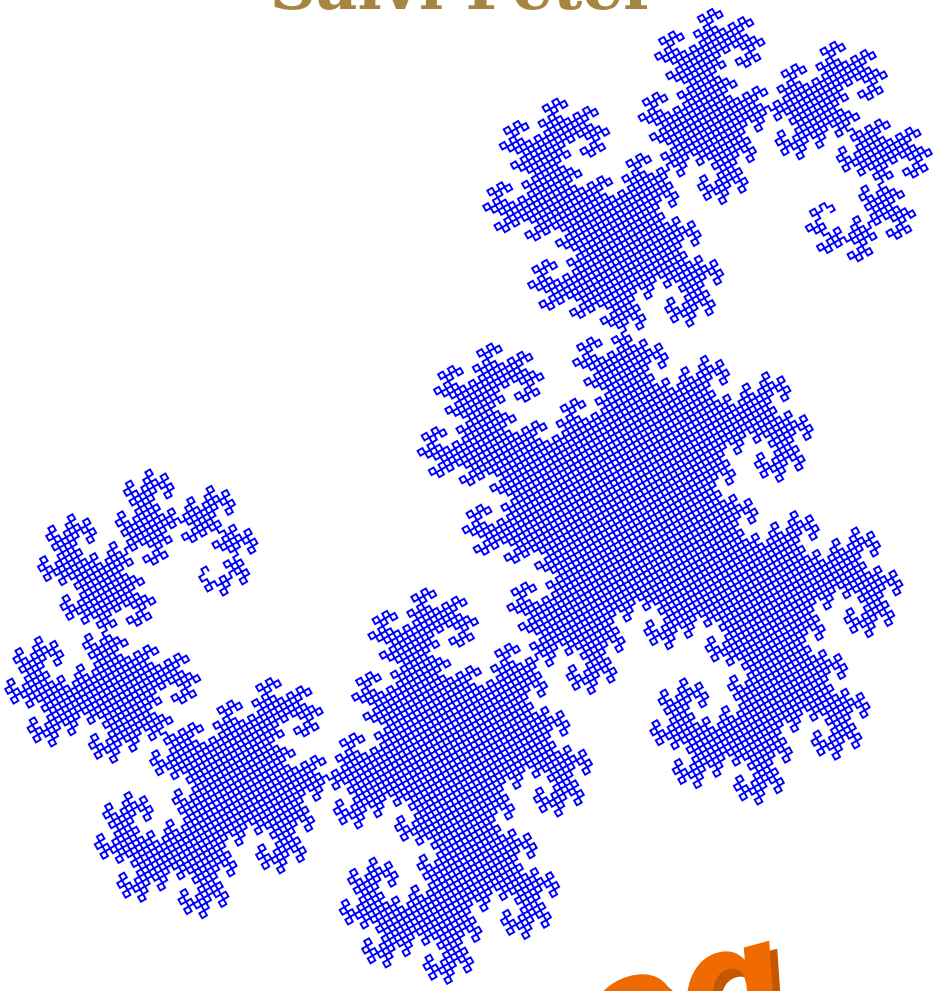
Salvi Péter



Oldd meg Prologban!

OLDD MEG PROLOGBAN!

OLDD MEG PROLOGBAN!

Salvi Péter

2022

Salvi Péter: Oldd meg Prologban!
Budapest, 2022
(2022-03-27 / 9c877ff verzió)


```
sárkány(N) :- sárkány(N, 'jobbra át').

sárkány(0, _) :- write('lépj egyet'), nl.
sárkány(N, X) :-
    N > 0, N1 is N - 1,
    sárkány(N1, 'jobbra át'),
    write(X), nl,
    sárkány(N1, 'balra át').

?- sárkány(14). % 16384 lépés
```

Ez a könyv LaTeX-ben készült, Computer Modern betűtípussal.
A borítón egy Lindenmayer-rendszer, az ún. *sárkány görbe* látható.

Zsófinak és Julinak

Tartalomjegyzék

Előszó

Ez a könyv elsősorban azoknak szól, akik még egyáltalán nem tudnak programozni, de szeretnék megismerni a számítógépes problémamegoldás alapjait. A „programozás” ma már egy rendkívül szerteágazó gyűjtőfogalommá vált, melynek egy-egy területe magában egy-egy szakmát képvisel. Itt nem lesz szó webfejlesztésről, látványos játékokról vagy neurális hálókról: kizárólag a problémamegoldásra fogunk koncentrálni, ami viszont – az én véleményem szerint – a programozás legizgalmasabb válfaja.

Ehhez a Prologot fogjuk használni. Ez az 50 éves múltra visszatekintő programozási nyelv sokáig állt a mesterséges intelligencia kutatás középpontjában, és bár az iparban nem hódított teret, az informatika oktatásában továbbra is jelentős szerepe van. Ez nem véletlen: a Prolog különösen alkalmas tanításra, ugyanis egészen minimális ismeretekkel is már érdekes feladatok megoldását teszi lehetővé. Ezt bizonyítják az egyes leckéket lezáró *projektek* is, amelyekben különböző, kicsit nagyobb lélegzetvételű problémákat vizsgálunk majd meg.

Hogyan használjuk a könyvet?

A leckékben szereplő példákat érdemes a számítógépen kipróbálni, és kísérletezgetni velük, átírni bennük ezt–azt, és megvizsgálni, hogy mi történik. Az efféle játék során mélyül el igazán a tudás; és ezt segítik a feladatok is, melyeknek megoldása a könyv végén található.

Mindehhez természetesen szükség van egy Prolog fordítóra. Ezekből sok létezik, és mindegyik meglehetősen különböző. A legtöbb azonban kompatibilis az 1995-ös ISO szabvánnyal, és ez a könyv csak ezt feltételezi, illetve még annyit, hogy tudja kezelni a magyar ábécé betűit. Az ingyenesen letölthető, nyílt forráskódú rendszerek közül ilyen az SWI-Prolog. Más implementációk, mint pl. az XSB vagy a GNU Prolog, jelenleg nem fogadják el az ékezetes karaktereket, ezért ezek használata esetén a programokat ékezetek nélkül kell begépelni.

A problémamegoldás Prologban mindig két részből áll:

1. Egy – általában `.pl` vagy `.P` kiterjesztésű – fájlban megadunk egy szabályrendszert (a program *forráskódját*). Ennek megírásához tetszőleges szövegszerkesztőt használhatunk.

2. Ezzel kapcsolatban kérdéseket teszünk fel a programnak.

A legtöbb Prolog rendszer indításkor mindössze egy kérdőjel kiírásával fogad minket; ezzel jelzi, hogy várja a kérdést:

```
?-
```

Az első feladatunk, hogy betöltsük az általunk írt programot. Ezt általában a `consult` paranccsal tehetjük meg, például:

```
?- consult('/home/salvi/szabályok.pl').
```

. . . ahol az aposztrófok között levő `/home/salvi/szabályok.pl` a forráskód teljes elérési útja. Ne felejtsük le a parancs végéről a pontot, majd nyomjuk meg az újsor billentyűt.

Ezután már jöhetnek a kérdések. A leckékben már csak ezek lesznek feltüntetve, de nem szabad megfeledkezni a szabályok betöltéséről sem. Amennyiben egy kérdésre több válasz is van, a következő választ általában a pontosvessző (`;`) lenyomásával kaphatjuk meg, a következő kérdés feltevéséhez pedig ilyenkor egy újsor billentyűt kell nyomni.

Bizonyos rendszerekben mindez kényelmesebben is megtehető, így pl. az SWI-Prolog internetes SWISH verziójában,[†] ahol a szabályok mindig automatikusan újraolvasódnak.

Források

Amikor elkezdtem ezeket a leckéket írni (eredetileg a honlapomra feltéve, blogposzt-jellegűen), a célom csak az volt, hogy az egyik kedvenc programozásról szóló könyvemet [1] elérhetőbbé és könnyebben emészthetővé tegyem a magyar laikus közönség számára. Ez aztán végül egy Prolog tankönyvvé vált, de a leckék – sokszor jelentősen átdolgozva, de néhol szinte szó szerint – az eredetit követik. Egészen pontosan az 1. lecke az 1.1–3 és 1.8, a 2. lecke a 2.1–6, a 3. lecke a 3.1–2, a 4. lecke a 3.3–4, az 5. lecke az 5.1–4 és 9.1, a 6. lecke pedig a 6.1–6 és 8.5 fejezeteknek felel meg.

[†] `https://swish.swi-prolog.org/`

Egy másik fontos forrás egy klasszikus Prolog tankönyv [2]; ebből származik az 5. lecke összefésüléses példája (11. fejezet), a 6. leckében a különbség-listák tárgyalása (15.1 fejezet), és a lecke végén található projekt (20.2 és 21.3 fejezetek).

A 4. leckében levő kígyó-kocka ötletét egy Haskell megoldás [3] adta; a 7. lecke alapját képező forráskód pedig az SWI-Prolog szoftvercsomag része.

[1] I. Bratko: *Prolog Programming for Artificial Intelligence*, 4th Ed., Pearson, 2011.

[2] L. Sterling, E. Shapiro, *The Art of Prolog*, 2nd Ed., MIT Press, 1994.

[3] M. P. Jones: *Solving the Snake Cube Puzzle in Haskell*. Journal of Functional Programming 23(2), pp. 145–160, 2013.

Köszönetnyilvánítás

Köszönettel tartozom mindenekelőtt Varga Csillának, aki lelkesen végigcsinálta a könyvnek egy korai változatát, és kérdéseivel rávilágított arra, hogy mit kell még pontosítanom vagy jobban elmagyaráznom; és természetesen a feleségemnek, Iida Norikónak, akinek a támogatása nélkül ez a könyv nem készülhetett volna el.

Salvi Péter
Budapest, 2022.

1. lecke

Tények és szabályok

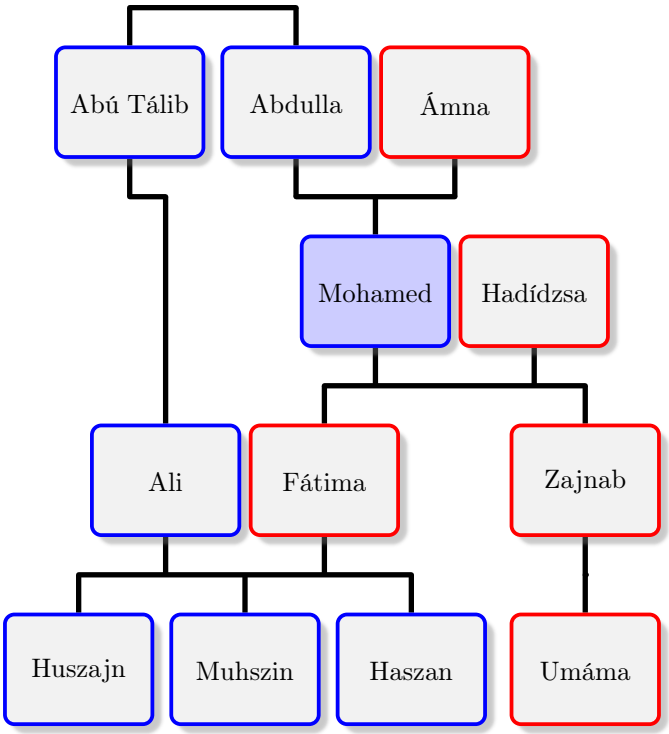
Az első leckében egy családfa vizsgálatán keresztül megismerkedünk a programok alkotóelemeivel és a *rekurzió*val.

Tények

Egy Prolog program *tényekből* és *szabályokból* áll, amelyekből a számítógép következtetéseket tud levonni. Ha megadunk egy tény- és szabályrendszert, utána kérdéseket tehetünk fel, amelyeket a gép legjobb tudása szerint megválaszol.

Példaként vegyük Mohamed próféta családját (1.1. ábra)! A szülő–gyerek kapcsolatokat az alábbi program foglalja össze:

```
1 szülő(abú_tálib, ali).
2 szülő(abdulla, mohamed).
```



1.1. ábra. Mohamed próféta családja (részlet).

```
3  szülő(ámna, mohamed).
4  szülő(mohamed, fátima).
5  szülő(hadídzsa, fátima).
6  szülő(mohamed, zajnab).
7  szülő(hadídzsa, zajnab).
8  szülő(ali, huszajn).
9  szülő(fátima, huszajn).
10 szülő(ali, muhszin).
11 szülő(fátima, muhszin).
12 szülő(ali, haszan).
13 szülő(fátima, haszan).
14 szülő(zajnab, umáma).
```

Ebben a programban minden sor egy *tény*. Egy tény dolgok (itt emberek) közti kapcsolatot ír le. A formája a következő: a kapcsolat nevével kezdődik (most ez a `szülő`), utána zárójelek között vesszővel elválasztva felsoroljuk a kapcsolatban levő dolgokat, és a végén egy ponttal (`.`) zárjuk.

Talán furcsa lehet, hogy minden kisbetűvel szerepel – majd később lesz szó arról, hogy mik az elnevezés pontos szabályai, egyelőre azt jegyezzük meg, hogy minden *konkrét* dolog kisbetűvel írandó, és nem lehet benne szóköz.

Egyszerű kérdések

Már egy ilyen egyszerű program esetén is lehet értelmes kérdéseket feltenni. Például megkérdezhetjük, hogy

```
?- szülő(mohamed, fátima).
```

Kitérő: *Mohamed családfája*

A programban a családfának csak egy nagyon kis szeletét használtuk, és még ezen a részen belül sem tartalmaz minden kapcsolatot, mert pl. Ali Fátima halála után feleségül vette Umámát is.

... amire a rendszer a `true` (igaz) vagy `yes` (igen) üzenettel válaszol; vagy hogy

```
?- szülő(ali, zajnab).
```

... amire a `false` (hamis) vagy `no` (nem) eredményt adja. (A továbbiakban ez a könyv a `true` és `false` válaszokat fogja használni.)

Egy kicsit érdekesebb a következő kérdés:

```
?- szülő(mohamed, Kicsoda).
```

Erre azt a feleletet kapjuk, hogy `Kicsoda = fátima`. Ha további válaszokat kérünk, akkor még azt is megmondja, hogy `Kicsoda = zajnab`.

