >>= \ lN ->
    let full = fN ++ " " ++ lN
    in putStrLn $ "Hello " ++ full
```

- A két függvény ekvivalens;
- A **do** jelölés egyszerűsíti a programozást.
- Az `()` az „üres típus”.

A „do”

Példa II

Példa visszatérített értékre:

```
nameRet :: IO String
nameRet  =  do
    putStr "First name: "
    fN  <-  getLine
    putStr "Last name: "
    lN  <-  getLine
    let full = fN ++" "++ lN
    return full
```

```
import System.Random
-- {{ :set -package random }}

newRand = randomIO :: IO Int

randList n
    | n > 0 = randList_ (n,[])
    | otherwise = return []

randList_ (0,xs) = return xs
randList_ (n,xs) = do
    x <- newRand
    randList_ (n-1,x:xs)
```

Típus-aláírások:

```
 newRand  :: IO Int
randList  :: (Ord a, Num a) => a -> IO [Int]
randList_ :: (Eq a, Num a) => (a, [Int]) -> IO [Int]
```