Mi történt itt? Azáltal, hogy a második helyre egy nagybetűs szót (`Kicsoda`) írtunk, azt mondtuk, hogy ez egy határozatlan, ismeretlen érték, egy *változó*. A kérdést tehát magyarul úgy lehetne megfogalmazni: „Mohamed kinek a szülője?"

Erre megkeresi az első választ, amit talál (`fátima`), és ha továbbit kérünk tőle, akkor megtalálja `zajnab`ot is, és látja, hogy nincs több, és így leáll.

Ha kisbetűvel írtuk volna:

```
?- szülő(mohamed, kicsoda).
```

... akkor a `false` eredményt kaptuk volna, hiszen `kicsoda` itt egy konkrét dolgot jelöl, a kérdés tehát azt jelenti: „Igaz-e, hogy Mohamed szülője Kicsodának?"

A kérdés a másik irányban is feltehető:

```
?- szülő(Ki, ali).
```

. . . tehát „Ki Ali szülője?”, amire a `Ki = abú_tálib` feleletet kapjuk.

Még tovább is mehetünk, és rákérdezhetünk az összes szülő-gyerek kapcsolatra:

```
?- szülő(Ki, Kinek).
```

. . . azaz „Ki kinek a szülője?”. Az első válasz az lesz, hogy

```
Ki = abú_tálib,
Kinek = ali
```

További válaszok kérésével sorban megkapjuk az adatbázisunk minden egyes elemét.

Összetett kérdések

Tegyük fel, hogy arra vagyunk kíváncsiak, hogy kik Mohamed unokái. Hogyan tudjuk ezt megkérdezni? Az unoka definíció szerint a gyerek gyereke, tehát tudjuk, hogy van valaki, aki szülője az unokának, és akinek a szülője Mohamed. A logikai *és*sel összekötött, együtt teljesítendő feltételeket vesszővel választjuk el:

```
?- szülő(mohamed, Valaki), szülő(Valaki, Unoka).
```

Négy megoldást is kapunk:

```
Unoka = huszajn,
Valaki = fátima ;
Unoka = muhszin,
Valaki = fátima ;
```

```
Unoka = haszan,
Valaki = fátima ;
Unoka = umáma,
Valaki = zajnab
```

(Ezek páronként értendők, tehát Fátimától van Huszajn, Muhszin és Haszan, és Zajnabtól Umáma.)

Ez azért működik, mert egy változónak minden előfordulása azonos értéket kell, hogy kapjon, tehát a két `szülős` kifejezésben a `Valaki` ugyanarra vonatkozik. A kérdés két tagjának sorrendje felcserélhető, tehát ugyanezt az eredményt adja ez is:

```
?- szülő(Valaki, Unoka), szülő(mohamed, Valaki).
```

(Majd látni fogjuk viszont, hogy a számításigényük nem azonos, érdemesebb az erősebb megszorítással kezdeni.)

Hasonlóan rákérdezhetünk, hogy kik Huszajn nagyszülei:

```
?- szülő(Valaki, huszajn), szülő(Nagyszülő, Valaki).
```

... amire megkapjuk Abú Tálibot, Mohamedet és Hadídzsát.

Ha arra vagyunk kíváncsiak, hogy Haszan testvére-e Huszajnnak, így fogalmazhatjuk meg:

```
?- szülő(Valaki, haszan), szülő(Valaki, huszajn).
```

Azt kapjuk, hogy `Valaki = ali`, tehát a válasz igen. Ha `huszajn` helyett `abdullát` írunk, akkor `false` választ kapunk.

1. Feladat. Válaszold meg az alábbi kérdéseket!

```
?- szülő(huszajn, X).
?- szülő(X, huszajn).
?- szülő(ámna, X), szülő(X, fátima).
?- szülő(ámna, X), szülő(X, Y), szülő(Y, haszan).
```

2. Feladat. Fogalmazd meg Prologban!

1. Ki Ali szülője?
2. Umámának van gyereke?
3. Ki Zajnab nagyszülője?

3. Feladat. Készítsd el a saját családfádat (nagyszülőkig és unokatestvérekig)!

`false` = *nincs több megoldás*

A Prolog gyakran olyankor is felajánlja a további megoldások keresését, amikor nincs több – egyszerűen azért, mert még *ő* sem tudja. Ha ezután mégiscsak kiderül, hogy nincs más megoldás, akkor ezt `false`-szal jelzi. Bizonyos esetekben ez megkerülhetetlen, máskor az adott rendszertől függ, hogy észreveszi-e, hogy felesleges a további keresés.

Szabályok

Eddig csak tényekkel foglalkoztunk, de valójában a Prolog programok nagy része *szabályokból* áll. Először is egészítsük ki a programunkat a szereplőink nemével!

```
férfi(abú_tálib).
férfi(abdulla).
férfi(mohamed).
férfi(ali).
férfi(huszajn).
férfi(muhszin).
férfi(haszan).

nő(ámna).
nő(hadídzsa).
nő(fátima).
nő(zajnab).
nő(umáma).
```

Ha most a `szülő` mellett szeretnénk `anya` és `apa` kapcsolatokat is létrehozni, ezt megtehetjük egyenként:

```
anya(ámna, mohamed).
anya(hadídzsa, fátima).
...
apa(abú_tálib, ali).
apa(abdulla, mohamed).
...
```

Ez azonban elég sok munka, és érezzük, hogy felesleges, hiszen kikövetkeztethető. Szabályok segítségével ezt sokkal egyszerűbben megoldhatjuk:

```
anya(X, Y) :- szülő(X, Y), nő(X).
apa(X, Y) :- szülő(X, Y), férfi(X).
```

A `:-` jelet itt úgy olvashatjuk ki, hogy „akkor, ha”, tehát „X anyja Y-nak akkor, ha X szülője Y-nak és X nő”. A baloldalon levő részt a szabály *fej*ének, a jobboldalát a szabály *törzs*ének nevezzük.

Mi történik, amikor feltesszük az alábbi kérdést?

```
?- anya(ámna, mohamed).
```

A rendszer megtalálja az `anya` szabályt, és *egyesíti* az `X`-et `ámná`-val, az `Y`-t pedig `mohamed`del. (Az egyesítésről még később szó lesz, itt egyszerűen helyettesítést jelent.) Ezután megnézi, hogy a szabály jobboldalán levő feltétel teljesül-e; ez most a

```
szülő(ámna, mohamed), nő(ámna).
```

... és ezek a tények szerepelnek a programban, tehát `true` válasszal tér vissza.

Definiáljuk a „nagyszülő” kapcsolatot!

```
1 nagyszülő(X, Z) :- szülő(X, Y), szülő(Y, Z).
```

Ez pontosan követi azt, ahogy korábban megkerestük valakinek a nagyszülőjét.

Problémás esetek

Hogyan tudnánk megadni a „fivér” kapcsolatot?

```
1 fivér(X, Y) :- férfi(X), szülő(Z, X), szülő(Z, Y).
```

Tehát X fivére Y-nak, ha férfi és van közös szülője Y-nal.

Itt érdemes megjegyezni, hogy bár Abú Tálib és Abdulla testvérek voltak, a

```
?- fivér(abú_tálib, abdulla).
```

kérdésre `false` a válasz, mivel a programnak nincs arról tudomása, hogy lenne közös szülőjük. (A Prolog mindent hamisnak vesz, amit az általa ismert adatokból nem tud kikövetkeztetni, és ez időnként furcsa következményekkel járhat – erről majd később.)

Egy másik problémába ütközünk, ha Mohamed fivéreire vagyunk kíváncsiak:

```
?- fivér(mohamed, X).
```

Az eredmény, meglepő módon, `X = mohamed`! Ebből látszik, hogy a definíciónk nem volt elég pontos; hozzá kell venni azt is, hogy senki nem fivére önmagának:

```
1 fivér(X, Y) :-
2     férfi(X),
3     szülő(Z, X), szülő(Z, Y),
4     X \= Y.
```

Itt a `\=` jelentése „nem azonos”. Ez a példa azt is illusztrálja, hogy a szabályok több sorban is írhatóak; ilyenkor a második és további sorokat beljebb szokás húzni.

Többszörös megoldások

A „`?- fivér(huszajn, muhszin).`” kérdésre érdekes módon két `true` választ kapunk. Ennek az az oka, hogy a fivéri kapcsolatot kétféleképpen is be lehet bizonyítani, Alin és Fátimán keresztül.

Egyedüli változók

Készítsük el a

```
1 vangyereke(X) :- szülő(X, Y).
```

szabályt. Ennek az az érdekessége, hogy attól függően, hogy hogyan olvassuk ki, az állítás *minden* Y-ra vonatkozik, vagy csak azt mondja, hogy *létezik* egy fajta Y:

1. (Minden X és Y esetén,) ha X szülője Y-nak, akkor X-nek van gyereke.

2. X-nek akkor van gyereke, ha létezik olyan Y, hogy X szülője Y-nak.

A két olvasat egyenértékű; ez a kettősség annak a következménye, hogy az `Y` változó csak a szabály törzsében fordul elő.

A fenti példa más miatt is érdekes. Mivel az `Y` változó a szabályban mindössze egyszer fordul elő, nem lehet hatással az eredmény kiszámítására, tehát érdektelen. Ezt úgy szokás jelezni, hogy a helyére egy alsóvonást (`_`) írunk, vagy egy ezzel kezdődő nevet (pl. `_Y`). Ha kérdésben szerepel ilyen változó, akkor a válaszban ennek az értéke nem fog megjelenni. Így pl. az alábbi kérdés megadja Mohamed unokáit anélkül, hogy megmutatná a szülőket:

```
?- szülő(mohamed, _Valaki), szülő(_Valaki, Unoka).
```

Ha több alsóvonás szerepel, ezek értéke lehet különböző, tehát a

```
?- szülő(_, _).
```

kérdésre `true` lesz a válasz, pedig senki sem szülője önmagának.

Ha egy egyszer szereplő változónak „normális” nevet adunk, akkor erre a fordító általában figyelmeztet minket, mert ez gyakran egy elírás következménye. Pl. az alábbi programban a `Valaki` változóból egyszer kimaradt egy `a` betű:

```
1 nagyszülő(X, Y) :-
2     szülő(X, Valaki), szülő(Valki, Y).
```

4. Feladat. Fordítsd le Prologra!

1. Akinek van gyereke, az boldog. (`boldog` szabály)
2. Minden X-re, ha X-nek van egy gyereke akinek van egy fivére, akkor X-nek két gyereke van. (`kétgyerekes` szabály)

5. Feladat. Készítsd el az `unoka` szabályt! Teszteld a saját családfádon!

6. Feladat. Írj egy `nagynéni` szabályt a `szülő` és `nővér` segítségével, és keresd meg vele Zajnab unokaöccseit!

1.1. Rekurzió

Adjunk még egy utolsó szabályt a programunkhoz: az *ős* fogalmát. Valakinek az őseit úgy kapjuk meg, hogy felfelé megyünk a családfában: ős a szülő, a nagyszülő, a dédszülő stb. Ezt elkezdhetjük írni szabályokkal:

```
ős(X, Z) :- szülő(X, Z).
ős(X, Z) :- szülő(X, Y), szülő(Y, Z).
ős(X, Z) :-
    szülő(X, Y1), szülő(Y1, Y2), szülő(Y2, Z).
ős(X, Z) :-
    szülő(X, Y1), szülő(Y1, Y2),
    szülő(Y2, Y3), szülő(Y3, Z).
...
```

Ez nagyon jól működik, de véges – bármennyi ilyen programsort írunk, mindig feljebb tudunk menni eggyel a családfában, és azt már nem kezeli.

Egy kis trükkel meg tudjuk ezt oldani: azt mondjuk, hogy ha X gyereke őse Z-nek, akkor X is őse Z-nek:

```
ős(X, Z) :- szülő(X, Y), ős(Y, Z).
```

Az `ős` így önmagára hivatkozik, más szóval *rekurzív*.

Ez így magában még azonban nem elég, mert így ahhoz, hogy valaki ős legyen, mindig valaki másnak is ősnek kéne lennie, amit a végtelenségig kéne folytatnunk. Valahol ennek meg kell állnia; ehhez elég hozzávenni a legegyszerűbb esetet, vagyis azt, amikor maga a szülő az ős:

```
ős(X, Z) :- szülő(X, Z).
ős(X, Z) :- szülő(X, Y), ős(Y, Z).
```

Ez a kettő együtt már működik. X őse Z-nek, ha vagy (i) X szülője Z-nek, vagy (ii) X szülője Y-nak és Y őse Z-nek.

Próbáljuk is ki:

```
?- ős(X, huszajn).
X = ali ;
X = fátima ;
X = abú_tálib ;
X = abdulla ;
X = ámna ;
X = mohamed ;
X = hadídzsa
```

7. Feladat. Tegyük fel, hogy az `ős` definícióját megváltoztatjuk!

```
1 ős(X, Z) :- szülő(X, Z).
2 ős(X, Z) :- szülő(Y, Z), ős(X, Y).
```

Jó ez így? Miért?

A teljes program

Ay alábbi programban szerepel minden tény és szabály, amiről szó volt. A `%` jel megjegyzések hozzáadására használható, a rendszer szempontjából a `%`-tól jobbra levő szöveg érdektelen, mintha ott sem lenne.

```
1 % szülő(X, Y) => X az Y szülője
2 szülő(abú_tálib, ali).
3 szülő(abdulla, mohamed).
4 szülő(ámna, mohamed).
5 szülő(mohamed, fátima).
```

```
6  szülő(hadídzsa, fátima).
7  szülő(mohamed, zajnab).
8  szülő(hadídzsa, zajnab).
9  szülő(ali, huszajn).
10 szülő(fátima, huszajn).
11 szülő(ali, muhszin).
12 szülő(fátima, muhszin).
13 szülő(ali, haszan).
14 szülő(fátima, haszan).
15 szülő(zajnab, umáma).
16
17 férfi(abú_tálib).
18 férfi(abdulla).
19 férfi(mohamed).
20 férfi(ali).
21 férfi(huszajn).
22 férfi(muhszin).
23 férfi(haszan).
24
25 nő(ámna).
26 nő(hadídzsa).
27 nő(fátima).
28 nő(zajnab).
29 nő(umáma).
30
31 anya(X, Y) :- szülő(X, Y), nő(X).   % X az Y anyja
32 apa(X, Y) :- szülő(X, Y), férfi(X). % X az Y apja
33
34 nagyszülő(X, Z) :- szülő(X, Y), szülő(Y, Z).
```

```
35
36 fivér(X, Y) :-
37     férfi(X),
38     szülő(Z, X), szülő(Z, Y),
39     X \= Y.
40
41 vangyereke(X) :- szülő(X, _).
42
43 ős(X, Z) :- szülő(X, Z).
44 ős(X, Z) :- szülő(X, Y), ős(Y, Z).
```

1.2. Projekt: a négyszín-tétel

Mindössze négy szín elég ahhoz, hogy tetszőleges térképen az országokhoz színeket rendeljünk úgy, hogy minden országhatár két különböző színt válasszon el. (Ez feltételezi, hogy az országok mindig összefüggőek, tehát nincsen másik ország, ami kettéválasztaná őket – ez egy valódi térképnél gyakran nem teljesül.)

Próbáljuk meg kiszínezni Európa egy szeletét!

Jelölje `t` azt, hogy két ország *találkozik*. A találkozáskor a színeknek különbözőeknek kell lennie, tehát felvehetjük a következő tényeket (egyelőre 2 színnel):

```
t(piros, zöld). t(zöld, piros).
```

Ahhoz, hogy a térképünk megfeleljen a követelményeknek, minden határnál teljesülnie kell egy ilyen találkozásnak:

```
térkép(DE, CH, IT, PL, CZ, AT, SI, HR, SK, HU) :-
    t(DE, PL), t(DE, CZ), t(DE, AT), t(DE, CH),
    t(CH, AT), t(CH, IT),
    t(IT, AT), t(IT, SI),
    t(PL, CZ), t(PL, SK),
    t(CZ, AT), t(CZ, SK),
    t(AT, SK), t(AT, HU), t(AT, SI),
    t(SI, HU), t(SI, HR),
    t(HR, HU),
    t(SK, HU).
```

Kitérő: *a négyszín-tétel*

Ez volt az első olyan matematikai tétel, amelynek bizonyítását számítógéppel adták meg, és ezért sok matematikus kezdetben nem is fogadta el bizonyítottnak, és csak „négyszín-sejtésként” hivatkoztak rá. Minden joguk megvolt rá: a több, mint egy hónapon át futó 1976-os eredeti program állítólag tele volt hibákkal. 20 évvel később azonban egy sokkal hatékonyabb megoldást is találtak, és 2004-ben egy tételbizonyító rendszer is belátta, hogy az állítás mindig igaz.

Mivel a színekre vonatkozó szabályt mindkét sorrendben felvettük, az országok sorrendje nem számít, és így elég mindig csak az egyik verziót felírni. Ha most megkérdezzük, hogy

```
?- térkép(DE, CH, IT, PL, CZ, AT, SI, HR, SK, HU).
```

. . . akkor (nem túl meglepő módon) tagadó választ kapunk. Vegyünk hozzá még egy színt!

```
1 t(piros, kék). t(zöld, kék).
2 t(kék, piros). t(kék, zöld).
```

. . . még mindig nem elég. (Ez is elég világos – ha Ausztria pl. piros, akkor a szomszédos országokat körbejárva felváltva kéne zöldet és kéket kapnunk, de páratlan számú ország van körülötte.) Vegyük akkor hozzá a sárgát is!

```
1 t(piros, sárga). t(zöld, sárga). t(kék, sárga).
2 t(sárga, piros). t(sárga, zöld). t(sárga, kék).
```

Ha most tesszük fel a kérdést, akkor már talál megoldást:

```
AT = PL = zöld,
CH = CZ = SI = kék,
DE = HR = IT = SK = piros,
HU = sárga
```

Az egyetlen szépséghibája, hogy Szlovénia kék lett, és így egybefolyhat a tenger kékjével. Több lehetőségünk is van:

– Lecseréljük a kéket egy másik színre

- Addig kérünk további megoldásokat, amíg már egy tengerparti ország sem lesz kék színű
- Felcseréljük a kék és sárga színeket

Tegyük fel, hogy szeretjük a kék színt, és nem akarjuk lecserélni (egyébként is akkor már 5 szín kéne!), a másik két megoldás pedig nem elég automatikus. Mit tehetünk?

Felvehetjük plusz „országként" a tengert, és megkövetelhetjük, hogy mindig kék legyen:

```
térkép(DE, CH, IT, PL, CZ, AT, SI, HR, SK, HU) :-
    Tenger = kék,
    t(DE, PL), t(DE, CZ), t(DE, AT), t(DE, CH),
    t(CH, AT), t(CH, IT),
    t(IT, AT), t(IT, SI),
    t(PL, CZ), t(PL, SK),
    t(CZ, AT), t(CZ, SK),
    t(AT, SK), t(AT, HU), t(AT, SI),
    t(SI, HU), t(SI, HR),
    t(HR, HU),
    t(SK, HU),
    t(DE, Tenger), t(IT, Tenger), t(PL, Tenger),
    t(SI, Tenger), t(HR, Tenger).
```

Így már tökéletes megoldást kapunk. (Az egyenlőségjellel (`=`) két kifejezést tudunk *egyesíteni*, erről majd többet a következő leckében.)

Megtehettük volna azt is, hogy a `Tenger`ek helyett egyszerűen `kék`et írunk, és akkor nincsen szükség a `Tenger = kék` sorra

sem, de egy kicsit kevésbé olvasható lenne a forráskód. Ilyen rövid programoknál még ez nem olyan lényeges, de általában a programozásban fontos arra törekedni, hogy ne csak a számítógép, hanem egy másik ember (és pár hét múlva mi magunk) is megértse, amit írtunk.

2. lecke

Típusok és egyesítés

Az első leckében megismertük a tényeket és szabályokat, és kaptunk egy első képet arról, hogy hogyan is néz ki egy Prolog program. A következőkben azzal fogunk foglalkozni, hogy megvizsgáljuk az alapvető alkotóelemeket, valamint a programok fő mozgató rugóját, az *egyesítést*.

A Prolog *kifejezés*eknek négy típusa van: atom, szám, változó vagy struktúra. Az egyes típusok „helyesírására" más és más szabályok vonatkoznak – nézzük meg ezeket közelebbről!

Atomok

Láttuk, hogy vannak „konkrét dolgok", amelyeket kisbetűvel írunk. Ezeket *atom*oknak szokás nevezni. Az atomok neve háromféleképpen képezhető:

1. Betűk, számok és az alsóvonás (`_`) karakter kombinációja, de az első mindig egy kisbetű. Pl.: `nil`, `foo1`, `bar_42`, `baz__`, `ez_1_hosszú_név`.

2. Különleges karakterek sorozata (ezekből lehet válogatni: `#$&*+-./:<=>?@^~`), pl. `.:.` vagy `==>` vagy `+`.

3. Aposztrófok közti karaktersorozat, pl. `'Peti'` vagy `'Abú Tálib'`.

Az első fajtára már sok példát láttunk; a másodikról majd később lesz szó; a harmadikat pedig jellemzően akkor használjuk, ha nagybetűvel kezdődő vagy szóközt tartalmazó nevet szeretnénk adni (ritka).

Kitérő: *foo–bar–baz*

A programozók előszeretettel használják a `foo`, `bar` és `baz` neveket, amikor valamilyen példát mutatnak, ahol maga a név nem fontos. Hasonlóan, ha egy példában egy egész számot kell választani, ez leggyakrabban a `42` (a *Galaxis útikalauz stopposoknak* nyomán). Ilyen és hasonló programozó-szubkultúrával kapcsolatos érdekességekről rengeteget lehet olvasni a *Zsargon fájl*ban,[†] nagyon szórakoztató olvasmány. (Sajnos csak angolul hozzáférhető, de sokszor nem is igazán lehetne lefordítani.)

[†] http://www.catb.org/jargon/html/

Számok

A Prolog egész és valós számokat tud kezelni, a megszokott jelölésekkel (de angol szokás szerint a tizedesvessző helyett ponttal). A valós számoknál lehet használni az ún. *exponenciális formát* is, ahol egy `e` betű után a 10 kitevője szerepel, tehát pl. ugyanazt a számot jelöli a `1.23e5` és `123000.0`, vagy a `4.56e-3` és `0.00456`.

Egy egész és egy valós szám nem ugyanaz még akkor sem, ha ugyanaz az értékük, tehát

```
?- 1.23e5 = 123000.
```

értéke `false`. Van viszont egy matematikai egyenlőségvizsgálat (`=:=`), és arra már teljesül, hogy

```
?- 1.23e5 =:= 123000.
```

(A számok közti műveletekről majd később lesz még szó.)

Változók

Ahogy láttuk, a „határozatlan dolgok” (*változók*) nagybetűvel vagy alsóvonással kezdődnek, pl. `X`, `_`, `_Y`, `_z`.

Ezek közül a `_` változónév különleges: minden alkalommal egy független változót jelöl. Tegyük fel például, hogy egy sakkprogramban az állás

```
tábla(világos, futó, c, 3).
```

alakú tényekkel van leírva. Ekkor világos összes bábuját lekérdezhetjük úgy, hogy

```
?- tábla(világos, X, _, _).
```

De ugyanez nem működne, ha más azonos változónevet adnánk meg, pl.

```
?- tábla(világos, X, _hely, _hely).
```

A többi változó egy-egy szabályon belül mindig ugyanazt jelöli, de a szabályok közt már azok is függetlenek, pl. az alábbi programban az `X` és `Z` a szabályok bal- és jobboldalán azonos értéket kell, hogy felvegyen, de a két szabálynak nincs hatása egymásra:

```
1 ős(X, Z) :- szülő(X, Z).
2 ős(X, Z) :- szülő(X, Y), ős(Y, Z).
```

Összetett struktúrák

A *struktúra* értelmileg összetartozó dolgokat kapcsol össze. Az alakja olyan, mint a tényeké: egy névvel (atommal) kezdődik, ez a struktúra *funktora*, majd utána zárójelek közt, vesszőkkel elválasztva a hozzátartozó további „dolgok” – ezek a struktúra *paraméterei*, a számuk pedig a funktor *aritása*. A paraméterek maguk is tetszőleges Prolog kifejezések lehetnek, tehát atomok, számok, változók vagy struktúrák.

Egy kétdimenziós pontot leírhatunk egy `pont(X,Y)` struktúrával. Ha ezután egy háromszöget akarunk létrehozni, akkor azt 6 koordináta helyett leírhatjuk 3 ponttal: `háromszög(P1,P2,P3)` – ez egy újabb struktúra. Például

```
?- P1 = pont(1,0), P2 = pont(0,2), P3 = pont(-1,0),
   T = háromszög(P1, P2, P3).
```

esetén a `T` értéke

```
háromszög(pont(1,0),pont(0,2),pont(-1,0))
```

lesz.

Lehetnek azonos névvel különböző aritású funktorok, pl. lehet készíteni egy 3D `pont(X,Y,Z)` funktort is. (Hasonlóan, az eltérő aritású tények/szabályok is megférnek egymás mellett.)

Struktúra vagy tény?

Fontos, hogy ne keverjük össze a tényeket és struktúrákat. Alakra ugyanúgy néznek ki – és később látni fogjuk, hogy ez nem véletlen –, de mást jelent a

```
gyártó(swift, suzuki).
```

tény és az `autó(swift,suzuki)` kifejezés. Az előbbi kifejez egy kapcsolatot a modellnév és a gyártó között, míg az utóbbi a két adatot egy egységbe kapcsolja össze: az autó egy olyan dolog, amihez tartozik egy modellnév és egy gyártó.

8. Feladat. Milyen típusúak (atom, szám, változó, struktúra) az alábbi kifejezések? Vigyázat, van köztük olyan, ami nem helyes Prolog kifejezés!

– `Foo`

- `foo`
- `'Foo'`
- `_foo`
- `'Foo bar baz'`
- `foo(bar, baz)`
- `42`
- `15(X, Y)`
- `+(bal, jobb)`
- `három(Kis(Cica))`

9. Feladat. Milyen struktúrával lehetne jól leírni egy téglalapot? Egy négyzetet? Egy kört?

Egyesítés

Két kifejezés egyesíthető, ha azonosak, vagy ha a bennük levő változókat be lehet úgy állítani, hogy azonossá váljanak. Például a `dátum(Év,Hónap,22)` és a `dátum(X,március,Y)` kifejezések egyesíthetőek, ezt az `=` segítségével tudjuk ellenőrizni:

```
?- dátum(Év,Hónap,22) = dátum(X,március,Y).
Hónap = március,
X = Év,
Y = 22
```

(Mostantól az egyszerűség kedvéért a kérdésekre kapott eredményeket egyszerűen a kérdés alá fogom írni.)

Ugyanakkor pl. a `dátum(Év,Hónap,21)` és `dátum(X,Y,22)` kifejezések nem egyesíthetőek, mivel a harmadik argumentum különböző; a `dátum(X,Y,Z)` és `pont(X,Y,Z)` kifejezések pedig azért nem, mert más a legkülső (*elsődleges*) funktoruk.

Visszatérve az első példára, az egyenlőség végtelen sok más módon is kielégíthető, pl.

```
Év = 1982,
Hónap = március,
X = 1982,
Y = 22
```

... de ezek kevésbé általánosak. Az egyesítés mindig a legáltalánosabbat adja.

A pontos szabályok a következők:

1. Ha S és T *konstansok* (tehát atomok vagy számok), akkor csak abban az esetben egyesíthetőek, ha azonosak.

2. Ha S változó, akkor a két kifejezés egyesíthető, és innentől kezdve S „értéke” T lesz. (Fordítva hasonlóan.)

3. Ha S és T is struktúrák, akkor pontosan akkor egyesíthetőek, ha

   – megegyezik az elsődleges (legkülső) funktoruk

   – ugyanannyi az aritásuk

– minden argumentumuk páronként egyesíthető (az ezekben szereplő változók ebben a rekurzív egyesítésben kaphatnak értéket)

Például a

```
?- háromszög(pont(1,1),A,pont(2,3)) =
   háromszög(X,pont(4,Y),pont(2,Z)).
A = pont(4,Y),
X = pont(1,1),
Z = 3.
```

egyesítés az alábbi lépésekből áll:

1. Az elsődleges funktor (`háromszög`) és az aritás (3) megegyezik.
2. `pont(1,1) = X` (itt az `X` megkapja a `pont(1,1)` értéket)
3. `A = pont(4,Y)` (itt az `A` megkapja a `pont(4,Y)` értéket)
4. `pont(2,3) = pont(2,Z)`
   a) Az elsődleges funktor (`pont`) és az aritás (2) megegyezik.
   b) `2 = 2`
   c) `3 = Z` (itt a `Z` megkapja a `3` értéket)

Egyesítés tényekben

Az egyesítést ki lehet használni egy tényen belül is. Egy szakasz függőlegességét megadhatjuk így:

```
1 függőleges(szakasz(pont(X,_),pont(X,_))).
```

Most feltehetünk mindenféle érdekes kérdést:

```
?- függőleges(szakasz(pont(1,1),pont(1,2))).
true
?- függőleges(szakasz(pont(1,1),pont(2,Y))).
false
?- függőleges(szakasz(pont(1,1),pont(X,2))).
X = 1
?- függőleges(szakasz(pont(X,3),P)).
P = pont(X,_)
```

Az utolsónál a rendszer az érdektelen y-koordinátára valójában egy (implementációtól függő) automatikusan generált nevet fog adni a `_` helyett, pl. `_123`.

10. Feladat. Definiáld szakaszokra a vízszintességet is, majd döntsd el a segítségével, hogy van-e olyan szakasz, ami egyszerre vízszintes és függőleges is!

11. Feladat. Egyesíthetőek-e az alábbi kifejezéspárok? Ha igen, milyen értéke lesz a változóknak?

- `pont(A,B) = pont(1,2)`
- `pont(A,B) = pont(X,Y,Z)`
- `plusz(2,2) = 4`
- `2+D = E+2`
- `f(g(X),Y) = f(Y,h(X))`

- `háromszög(pont(-1,0),P2,P3) = háromszög(P1,pont(1,0),(pont(0,Y))`

12. Feladat. Készíts egy szabályt, ami eldönti, hogy egy téglalap oldalai a tengelyekkel párhuzamosak-e!

2.1. Kétféle olvasat

Egy P `:-` Q`,` R`.` alakú szabályt kétféleképpen lehet értelmezni:

1. Leíró (*deklaratív*) olvasatok:
   - P igaz akkor, ha Q és R igaz.
   - Q és R-ből következik P.
2. Működés szerinti (*procedurális*) olvasatok:
   - Ahhoz, hogy megoldjuk P-t, *először* megoldjuk Q-t, és *aztán* megoldjuk R-et.
   - Ahhoz, hogy kielégítsük P-t, *először* kielégítjük Q-t és *aztán* kielégítjük R-et.

A legfontosabb különbség az, hogy a procedurális olvasatokban számít a kifejezések sorrendje.

Tehát például az

```
1 ős(X, Z) :- szülő(X, Y), ős(Y, Z).
```

szabályt kiolvashatjuk úgy, hogy

- `X` őse `Z`-nek, ha `X` szülője `Y`-nak és `Y` őse `Z`-nek.
- Ha `X` szülője `Y`-nak és `Y` őse `Z`-nek, akkor `X` őse `Z`-nek.
- Ahhoz, bebizonyítsuk, hogy `X` őse `Z`-nek, először azt kell belátni, hogy `X` szülője `Y`-nak, majd pedig azt, hogy `Y` őse `Z`-nek.
- Ahhoz, hogy találjunk egy `Z`-t, amelynek `X` az őse, először találjunk egy `Y`-t, amelynek `X` a szülője, majd egy `Z`-t, amelynek `Y` az őse.

Logikai vagy

A deklaratív olvasat pusztán logika. A kifejezéseket eddig mindig vesszővel (`,`) kapcsoltuk össze, ami logikai *és*t jelent. A logikai (megengedő) *vagy*ot a pontosvessző (`;`) jelöli. A vesszőnek van elsőbbsége, tehát a P`:-` Q`,` R`;` S`,` T`.` kifejezést úgy értelmezzük, mintha P`:-` `(`Q`,` R`)` `;` `(`S`,` T`).` lenne. (Az elsőbbségről még később lesz szó bővebben.)

A pontosvessző helyett mindig írhatunk külön szabályokat, és fordítva, pl. az

```
1 ős(X, Z) :- szülő(X, Z).
2 ős(X, Z) :- szülő(X, Y), ős(Y, Z).
```

szabályt írhattuk volna egy sorban is:

```
1 ős(X, Z) :- szülő(X, Z); szülő(X, Y), ős(Y, Z).
```

A külön szabályokba írt változat általában jobban olvasható.

2.2. Nyomkövetés

Ahhoz, hogy jobban megértsük a procedurális olvasatot, kövessük végig, hogy mit csinál a rendszer egy egyszerű feladat megoldásakor! Legyen a program a következő:

```
1 nagy(medve).
2 nagy(elefánt).
3 kicsi(macska).
4 barna(medve).
5 fekete(macska).
6 szürke(elefánt).
7 sötét(Z) :- fekete(Z).
8 sötét(Z) :- barna(Z).
```

A kérdés pedig:

```
?- sötét(X), nagy(X).
```

„Mi az ami sötét színű és nagy?” A megoldáshoz vezető lépések:

1. A teljesítendő célok listája `sötét(X), nagy(X)`.
2. Végigmegyünk a programon az elejétől kezdve, hogy találunk-e a `sötét(X)`-el egyesíthető szabály-fejet (vagy tényt, hiszen a tények tulajdonképpen szabályok, amelyeknek a törzse `true`). A 7-es sor az első ilyen. Az egyesítés miatt `Z = X`, a `sötét(X)`-et helyettesítjük a szabály törzsével, az új cél-lista `fekete(X), nagy(X)`.
3. Végigmegyünk a programon az elejétől kezdve, hogy találunk-e a `fekete(X)`-el egyesíthető szabály-fejet. Az 5-ös

sor az első ilyen. Az egyesítés miatt `X = macska`, és mivel az 5-ös sornak nincsen törzse, az új cél-lista `nagy(macska)`.

4. Most a `nagy(macska)`-val egyesíthető szabályt keresünk, de ilyen nincsen.
   - Visszalépünk egyet, és az `X` változót ismét szabaddá tesszük. Keresünk egy újabb egyezést a `fekete(X)`-el, onnan, ahol legutóbb abbahagytuk (5-ös sor után), de ilyen sincsen.
   - Visszalépünk még egyet, és újabb egyezést keresünk a `sötét(X)`-el, onnan kezdve, ahol legutóbb abbahagytuk (7-es sor után). Az első ilyen a 8. sorban van. Az egyesítés miatt `Z = X`, a `sötét(X)`-et helyettesítjük a törzzsel, az új cél-lista tehát `barna(X), nagy(X)`.
5. A program elejétől keresünk a `barna(X)`-el egyesíthető szabályt. A 4-es sor az első ilyen. Az egyesítés miatt `X = medve`, és nincs törzs, tehát az új cél-lista `nagy(medve)`.
6. A program elejétől keresünk a `nagy(medve)`-vel egyesíthető szabályt. Rögtön az első sorban meg is találjuk; nincs törzse, és így a cél-listánk elfogyott, készen vagyunk! Az `X` értéke tehát `medve`.

Bár nem része a szabványnak, a legtöbb Prolog rendszer lehetővé teszi a program végigkövetését a `trace` és `notrace` parancsok segítségével:

```
?- trace, sötét(X), nagy(X), notrace.
```

Ez minden egyes lépést kiír; ezek a következő típusúak lehetnek:

Call	Egy új cél keresése indul
Redo	Újra próbálkozik egy másik egyesítéssel
Fail	A cél kielégítése sikertelen
Exit	A cél kielégítése sikeres

A futtatás eredménye:

```
?- trace, sötét(X), nagy(X), notrace.
Call: sötét(_106)
  Call: fekete(_106)
  Exit: fekete(macska)
Exit: sötét(macska)
Call: nagy(macska)
Fail: nagy(macska)
Redo: sötét(_106)
  Call: barna(_106)
  Exit: barna(medve)
Exit: sötét(medve)
Call: nagy(medve)
Exit: nagy(medve)
X = medve.
```

13. Feladat. Írd újra az alábbi programot pontosvessző nélkül!

```
1 kiolvas(Szám, Szó) :-
2     Szám = 1, Szó = egy;
3     Szám = 2, Szó = kettő;
4     Szám = 3, Szó = három.
```

14. Feladat. Mit ad az

```
1 f(1, egy).
2 f(s(1), kettő).
3 f(s(s(1)), három).
4 f(s(s(s(X))), N) :- f(X, N).
```

program az alábbi kérdésekre?

```
?- f(s(1), A).
?- f(s(s(1)), kettő).
?- f(s(s(s(s(s(s(1)))))), C).
?- f(D, három).
```

15. Feladat. Vezesd végig a

```
?- nagy(X), sötét(X)
```

kérdés megoldását!

16. Feladat. A `trace`-t be lehet tenni egy szabály törzsébe is, és akkor onnantól kapcsolódik be a nyomkövetés. Írd át a 7. sort az alábbira:

```
1 sötét(Z) :- trace, fekete(Z).
```

Ezután nézd meg a

```
?- sötét(X), nagy(X).
```

kérdést! Amikor belép a nyomkövetésbe, hagyd továbbmenni – mi történik? Mi a helyzet akkor, ha a 8. sorhoz hozzáadod a `notrace`-t, és újra kipróbálod?

A sorrend fontossága

Ha nem vigyázunk, könnyen írhatunk végtelen rekurziót. Ez a legtisztább formájában úgy néz ki, hogy

```
1 p :- p.
```

Ha most kiértékeljük a

```
?- p.
```

kérdést, akkor a gép csak dolgozik és dolgozik, és nem áll le. (Implementációtól függően ilyenkor vagy van egy „Abort” gomb, vagy a ctrl + C billentyűkombináció megnyomásával lehet a programot leállítani.) Bonyolultabb esetekben előfordulhat, hogy egy idő után valami olyan hibaüzenetet kapunk, hogy „stack limit exceeded”, ami lényegében azt jelenti, hogy a rekurzió túl mély lett.

Az érdekes az, hogy egy program, amely deklaratív olvasatban helyes, vezethet végtelen rekurzióra (tehát a procedurális olvasat szerint lehet hibás). Nézzük meg megint az `ős` szabályt!

```
1 ős(X, Z) :- szülő(X, Z).
2 ős(X, Z) :- szülő(X, Y), ős(Y, Z).
```

Itt két sorrendről beszélhetünk: a szabályok sorrendjéről, és a szabályon belüli kifejezések sorrendjéről. Vizsgáljuk meg az összes verziót!

```
1 % A múltkori családfa egy része
2 szülő(ámna, mohamed).
3 szülő(abdulla, mohamed).
```

```
4  szülő(mohamed, zajnab).
5  szülő(mohamed, fátima).
6  szülő(fátima, huszajn).
7
8  % Eredeti
9  ős1(X, Z) :- szülő(X, Z).
10 ős1(X, Z) :- szülő(X, Y), ős1(Y, Z).
11
12 % Szabályok felcserélve
13 ős2(X, Z) :- szülő(X, Y), ős2(Y, Z).
14 ős2(X, Z) :- szülő(X, Z).
15
16 % Kifejezések felcserélve
17 ős3(X, Z) :- szülő(X, Z).
18 ős3(X, Z) :- ős3(Y, Z), szülő(X, Y).
19
20 % Mindkettő felcserélve
21 ős4(X, Z) :- ős4(Y, Z), szülő(X, Y).
22 ős4(X, Z) :- szülő(X, Z).
```

A deklaratív olvasat a cserék során lényegesen nem változik, az mindig jó lesz. Mi a helyzet a procedurális olvasattal?

Az `ős1` a már ismert verzió:

```
?- trace, ős1(abdulla, fátima).
Call: ős1(abdulla, fátima)
  Call: szülő(abdulla, fátima)
  Fail: szülő(abdulla, fátima)
Redo: ős1(abdulla, fátima)
  Call: szülő(abdulla, Y)
```

```
  Exit: szülő(abdulla, mohamed)
  Call: ős1(mohamed, fátima)
    Call: szülő(mohamed, fátima)
    Exit: szülő(mohamed, fátima)
  Exit: ős1(mohamed, fátima)
Exit: ős1(abdulla, fátima)
true
```

Az `ős2` esetében először mindig nem-szülő őst próbál keresni:

```
?- trace, ős2(abdulla, fátima).
Call: ős2(abdulla, fátima)
  Call: szülő(abdulla, Y1)
  Exit: szülő(abdulla, mohamed)
  Call: ős2(mohamed, fátima)
    Call: szülő(mohamed, Y2)
    Exit: szülő(mohamed, zajnab)
    Call: ős2(zajnab, fátima)
      Call: szülő(zajnab, Y3)
      Fail: szülő(zajnab, Y3)
    Redo: ős2(zajnab, fátima)
      Call: szülő(zajnab, fátima)
      Fail: szülő(zajnab, fátima)
    Fail: ős2(zajnab, fátima)
    Redo: szülő(mohamed, Y2)
    Exit: szülő(mohamed, fátima)
    Call: ős2(fátima, fátima)
      Call: szülő(fátima, Y3)
      Exit: szülő(fátima, huszajn)
      Call: ős2(huszajn, fátima)
```

```
        Call: szülő(huszajn, Y4)
        Fail: szülő(huszajn, Y4)
      Redo: ős2(huszajn, fátima)
        Call: szülő(huszajn, fátima)
        Fail: szülő(huszajn, fátima)
      Fail: ős2(huszajn, fátima)
    Redo: ős2(fátima, fátima)
      Call: szülő(fátima, fátima)
      Fail: szülő(fátima, fátima)
    Fail: ős2(fátima, fátima)
  Redo: ős2(mohamed, fátima)
    Call: szülő(mohamed, fátima)
    Exit: szülő(mohamed, fátima)
  Exit: ős2(mohamed, fátima)
Exit: ős2(abdulla, fátima)
true
```

Az `ős3` a rekurzív szabálynál először az ősséget ellenőrzi, csak aztán a szülőséget:

```
?- trace, ős3(abdulla, fátima).
Call: ős3(abdulla, fátima)
  Call: szülő(abdulla, fátima)
  Fail: szülő(abdulla, fátima)
Redo: ős3(abdulla, fátima)
  Call: ős3(X, fátima)
    Call: szülő(X, fátima)
    Exit: szülő(mohamed, fátima)
  Exit: ős3(mohamed, fátima)
  Call: szülő(abdulla, mohamed)
```

```
  Exit: szülő(abdulla, mohamed)
Exit: ős3(abdulla, fátima)
true
```

Az `ős4`-nél végtelen rekurzió jön létre:

```
?- trace, ős4(abdulla, fátima).
Call: ős4(abdulla, fátima)
  Call: ős4(X1, fátima)
    Call: ős4(X2, fátima)
      Call: ős4(X3, fátima)
        Call: ős4(X4, fátima)
          ...
```

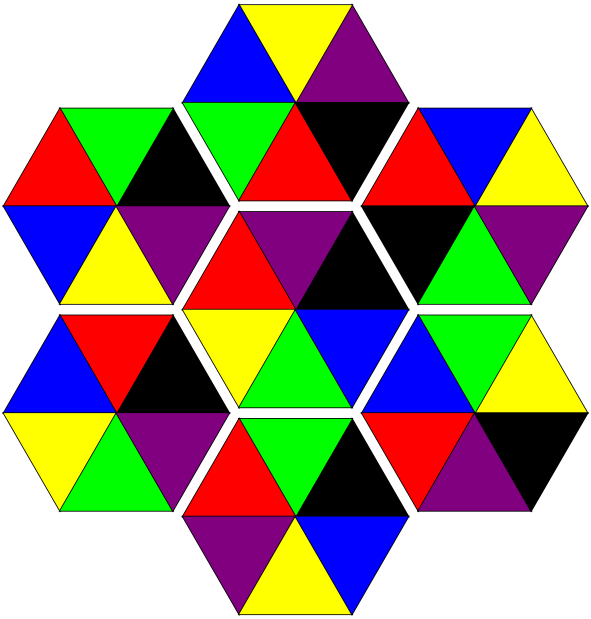
Egy jó ökölszabály, hogy érdemes az egyszerűbb szabályokat előre tenni (`ős1` és `ős3`), és a szabályokon belül is az egyszerűbb kifejezések jöjjenek előbb (`ős1`).

17. Feladat. Az `ős3` esetében, ha további megoldást kérünk (ami itt az „Abdulla őse Fátimának” állítás egy másik bizonyítását jelentené), akkor megint végtelen rekurzióba kerül a program. Miért? Nézd meg nyomkövetéssel!

2.3. Projekt: színes hatszögek

Most már készen állunk arra, hogy egy kicsit komolyabb feladatot is megnézzünk.

Az alábbi játékban az a cél, hogy a 7 színes hatszöget úgy helyezzük el a képen látható alakzatban, hogy mindig csak azonos színek találkozzanak:



Ahhoz, hogy megoldjuk ezt a feladatot, először valahogy le kell írnunk az ismert tényeket – tehát itt azt, hogy milyen lapok léteznek. A lapok színeit egy `l` struktúrába fogjuk össze, és a következő tényeket kapjuk:

```
1 lap(l(fekete,lila,sárga,kék,zöld,piros)).
2 lap(l(fekete,zöld,piros,kék,sárga,lila)).
3 lap(l(fekete,zöld,lila,sárga,kék,piros)).
4 lap(l(fekete,lila,piros,sárga,zöld,kék)).
```

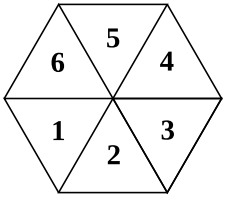
```
lap(l(fekete,piros,kék,sárga,zöld,lila)).
lap(l(fekete,sárga,zöld,kék,piros,lila)).
lap(l(fekete,zöld,piros,lila,sárga,kék)).
```

Így például végig tudunk menni a lapokon a

```
?- lap(L).
```

kérdés segítségével.

A lapok tetszőlegesen elforgathatóak, tehát a forgatásukra is kell valami mód. Ehhez először egyezzünk meg abban, hogy pontosan milyen elhelyezést jelent a színeknek egy sorrendje:



Tehát az első szín van délnyugatra, a második délre, ... a hatodik északnyugatra. Egy forgatás során annyi történik, hogy valamelyik másik szín kerül előre, de utána a sorrend változatlan. Ezt így írhatjuk le:

```
forgat(l(A,B,C,D,E,F), l(A,B,C,D,E,F)).
forgat(l(A,B,C,D,E,F), l(B,C,D,E,F,A)).
forgat(l(A,B,C,D,E,F), l(C,D,E,F,A,B)).
forgat(l(A,B,C,D,E,F), l(D,E,F,A,B,C)).
forgat(l(A,B,C,D,E,F), l(E,F,A,B,C,D)).
forgat(l(A,B,C,D,E,F), l(F,A,B,C,D,E)).
```

A következő feladat, hogy arra adjunk egy szabályt, hogy mikor kapcsolódik helyesen két lap. Ehhez azt is kell tudni, hogy egymáshoz képest hogyan helyezkednek el. A `kapcsolódik(L1, L2, Irány, F1, F2)` azt mondja, hogy ha az `L1` laptól `Irány` irányba helyezzük le az `L2` lapot, akkor a két lap elforgatható egy `F1` illetve `F2` helyzetbe úgy, hogy a találkozásuknál a színek megegyeznek. Itt az irány leírására használhatnánk a fent bevezetett számokat, de jobban olvasható talán, ha égtájakat használunk. Nézzük meg először a `dny` (délnyugat) irányt!

```
kapcsolódik(L1, L2, dny, F1, F2) :-
    forgat(L1, F1), forgat(L2, F2),
    F1 = l(X,_,_,_,_,_),
    F2 = l(_,_,_,X,_,_).
```

A törzs első sora csak annyit mond, hogy az `F1` és `F2` az `L1` és `L2` elforgatottja; a második és harmadik sor pedig azt biztosítja, hogy az elforgatott lapokon a *megfelelő* helyen levő szín megegyezik (a többi nem számít). Mivel az `L1`-től délnyugatra van az `L2`, ezért az `L1` délnyugati (első) színe az érdekes, az `L2`-nek pedig a szemben levő, tehát északkeleti (negyedik) színe.

Teljesen hasonlóan felírhatjuk a többi égtájra is a kapcsolódási szabályokat:

```
kapcsolódik(L1, L2, d, F1, F2) :-
    forgat(L1, F1), forgat(L2, F2),
    F1 = l(_,X,_,_,_,_),
    F2 = l(_,_,_,_,X,_).
kapcsolódik(L1, L2, dk, F1, F2) :-
    forgat(L1, F1), forgat(L2, F2),
```

```
    F1 = l(_,_,X,_,_,_),
    F2 = l(_,_,_,_,_,X).
kapcsolódik(L1, L2, ék, F1, F2) :-
    forgat(L1, F1), forgat(L2, F2),
    F1 = l(_,_,_,X,_,_),
    F2 = l(X,_,_,_,_,_).
kapcsolódik(L1, L2, é, F1, F2) :-
    forgat(L1, F1), forgat(L2, F2),
    F1 = l(_,_,_,_,X,_),
    F2 = l(_,X,_,_,_,_).
kapcsolódik(L1, L2, ény, F1, F2) :-
    forgat(L1, F1), forgat(L2, F2),
    F1 = l(_,_,_,_,_,X),
    F2 = l(_,_,X,_,_,_).
```

Most már minden megvan ahhoz, hogy megkeressük a megoldást. Ezt úgy találjuk meg, hogy veszünk 7 különböző (!) lapot, és felírjuk a rájuk vonatkozó kapcsolódási feltételeket:

```
megoldás(F1, F2, F3, F4, F5, F6, F7) :-
    lap(L1),
    lap(L2), L2 \= L1,
    lap(L3), L3 \= L1, L3 \= L2,
    lap(L4), L4 \= L1, L4 \= L2, L4 \= L3,
    lap(L5), L5 \= L1, L5 \= L2, L5 \= L3, L5 \= L4,
    lap(L6), L6 \= L1, L6 \= L2, L6 \= L3, L6 \= L4,
             L6 \= L5,
    lap(L7), L7 \= L1, L7 \= L2, L7 \= L3, L7 \= L4,
             L7 \= L5, L7 \= L6,
    kapcsolódik(L1, L2, dk,  F1, F2),
```

```
12      kapcsolódik(F1, L7, ék,  F1, F7),
13      kapcsolódik(F2, L3, ék,  F2, F3),
14      kapcsolódik(F2, F7, é,   F2, F7),
15      kapcsolódik(F3, L4, é,   F3, F4),
16      kapcsolódik(F3, F7, ény, F3, F7),
17      kapcsolódik(F4, L5, ény, F4, F5),
18      kapcsolódik(F4, F7, dny, F4, F7),
19      kapcsolódik(F5, L6, dny, F5, F6),
20      kapcsolódik(F5, F7, d,   F5, F7),
21      kapcsolódik(F6, F1, d,   F6, F1),
22      kapcsolódik(F6, F7, dk,  F6, F7).
```

Itt a lapok számozása szintén a fenti módon történik, a 7-es számú a középső lap. Az első hét sor csak felveszi a lapokat, és biztosítja, hogy mindegyik különböző legyen. Utána jönnek a szabályok – itt egy dologra kell figyelni, hogy (a procedurális olvasat értelmében) a `kapcsolódik` szabályok kielégítése sorrendben történik, ezért amint egy lapot „letettünk”, annak meghatározódik a forgatása, tehát onnantól kezdve a forgatás nélküli `L` helyett az elforgatott `F`-et kell használni. Ezzel megköveteljük, hogy a `kapcsolódik`ban a „sima” és „elforgatott” változat megegyezzen, tehát ne tudja tovább forgatni a lapot.

Keressük meg akkor a megoldást!

```
?- megoldás(F1, F2, F3, F4, F5, F6, F7).
F1 = l(kék, zöld, piros, fekete, lila, sárga),
F2 = l(kék, sárga, lila, fekete, zöld, piros),
F3 = l(fekete, piros, kék, sárga, zöld, lila),
F4 = l(sárga, zöld, kék, piros, lila, fekete),
F5 = l(sárga, kék, fekete, zöld, piros, lila),
```

```
F6 = l(fekete, lila, piros, sárga, zöld, kék),
F7 = l(fekete, zöld, lila, sárga, kék, piros)
```

Ebben a programban rengeteg az ismétlés, ami nem túl szép, de a jelenlegi eszközeinkből ennyire futja. Nemsokára megismerkedünk a listákkal és a számolással, amelyek segítségével a `forgat` és a `kapcsolódik` szabályokat egy–egy sorban meg lehetne oldani, és a *megoldás*ban a lapok különbözőségét is könnyebben lehetne biztosítani.

3. lecke

Listák

Ahogy a múltkor láttuk, egy Prolog kifejezés vagy egyszerű (konstans vagy változó), vagy egy fix aritású összetett struktúra. Gyakran előfordul azonban, hogy nem tudjuk előre, hogy hány adattal akarunk foglalkozni. Hogyan tudnánk dolgoknak egy listáját egy struktúrával kifejezni? Természetesen rekurzívan! Kezdjük először csak két dologgal:

```
lista(a, b)
```

Ha hármat akarunk beletenni egy listába, akkor megtehetnénk, hogy eggyel több argumentumot veszünk bele:

```
lista(a, b, c)
```

...de ez több okból sem igazán jó megoldás. Egyrészt ez így egy másik funktor lesz (hiszen más az aritása), másrészt ezzel

még mindig csak *ismert* hosszúságú listákat tudnánk készíteni. Helyette csinálhatjuk viszont a következőt:

```
lista(a, lista(b, c))
```

Ezt tetszőlegesen lehet folytatni, pl.

```
lista(a, lista(b, lista(c, lista(d, e))))
```

Ez egyrészt azért jó, mert így mindig egy 2-aritású `lista` funktorunk van, másrészt ezzel tudunk olyat írni, hogy

```
lista(a, Maradék)
```

ami egy tetszőleges listát ír le, melynek az első eleme `a`, vagy

```
lista(a, lista(b, Maradék))
```

ami egy olyan lista, amelynek az első két eleme `a` és `b`.

Ennek így egy szépséghibája van: az utolsó elem kezelése kicsit más, mint a többié, és emiatt pl. az előző példákban a „tetszőleges" lista mégsem teljesen tetszőleges, mert nem lehet egy- ill. kételemű. Ezt azzal tudjuk megoldani, hogy bevezetünk egy olyan konstanst, amivel a lista végét jelöljük. Pl. az `a`, `b` és `c` elemeket tartalmazó lista ekkor így néz ki:

```
lista(a, lista(b, lista(c, vége)))
```

A szokásos Prolog jelölés a `lista` helyett a pont (`.`) funktor, a `vége` helyett pedig a szögletes zárójelpár (`[]`) atom.[†] Az előző listát tehát erre átírva ilyet kapunk:

[†]Az SWI-Prolog a pontot lecserélte a kicsit furcsán kinéző `'[|]'` funktorra.

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

Ez sajnos nem a legkönnyebben olvasható. Szerencsére van rá egy egyszerűsített jelölés:

```
?- .(X, Y) = [X|Y].
true
```

Sőt, nem csak

```
?- .(a, .(b, .(c, []))) = [a | [b | [c | []]]].
true
```

teljesül, hanem az egymásba ágyazást vesszővel lehet helyettesíteni:

```
?- [a | [b | [c | []]]] = [a, b, c | []].
true
```

És végül van még egy utolsó „egyszerűsítés” (a szakszó erre a *szintaktikus cukor*): ha a függőleges vonal jobboldalán a `[]` van, akkor ezt el lehet hagyni:

```
?- [a, b, c | []] = [a, b, c].
true
```

Ez tehát a Prolog *láncolt lista* struktúrája; a `[]` neve *üres lista*, és egy `[X|Y]` listában az `X` a lista *eleje*, az `Y` pedig a lista *maradéka*. A maradék mindig vagy egy lista, vagy az üres lista atom.

A gyakorlatban mindig ezt az egyszerű formát használjuk, de fontos érteni, hogy ez valójában egymásba ágyazott funktorokból áll, és ezért pl. `[a, b, c] = [a | [b, c]] = [a, b | [c]] = [a, b, c | []]`.

3.1. Műveletek listákon

Az alábbiakban nézzünk meg néhány hasznos műveletet, amelyeket listákra alkalmazhatunk.

Tartalmazás

Az egyik legfontosabb kérdés, amit listákkal kapcsolatban feltehetünk, az az, hogy valami benne van-e:

```
1 tartalmaz(X, [X|_]).
2 tartalmaz(X, [_|Maradék]) :- tartalmaz(X, Maradék).
```

Szavakban megfogalmazva, egy lista akkor tartalmaz valamit, (i) ha az az eleje, vagy (ii) ha a maradéka tartalmazza azt. Ezzel

```
?- tartalmaz(b, [a,b,c]).
true
?- tartalmaz(b, [a,[b,c]]).
false
?- tartalmaz([b,c], [a,[b,c]]).
true
```

Emellett a

```
?- tartalmaz(X, [a,b,c]).
```

kérdésre megkapjuk az `X = a`, `X = b` és `X = c` megoldásokat, sőt, meg is fordíthatjuk, és feltehetjük a kérdést, hogy „Milyen listák tartalmazzák az `a`-t?”

```
?- tartalmaz(a, L).
```

Erre a következő (jellegű) megoldásokat kapjuk:

```
L = [a | Maradék] ;
L = [X, a | Maradék] ;
L = [X1, X2, a | Maradék] ;
L = [X1, X2, X3, a | Maradék] ;
...
```

Az első egy tetszőleges lista, amelynek az első eleme `a`; a második egy olyan, amelynek a második eleme `a` és így tovább.

Feltehetünk összetett kérdéseket is, pl. milyen háromelemű listák vannak, amelyek tartalmazzák az `a`, `b` és `c` elemeket?

```
?- L = [_, _, _], tartalmaz(a, L),
   tartalmaz(b, L), tartalmaz(c, L).
L = [a, b, c] ;
L = [a, c, b] ;
L = [b, a, c] ;
L = [c, a, b] ;
L = [b, c, a] ;
L = [c, b, a]
```

Összefűzés

Két listát össze is tudunk csatolni. Legyen `hozzáfűz(L1, L2, L3)` igaz akkor, ha `L1` és `L2` egymás után rakva `L3`-at adja, pl.

```
?- hozzáfűz([a,b,c], [d,e], [a,b,c,d,e]).
true
```

Ha `L1` üres, akkor `L2` és `L3` megegyezik:

```
hozzáfűz([], L2, L2).
```

Egyébként `L1` első eleme az `L3` első eleme lesz; az `L3` maradéka pedig az `L1` maradékának és az `L2`-nek az összefűzéséből adódik:

```
hozzáfűz([X|M1], L2, [X|M3]) :- hozzáfűz(M1, L2, M3).
```

Annak ellenére, hogy onnan indultunk, hogy hogyan lehet két listát összefűzni, a kérdés ismét megfordítható, pl. „Hogyan lehet egy listát két listára osztani?”

```
?- hozzáfűz(L1, L2, [a,b,c]).
L1 = [],
L2 = [a, b, c] ;
L1 = [a],
L2 = [b, c] ;
L1 = [a, b],
L2 = [c] ;
L1 = [a, b, c],
L2 = []
```

Vagy: „Igaz-e, hogy az `[a,b]` listával kezdődik az `[a,b,c]` lista?”

```
?- hozzáfűz([a,b], _, [a,b,c]).
true
```

Megkereshetjük vele egy listában az előző és következő elemet:

```
?- hozzáfűz(_, [Előző, már, Következő | _],
   [jan, feb, már, ápr, máj, jún,
    júl, aug, szep, okt, nov, dec]).
Előző = feb,
Következő = ápr
```

A `tartalmaz` szabályt is leírhatjuk a segítségével:

```
1 tartalmaz(X, L) :- hozzáfűz(_, [X|_], L).
```

Szavakban: egy `L` lista akkor tartalmaz egy `X` elemet, ha szétválasztható két listára, amelyből a másodiknak az első eleme `X`. (Az első lista ill. a második maradéka is lehet üres.)

Hozzáadás és törlés

Új elemet egy lista elejéhez olyan könnyű hozzáadni, hogy erre nem is szokás külön szabályt írni, de ha akarunk, itt van:

```
1 hozzáad(X, L, [X|L]).
```

A listák végére (hatékonyan!) beszúrni kicsit bonyolultabb, majd később lesz róla szó.

Hogyan kell törölni egy elem egy előfordulását egy listából?

```
1 töröl(X, [X|M], M).
2 töröl(X, [Y|M], [Y|M1]) :- töröl(X, M, M1).
```

Tehát: ha `X` a lista első eleme, akkor a törlés után a lista maradékát kapjuk. Egyébként a lista első eleme és az eredmény első eleme megegyezik, és az eredmény maradéka pedig ugyanaz, mint az eredeti lista maradéka, amiből kitöröltük az `X`-et.

Egy példa a használatára:

```
?- töröl(a, [a,b,a,a], L).
L = [b, a, a] ;
L = [a, b, a] ;
L = [a, b, a]
```

Itt a második és harmadik megoldás azonosnak tűnik, de valójában az egyik az eredeti listából az `a` elem második, a másik pedig a harmadik előfordulását törölte.

Mi történik, ha az „eredeti” listát vesszük ismeretlennek?

```
?- töröl(a, L, [1,2,3]).
L = [a, 1, 2, 3] ;
L = [1, a, 2, 3] ;
L = [1, 2, a, 3] ;
L = [1, 2, 3, a]
```

Ahogy látszik, a „Mi az a lista, amiből ha kitörlünk egy `a`-t, akkor `[1,2,3]`-at kapunk?” kérdés egyszerűbben úgy fogalmazható meg, hogy „Mit kapunk, ha az `[1, 2, 3]` listába beleteszünk egy `a`-t?”

Ez annyira hasznos, hogy adhatunk neki egy új nevet:

```
1 betesz(X, L, L1) :- töröl(X, L1, L).
```

A törlés használható kiválasztásra is:

```
?- töröl(X, [a,b,c], _).
X = a ;
X = b ;
X = c
```

Ha a törölni kívánt elem nem szerepel a listában, az eredmény `false` lesz. Ezt kihasználva ezzel is definiálhatjuk a `tartalmaz` szabályt:

```
tartalmaz(X, L) :- töröl(X, L, _).
```

Részlisták

Következőnek nézzük meg, hogy mikor része egy lista egy másiknak:

```
?- részlista([c,d,e], [a,b,c,d,e,f]).
true
?- részlista([c,e], [a,b,c,d,e,f]).
false
```

Ahogy a második példából látszik, itt a „részét” nem úgy értelmezzük, hogy az első lista minden elemét tartalmazza a második (mint egy halmaznál), hanem hogy pontosan ugyanolyan sorrendben, más elemek közbeékelődése nélkül szerepelnek.

Ez a szabály könnyen megadható a `hozzáfűz` segítségével:

```
részlista(R, L) :-
    hozzáfűz(_, L1, L), hozzáfűz(R, _, L1).
```

Tehát `R` akkor része `L`-nek, ha valamilyen listát elé- és utánafűzve megkapjuk `L`-et. A definíció ezt két részletben írja le: az első tag azt mondja, hogy az `L` az `L1` listára végződik; a második pedig azt, hogy ez az `L1` lista az `R`-el kezdődik. A vég kezdete pedig épp azt jelenti, hogy `R` valahol belül van `L`-ben (vagy valamelyik szélén, ha az itt alsóvonással jelölt listák egyike az üres lista).

Szokás szerint nézzük meg, mit kapunk a „fordított” felhasználásban:

```
?- részlista(R, [a,b,c]).
R = [] ;
R = [a] ;
R = [a, b] ;
R = [a, b, c] ;
R = [] ;
R = [b] ;
R = [b, c] ;
R = [] ;
R = [c] ;
R = []
```

Ahogy várható volt, ezzel megkapjuk az `[a,b,c]` lista összes részlistáját – az üreset többször is. (Miért? Tipp: nézd meg a `hozzáfűz(X, _, [a,b,c])` kimenetét).

Permutációk

Két lista akkor *permutációja* vagy átrendezése egymásnak, ha ugyanazokat az elemeket tartalmazzák. Az üres listának csak az üres lista a permutációja:

```
1 permutáció([], []).
```

Ha az első lista nem üres, akkor visszavezethetjük a feladatot az eggyel rövidebb listákra:

```
1 permutáció([X|M], P) :-
2     permutáció(M, L), betesz(X, L, P).
```

Tehát úgy kapjuk az első lista permutációját, hogy vesszük a maradék (`M`) egy permutációját (`L`), és ebbe betesszük az első elemet (`X`).

Egy másik logika az lehet, hogy kiválasztunk (kitörlünk) egy elemet, a maradéknak vesszük egy permutációját, és az elejére beszúrjuk a kitörölt elemet:

```
1 permutáció(L, [X|P]) :-
2     töröl(X, L, M), permutáció(M, P).
```

Ellenőrizzük, hogy működik-e! A

```
?- permutáció([piros, zöld, kék], P).
```

kérdésre mindkettő visszaadja mind a 6 jó megoldást (bár különböző sorrendben) és közli, hogy nincs több. Viszont a fordított

```
?- permutáció(P, [piros, zöld, kék]).
```

esetben az első verzió a 6 megoldás után végtelen rekurzióba kerül, a másodiknál pedig már az első után beragad. Meg lehet oldani, hogy mindig jó legyen, csak kicsit ki kell egészíteni, de mivel szimmetrikus, nincs rá igazán szükség.

18. Feladat. Írj egy szabályt, amivel ki lehet venni egy listából az utolsó 3 elemet!

```
?- kivesz3([a,b,c,d,e], [a,b]).
true
```

19. Feladat. Írj egy szabályt, amivel ki lehet venni egy listából az első és utolsó 3 elemet!

```
?- kivesz33([a,b,c,d,e,f,g], [d]).
true
```

20. Feladat. Írj egy szabályt, amivel megkaphatjuk egy lista utolsó elemét! Készítsetek két verziót, egyet `hozzáfűzz`el, egyet anélkül!

```
?- utolsó([a,b,c], c).
true
```

21. Feladat. Írd meg a `páros_hosszú(L)` és `páratlan_hosszú(L)` szabályokat! Ezek akkor igazak, amikor az `L` lista páros- ill. páratlan számú elemből áll (nincs szükség számokra hozzá).

22. Feladat. Írj egy szabályt, ami megállapítja, hogy két lista megfordítottja-e egymásnak!

```
?- fordított([a,b,c], X).
X = [c,b,a]
```

23. Feladat. Írj egy szabályt, ami megállapítja, hogy egy szó *palindróma*-e, azaz visszafele is ugyanaz-e!

```
?- palindróma([g,ö,r,ö,g]).
true
```

24. Feladat. Írj egy szabályt, amivel egy listát eggyel „elforgathatunk” úgy, hogy az első lista első eleme a második végére kerül!

```
?- forgat([a, b, c, d], [b, c, d, a]).
true
```

25. Feladat. Írj egy szabályt, ami a részhalmazságot vizsgálja! Le is lehessen vele generálni az összes részhalmazt! (A halmaz itt egy olyan rendezett lista, amelyben minden elem egyszer fordul elő.)

```
?- részhalmaz([a, b, c], R).
R = [a, b, c] ;
R = [a, b] ;
R = [a, c] ;
R = [a] ;
R = [b, c] ;
R = [b] ;
R = [c] ;
R = []
```

(Az eredmény lehet más sorrendben.)

26. Feladat. Írj egy szabályt, amivel ellenőrizhető, hogy két lista ugyanolyan hosszú!

```
?- ugyanolyan_hosszú([1, 2, 3], [a, b, c]).
true
```

27. Feladat. Írj egy szabályt, amivel ki lehet „lapítani” egy listát, tehát a belső listák elemeit kiviszi a legkülső szintre:

```
?- lapít([a, b, [c, d], [], [[[e]]], f], L).
L = [a, b, c, d, e, f]
```

3.2. Projekt: Dobble Kids

A Dobble Kids társasjáték 30 kártyából áll, ahol minden kártyán 6 különböző állat szerepel (összesen 31 fajtából), amelyekre az az érdekes tulajdonság teljesül, hogy bármely két lapon pontosan egy azonos állat van. Egy ilyen kártyapakli generálása nem

könnyű feladat, és nagyon szép matematika van a hátterében. Meg lehet mutatni, hogy ha egy lapon k állat van, akkor összesen $k(k-1)+1$ különböző kártya készíthető. Viszont $k=6$ esetén ez 31, nem 30, tehát van egy olyan lap, amit még hozzá lehet-

ne venni a paklihoz, hogy a tulajdonság továbbra is fennálljon. Keressük meg ezt a lapot!

Először is készítsük el a pakliban levő kártyák listáját! Minden kártya állatok listája, tehát a pakli listák listája lesz:

```
lapok([[bagoly,bálna,hal,kacsa,rák,tehén],
       [bagoly,bárány,elefánt,polip,teknős,teve],
       [bagoly,béka,delfin,gorilla,kígyó,kutya],
       [bagoly,cápa,cica,kakas,nyuszi,pingvin],
       [bagoly,katica,kenguru,krokodil,tigris,zebra],
       [bagoly,ló,medve,oroszlán,papagáj,víziló],
       [bálna,bárány,cica,delfin,kenguru,papagáj],
       [bálna,béka,cápa,ló,teknős,zebra],
       [bálna,elefánt,kutya,nyuszi,oroszlán,tigris],
       [bálna,gorilla,kakas,krokodil,medve,teve],
       [bálna,katica,kígyó,pingvin,polip,víziló],
       [bárány,béka,kacsa,kakas,katica,oroszlán],
       [bárány,cápa,gorilla,hal,tigris,víziló],
       [bárány,kígyó,krokodil,ló,nyuszi,tehén],
       [bárány,kutya,medve,pingvin,rák,zebra],
       [béka,cica,elefánt,krokodil,rák,víziló],
       [béka,hal,kenguru,medve,nyuszi,polip],
       [béka,papagáj,pingvin,tehén,teve,tigris],
       [cápa,delfin,elefánt,katica,medve,tehén],
       [cápa,kacsa,krokodil,kutya,papagáj,polip],
       [cápa,kenguru,kígyó,oroszlán,rák,teve],
       [cica,gorilla,oroszlán,polip,tehén,zebra],
       [cica,kacsa,kígyó,medve,teknős,tigris],
       [delfin,hal,krokodil,oroszlán,pingvin,teknős],
```

```
25          [delfin,kacsa,nyuszi,teve,víziló,zebra],
26          [delfin,kakas,ló,polip,rák,tigris],
27          [elefánt,gorilla,kacsa,kenguru,ló,pingvin],
28          [elefánt,hal,kakas,kígyó,papagáj,zebra],
29          [gorilla,katica,nyuszi,papagáj,rák,teknős],
30          [kakas,kenguru,kutya,tehén,teknős,víziló]]).
```

Szintén hasznos lehet az összes állat listája:

```
állatok([bagoly,bálna,bárány,béka,cápa,cica,delfin,
         elefánt,gorilla,hal,kacsa,kakas,katica,
         kenguru,kígyó,krokodil,kutya,ló,medve,
         nyuszi,oroszlán,papagáj,pingvin,polip,rák,
         tehén,teknős,teve,tigris,víziló,zebra]).
```

Ha most ezekből az állatokból egy kártyát szeretnék képezni, akkor annak két dolgot kell teljesítenie: (i) 6 elemből kell állnia, és (ii) az állat-lista részhalmazának kell lennie. Az alábbi szabály az `A` listából választ ki 6 különböző elemet:

```
választ6(A, L) :-
    L = [_, _, _, _, _, _],
    részhalmaz(A, L).
```

A **`részhalmaz`** szabály feladatként is szerepelt. Ebben feltételezzük, hogy az elemek azonos sorrendben fordulnak elő (tehát nincs olyan elempár, amely a két listában fordított sorrendben szerepelne). Ez a megkötés, mivel most úgy használjuk, hogy a részhalmaz elemei változók, azt vonja maga után, hogy a keletkező részhalmazban az elemek olyan sorrendben lesznek, mint ahogy a teljesben szerepeltek.

```
1 részhalmaz([], []).
2 részhalmaz([X|M], [X|R]) :- részhalmaz(M, R).
3 részhalmaz([_|M], R) :- részhalmaz(M, R).
```

Az üres halmaznak csak az üres halmaz a részhalmaza. Egyébként úgy kapunk meg egy részhalmazt, hogy az első elemet (`X`) vagy hozzáadjuk (2. sor), vagy nem adjuk hozzá (3. sor) a maradék (`M`) egy részhalmazához (`R`).

A következő feladatunk, hogy ha készítettünk egy lapot, akkor eldöntsük, hogy jó-e. Ezt úgy tudjuk megtenni, hogy összehasonlítjuk a többi lappal egyenként, és megnézzük, hogy az állat-halmazok metszete 1-elemű-e:

```
1 jó_lap([], _).
2 jó_lap([L|M], X) :- metszet(X, L, [_]), jó_lap(M, X).
```

Itt az első argumentum a többi kártya listája; ha ez üres, az azt jelenti, hogy minden kártyát leteszteltünk, készen vagyunk. Egyébként megnézzük, hogy az aktuális kártyára a metszet 1-elemű-e (hogy mi ez az elem, az nem érdekes), és még ellenőrizni kell a maradékra is.

A metszet számítása van még hátra:

```
1 metszet([], _, []).
2 metszet([X|L1], L2, [X|L3]) :-
3     tartalmaz(X, L2), metszet(L1, L2, L3).
4 metszet([X|L1], L2, L3) :-
5     nemtartalmaz(X, L2), metszet(L1, L2, L3).
```

Az üres halmaz metszete bármivel üres. Ha az első elem (`X`) szerepel a másik halmazban (`L2`), akkor benne lesz a metszetben is; különben nem.

Azt, hogy egy elem *nincs benne* egy listában, a `nemtartalmaz` fejezi ki:

```
nemtartalmaz(_, []).
nemtartalmaz(X, [Y|M]) :- X \= Y, nemtartalmaz(X, M).
```

Ezzel minden megvan már, csak össze kell rakni a darabokat:

```
megoldás(X) :-
    állatok(A), lapok(L),
    választ6(A, X), jó_lap(L, X).
```

Röviden: kiválasztunk 6 állatot úgy, hogy az így készült kártya jó legyen a pakli minden lapjához.

Ha kipróbáljuk, kicsit hosszabb gondolkodás után megkapjuk a megoldást:

```
?- megoldás(X).
X = [cica, hal, katica, kutya, ló, teve]
```

Ezen a programon is sokat tudunk majd javítani. Magasabb rendű szabályokkal kényelmesen le lehet generálni az `állatok` listát, és nem kell kézzel megadni; valamint a `nemtartalmaz`ra nem lesz szükség, ha megtanuljuk a *vágásokat*.

4. lecke

Operátorok

Van még szintaktikus cukor bizonyos struktúrákon: ezek az *operátorok*. Amikor például leírunk egy olyan matematikai kifejezést, hogy $2a + bc$, vagy a szorzásokat csillaggal (`*`) jelölve, `2*a+b*c`, akkor a szorzás és összeadás műveletek az argumentumaik *között* szerepelnek. Ezeket hívjuk *infix* („közbülső") operátoroknak. Ha egy ilyen kifejezést a szokásos *prefix* („elülső") funktorokkal szeretnénk leírni, akkor ezt kapnánk:

```
+(*(2, a), *(b, c))
```

Vannak nyelvek, amelyek ezt preferálják, és vannak olyanok, amelyek a *posztfix* („hátulsó") operátorokat:

```
((2, a)*, (b, c)*)+
```

... de a Prolog ezekre a megszokott infix jelöléseket támogatja.

pre-, in- és posztfix

Kizárólag prefix operátorokkal működnek a *Lisp* nyelvcsalád nyelvei (Common Lisp, Scheme, Clojure, Racket etc.) Ezeknél a funktor neve is a zárójelen belülre kerül, tehát a fenti példából ez lesz: `(+ (* 2 a) (* b c))`
Ennek az egyik előnye a következetesség, a másik pedig, hogy a műveletek kiterjeszthetőek több argumentumra. Lispben például értelmes a `(+ 1 2 3 4 5)` kifejezés, amelynek értéke 15.
Kizárólag posztfix operátorokkal működnek a *veremnyelvek*, mint pl. a Forth vagy a PostScript. Itt minden operátornak csak egyféle aritása lehet, de cserébe megszabadulunk a zárójelektől. A fenti példa Forthban: `2 a * b c * +`
Ezt hívják *Reverse Polish Notation*-nek (RPN, „fordított lengyel jelölés"). Sok számológép is használja; elsőre elég furcsa, de nagyon kényelmes és gyors, ha megszokjuk.

Tetszőleges funktorból lehet operátort csinálni. Az egyargumentumú funktorok lehetnek pre- vagy posztfixek, a kétargumentumúak pedig csak infixek. Ahhoz, hogy egy ilyen kifejezést értelmezni lehessen, még azt is kell tudni, hogy melyik operátornak van elsőbbsége – például a szorzást előbb kell elvégezni, mint az összeadást, tehát elsőbbsége van.

A Prologban az ilyen jellegű „beállításokat" olyan szabályokkal lehet megadni, amelyeknek nincsen feje, csak törzse. Néhány

példa operátorok definíciójára (ezek részei az alapbeállításnak):

```
1 :- op(200, fy, -).
2 :- op(400, yfx, *).
3 :- op(500, yfx, +).
4 :- op(1000, xfy, ',').
5 :- op(1200, xfx, :-).
```

Itt az első, 1 és 1200 közti szám adja meg a *precedenciát* (minél kisebb, annál korábban kell elvégezni az adott műveletet), a második a típusát, és a harmadik magát az operátort.

Hétféle típus létezik:

- `xfx`, `xfy`, `yfx`: infix operátorok
- `fx`, `fy`: prefix operátorok
- `xf`, `yf`: posztfix operátorok

Ezeket úgy kell értelmezni, hogy az `f` mutatja az operátor helyét, az `x` és `y` pedig az argumentumo(ka)t. Ha egy argumentum `x`-el van jelölve, akkor – amennyiben az is egy operátoros kifejezés – az `x` operátorának szigorúan kisebb precedenciájúnak kell lennie az `f`-nél. Ezzel szemben az `y` esetében ez nem csak kisebb lehet, hanem egyenlő is. A zárójelezés minden előtt elsőbbséget élvez (0 a precedenciája).

Nézzünk ez alapján egy pár példát arra, hogy a Prolog különböző kifejezéseket hogyan elemez:

1. Mivel a `+` operátor `yfx` típusú, ha egy `a + b + c` kifejezésem van, akkor ezt nem értelmezhetem úgy, hogy `+(a,`

`+(b, c))`, mert akkor a külső `+` második argumentumában levő `+(b, c)` kifejezés precedenciája megegyezik a `+`-éval (hiszen az is `+`). Viszont ha `+(+(a, b), c)` módon értelmezem, ez a probléma nem lép fel: az első argumentumban levő `+(a, b)` kifejezés precedenciája ugyan ugyanaz, de ez megengedett, mert a baloldali egy `y`-argumentum. Tehát az `yfx` operátorokat **balról jobbra** kell zárójelezni.

2. A `-a * (b + c * d)` kiértékelését először a zárójeles részszel kell kezdeni. Ezt nem értelmezhetem úgy, hogy `*(+(b, c), d)`, mivel a `+` precedenciája nagyobb a `*`-énál, az `yfx` szerint pedig kisebbnek vagy egyenlőnek kéne lennie. Ezért a helyes értelmezés a `+(b, *(c, d))`. Ezután jön a 200-as precedenciájú 1-argumentumú `-` operátor, és végül a 400-as precedenciájú `*`. A teljes kifejezés tehát így néz ki: `*(-(a),+(b,*(c,d)))`.

3. A `P :- Q, R, S` kifejezés helyes zárójelezése `:-(P, ','(Q, ','(R, S)))`, mivel az `xfy` típusú vessző operátort **jobbról balra** kell zárójelezni, és a `:-` operátor precedenciája a legmagasabb.

4. Mivel a `:-` operátor `xfx` típusú, ezért **nem szerepelhet a saját argumentumaként**. Egy olyan kifejezésnek, hogy `a :- b :- c`, nincsen helyes zárójelezése.

5. A negálás `fy` típusú, ezért a `- -a` kifejezés is elfogadható (a két mínuszjelet el kell választani, különben egy `--` nevű operátor lesz belőle); ha `fx` típusú lenne, akkor ezt zárójelezni kéne `-(-a)` alakban.

A precedencia tehát többek közt azt is meghatározza, hogy mi egy kifejezésben az elsődleges funktor, ezért pl.

```
?- X+Y = 2*3+5.
X = 2*3
Y = 5
?- X*Y = 2*3+5.
false
```

Nem csak különleges karakterekkel megadott funktorokból készíthetünk operátorokat, hanem tetszőleges nevűekből. Ez magyarul kevésbé természetes, mint angolul, de azért nézzünk erre is példát!

```
:- op(600, xfx, benne_van).
X benne_van L :- tartalmaz(X, L).
```

Ez definiálja a `tartalmaz` funktor infix változatát. Ezután

```
?- b benne_van [a, b, c].
true
?- benne_van(b, [a, b, c]).
true
```

28. Feladat. Mit ad az alábbi program:

```
t(0+1, 1+0).
t(X+0+1, X+1+0).
t(X+1+1, Z) :- t(X+1, X1), t(X1+1, Z).
```

. . . a következő kérdésekre:

Kitérő: *izoláló nyelvek*

A magyar nyelv ragozási rendszere mellett egy-egy operátorrá tett szó magában még nem képes természetes mondatok készítésére, de más nyelvekben ez nagyon jól működik. Ezek az ún. *izoláló* nyelvek – jellemzően ilyenek a (dél)kelet-ázsiai nyelvek, pl. kínai, indonéz vagy thai, de bizonyos mértékig az angol is.

```
?- t(0+1, A).
?- t(0+1+1, B).
?- t(1+0+1+1+1, C).
?- t(D, 1+1+1+0).
```

4.1. Számolás

Ahogy az előző feladatban is látszott, a matematikai kifejezések a Prolog számára csak struktúrák, és ezért az

```
?- X = 1 + 2.
```

kérdésre azt a (nem túl sokatmondó) választ kapjuk, hogy

```
X = 1 + 2
```

Ha rá akarjuk kényszeríteni, hogy kiértékelje a kifejezést, az egyenlőség helyett az `is` operátort használhatjuk:

```
?- X is 1 + 2.
X = 3
```

A matematikai kifejezésekben használható az összeadás (`+`), kivonás/negáció (`-`), szorzás (`*`), osztás (`/`), egészosztás (`//`), hatványozás (`**`), és a maradékszámítás (`mod`). Néhány példa:

```
?- X is 5/2.
X = 2.5
?- X is 9//2.
X = 4 % 9-ben 4-szer van meg (teljesen) a 2
?- X is 2**128.
X = 340282366920938463463374607431768211456.
?- X is 10 mod 3.
X = 1 % 10-nek a 3-mal vett osztási maradéka 1
```

Szintén alkalmazhatóak olyan gyakran használt matematikai függvények, mint az abszolutérték (`abs`), a szinusz (`sin`), vagy a természetes alapú logaritmus (`log`).

A relációs jelek szintén „kiértékelő erővel” bírnak, tehát pl.

```
?- 2 + 2 > 3.
true
```

A nagyobb-egyenlő és kisebb-egyenlő relációk rendre `>=` és `=<`, a matematikai egyenlőség és különbözőség pedig `=:=` és `=\=`, pl.

```
?- 2 + 3 = 6 - 1.
false
?- 2 + 3 =:= 6 - 1.
```

```
true
?- 2 + 3 \= 6 - 1.
true
?- 2 + 3 =\= 6 - 1.
false
```

A matematikai kiértékelésnél feltétel, hogy a kifejezésben ne szerepeljen változó. Tehát a rendszer nem tudja azt megválaszolni, hogy:

```
?- X + 2 < X + 3.
```

... bármennyire is nyilvánvalónak tűnik nekünk.

Az `is` és az `=:=` időnként ugyan felcserélhető, de általában nem:

```
X is 1 + 2.  % OK, X = 3
X =:= 1 + 2. % Nem jó, nem lehet benne ismeretlen
3 is 1 + 2.  % OK, baloldalt lehet szám
3 =:= 1 + 2. % OK
1 + 2 is 3.  % Nem jó, baloldalt nem lehet kifejezés
1 + 2 =:= 3. % OK
```

Legnagyobb közös osztó

Két szám legnagyobb közös osztójának megkeresése klasszikus probléma. Az alábbi megoldás ötlete Euklidész *Elemek* c. művéből származik (i.e. 300 körül).

Legyen `lnko(N, M, O)` igaz, ha `N` és `M` legnagyobb közös osztója `O`. Ha a két szám azonos, akkor az osztó is az lesz:

```
lnko(N, N, N).
```

Ha az első szám a kisebb, akkor azt levonhatjuk a másodikból, és a legnagyobb közös osztó nem változik:

```
lnko(N, M, O) :-
    N < M,
    M1 is M - N,
    lnko(N, M1, O).
```

Végül, ha a második szám a kisebb, akkor egyszerűen cseréljük meg a két számot, és az előző esethez jutunk:

```
lnko(N, M, O) :-
    N > M,
    lnko(M, N, O).
```

Könnyen látszik, hogy a 3. típusú lépés után mindig 2. típusú jön, és a 2. típusú lépés után az `N + M` összeg csökken. Mivel a számok nem mehetnek 0 alá, előbb-utóbb biztosan eljut az 1. típusú lépéshez, ahol megkapjuk az eredményt.

Teszteljük!

```
?- lnko(1071, 462, X).
X = 21.
```

A fenti magyarázatok deklaratív jellegűek. A procedurális olvasata a programnak a következő:

1. Ha `N = M`, akkor a legnagyobb közös osztó `N`. Vége.
2. Ha `N < M`, akkor vonjuk ki `M`-ből `N`-et, és menjünk vissza az 1. lépésre.

3. Ha `N > M`, akkor cseréljük meg `N`-et és `M`-et, és menjünk vissza az 1. lépésre.

Az ilyen megoldási módszereket *algoritmusnak* nevezik.

Kitérő: *algoritmusok*

Bár Euklidész algoritmusa régi, de már sokkal régebben is léteztek hasonló módszerek, pl. az ókori Mezopotámiában a sumérok már i.e. 2500 körül ismertek algoritmust az osztásra. Maga az elnevezés egy Bagdadban tanító VIII–IX. századi perzsa matematikus nevéből származik, akit (arabosan) úgy hívtak, hogy Muhammad bin Músza *al-Khvárizmí* („a Hvárezmből való Mózes fia Mohamed").

Ha már a kezdeteknél tartunk, az első program, amelyet tényleg számítógépre írtak, Ada Lovelace (1815–1852) nevéhez fűződik, aki Byronnak, a híres költőnek a lánya volt. A program a Bernoulli-számokat számította ki; a gép pedig, amire írta, egy mechanikus számítógép volt, a *Difference Engine*. Ezt Charles Babbage (1791–1871) tervezte, de csak születésének 200 éves évfordulójára készült el (viszont működött!).

Listák hossza

Most, hogy már tudunk számolni, ki tudjuk számítani egy lista hosszát is:

```
hossz([], 0).
hossz([_|M], N) :- hossz(M, N1), N is 1 + N1.
```

Egy üres lista hossza 0, egyébként meg a maradék hossza plusz egy.

Ha a fenti definícióban az `is` helyett `=`-t használunk, akkor látjuk, hogyan számol:

```
?- hossz([a,b,c], X).
X = 1+(1+(1+0))
```

Egy másik lehetőség, hogy egy középső argumentumban számon tartjuk az eddigi hosszt:

```
hossz2([], N, N).
hossz2([_|M], C, N) :- C1 is 1 + C, hossz2(M, C1, N).
hossz2(L, N) :- hossz2(L, 0, N).
```

Itt a `hossz2(L1, C, N)`-re mindig igaz lesz, hogy a teljes lista hossza az az `L1` lista hossza + `C`-vel egyenlő.

A lényeges különbséget a `trace` mutatja:

```
?- trace, hossz([a, b, c], X).
Call: hossz([a, b, c], X)
  Call: hossz([b, c], X1)
    Call: hossz([c], X2)
      Call: hossz([], X3)
      Exit: hossz([], 0)
      Call: X2 is 1+0
      Exit: 1 is 1+0
    Exit: hossz([c], 1)
```

```
    Call: X1 is 1+1
    Exit: 2 is 1+1
  Exit: hossz([b, c], 2)
  Call: X is 1+2
  Exit: 3 is 1+2
Exit: hossz([a, b, c], 3)
X = 3

?- trace, hossz2([a, b, c], X).
Call: hossz2([a, b, c], X)
  Call: hossz2([a, b, c], 0, X)
    Call: C1 is 0+1
    Exit: 1 is 0+1
    Call: hossz2([b, c], 1, X)
      Call: C2 is 1+1
      Exit: 2 is 1+1
      Call: hossz2([c], 2, X)
        Call: C3 is 2+1
        Exit: 3 is 2+1
        Call: hossz2([], 3, X)
        Exit: hossz2([], 3, 3)
      Exit: hossz2([c], 2, 3)
    Exit: hossz2([b, c], 1, 3)
  Exit: hossz2([a, b, c], 0, 3)
Exit: hossz2([a, b, c], 3)
X = 3
```

Az első verzióban, miután elértünk az üres listáig, még hozzá kell adogatnunk az 1-eket a hosszhoz. A második verzióban viszont

ilyenkor már nincs más feladatunk, csak visszatérni a már kiszámolt eredménnyel. Ez utóbbit úgy hívják, hogy „vég-rekurzió”, mivel a szabálynak a legvégén van a rekurziós lépés. Az algoritmusok ilyen felírása gyakran hatékonyabb, ahogy azt mindjárt látni fogjuk.

Fibonacci-számok

A Fibonacci-számokat úgy képezzük, hogy elkezdjük két 1-essel, és utána mindig az előző két szám összegét vesszük:

```
1 1 2 3 5 8 13 21 34 55 ...
```

Számoljuk ki az n-edik Fibonacci-számot!

```
1 fib(1, 1).
2 fib(2, 1).
3 fib(N, M) :-
4     N1 is N - 1, N2 is N - 2,
5     fib(N1, K1), fib(N2, K2),
6     M is K1 + K2.
```

Próbáljuk ki!

```
?- fib(10, X).
X = 55
?- fib(20, X).
X = 6765
```

Működik, de nagyobb számokra (pl. 40) már nem tudja kiszámolni. A problémát az okozza, hogy a kisebb Fibonacci-számokat feleslegesen újra és újra, rengetegszer kiszámolja.

Próbáljuk ezt is vég-rekurzióval megoldani! Két plusz argumentumként vegyük fel az „előző” két számot (`K1` és `K2`):

```
fib2(1, M, _, M).
fib2(N, K1, K2, M) :-
    N1 is N - 1, K3 is K1 + K2,
    fib2(N1, K2, K3, M).
fib2(N, M) :- fib2(N, 1, 1, M).
```

Nézzük meg, mit csinál! Először beállítja `K1`-et és `K2`-t 1-re, majd minden lépésben (i) csökkenti az `N`-et, (ii) a `K1` helyére rakja a `K2`-et, és (iii) a `K2` helyére rakja a (régi) `K1` és `K2` összegét. Ez alapján könnyen belátható, hogy a `fib2(I, K1, K2, M)`-re mindig igaz lesz, hogy az `N-I+1`-edik Fibonacci szám a `K1` lesz (`I = N` esetén világos; a második szabály pedig nem rontja el).

Így már nagy értékekre is jól működik, hiszen minden Fibonacci-számot csak egyszer számol ki.

29. Feladat.

Írj szabályt, amivel két szám közül ki lehet választani a nagyobbat!

```
?- max(2, 5, X).
X = 5
```

30. Feladat.

Írj szabályt, amivel egy listából ki lehet választani a legnagyobb elemet!

```
?- maximum([5, 2, 8, 3], X).
X = 8
```

31. Feladat.

Írj szabályt, amivel ki lehet számolni egy listában levő számok összegét!

```
?- összeg([5, 2, 8, 3], X).
X = 18
```

32. Feladat.

Írj szabályt, ami eldönti, hogy egy lista elemei növekvő sorrendben vannak-e!

```
?- növekvő([2, 5, 6, 8]).
true
?- növekvő([2, 6, 5, 8]).
false
```

33. Feladat.

Írj szabályt, amivel egy listából ki lehet választani elemeket úgy, hogy az összegük egy adott szám legyen!

```
?- részösszeg([1, 2, 5, 3, 2], 5, X).
X = [1, 2, 2]
X = [2, 3]
X = [5]
...
```

34. Feladat.

Írj egy `között(N1, N2, X)` szabályt, ami eldönti, hogy az `X` az `N1` és az `N2` között van-e (a határokat beleértve)!

```
?- között(2, 5, 3).
true
?- között(2, 5, 5).
true
?- között(2, 5, X).
X = 2
X = 3
...
```

35. Feladat.

Készíts `ha`, `akkor`, `egyébként` és `:=` operátorokat, hogy lehessen ilyeneket írni:

```
ha X > Y akkor Z := A egyébként Z := B
```

Ennek hatására a `Z` egyesül a `A`-val vagy `B`-vel attól függően, hogy `X` nagyobb-e, mint `Y`. Például:

```
?- X = 2, Y = 3,
   X2 is 2 * X, X4 is 4 * X,
   ha Y > X2 akkor Z := Y egyébként Z := X4,
   ha Z > 5 akkor W := 1 egyébként W := 0.
X = 2
Y = 3
Z = 8
W = 1
X2 = 4
X4 = 8
```

Prefix jelölésben ugyanez így nézne ki:

```
ha(akkor(>(X, Y), egyébként(:=(Z, A), :=(Z, B))))
```

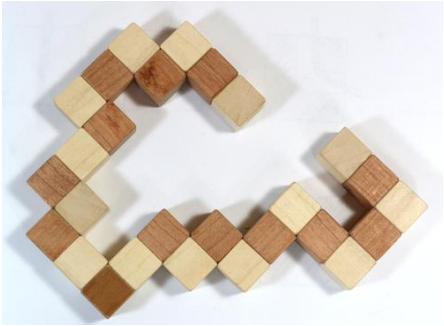
A szabály maga tehát nagyon egyszerű:

```
1 ha(akkor(>(X, Y), egyébként(:=(Z, V), _))) :-
2     X > Y, Z = V.
3 ha(akkor(>(X, Y), egyébként(_, :=(Z, V)))) :-
4     X =< Y, Z = V.
```

Itt érdemes megjegyezni, hogy a baloldalon szereplő > operátornak nincsen semmilyen matematikai értelme, csak egy struktúra funktoraként szerepel.

A feladat tehát az, hogy állítsd be úgy az operátorokat, hogy a fenti példa működjön.

4.2. Projekt: Kígyó-kocka



Oldjuk meg a jól ismert „kígyó-kocka” feladványt! Az egyes kis szakaszok fixek, de a derékszögű fordulásoknál egymáson elforgathatóak; a feladat, hogy kirakjunk belőle egy 3x3-as kockát.

Szokás szerint azzal kezdjük, hogy felvesszük az adatokat. Ez a készítendő kocka mérete, és a kígyót alkotó szakaszok hosszai:

```
méret(3).
kígyó([3,2,2,3,2,3,2,2,3,3,2,2,2,3,3,3,3]).
```

A teljes kockán belül minden pozíciót 3 koordinátával tudunk jellemezni:

```
pozíció(p(X, Y, Z)) :-
    méret(N), között(1, N, X),
    között(1, N, Y), között(1, N, Z).
```

Tehát a jelenlegi 3×3-as kocka esetében minden koordináta 1 és 3 között változhat.

A `között` szabály feladatként szerepelt, így definiálhatjuk:

```
között(N, M, N) :- N =< M.
között(N, M, X) :-
    N < M, N1 is N + 1,
    között(N1, M, X).
```

Itt arra érdemes figyelni, hogy `N > M` esetén ez nem kielégíthető.

Hatféle irányról beszélhetünk, a három tengely irányában pozitív és negatív irányban. Ezeket leírhatjuk olyan módon, mint pl. `i(x, -1)` vagy `i(z, 1)`. Az összes irányt tehát így fogalmazhatjuk meg:

```
irány(i(T,I)) :-
    tartalmaz(T, [x,y,z]),
    tartalmaz(I, [-1,1]).
```

A `T` itt az egyik tengely, az `I` pedig 1 vagy -1, ami a pozitív ill. negatív irányt jelöli.

Két egymás után következő szakasz mindig merőleges egymásra, tehát a *tengelyük* nem lehet azonos:

```
köv_irány(i(T,_), i(T1,I)) :-
    irány(i(T1,I)), T \= T1.
```

Hogyan változik egy pozíció, ha ellépünk egy adott irányban? A `lépés(P1, I, P2)` akkor lesz igaz, amikor a `P1`-ből `I` irányba ellépve a `P2`-be jutunk:

```
lépés(p(X,Y,Z), i(x,I), p(X1,Y,Z)) :- X1 is X + I.
lépés(p(X,Y,Z), i(y,I), p(X,Y1,Z)) :- Y1 is Y + I.
lépés(p(X,Y,Z), i(z,I), p(X,Y,Z1)) :- Z1 is Z + I.
```

(Ezért volt jó a pozitív/negatív irányt 1-el és -1-el jelölni.)

Próbáljuk meg felírni a megoldást! Legyen erre a szabályunk `megoldás(K, PL, I, X)`; itt `K` a kígyó hátralevő része, `PL` a már kitöltött pozíciók listája (az elején a kígyó feje, amit épp vizsgálunk), az `I` az aktuális irány, és `X` az irányváltoztatások listája. A reláció akkor teljesül, ha a `PL` pozíció-lista első elemétől `I` irányban indulva le tudjuk rakni a `K` listában levő szakaszokat az `X` listában levő irányokat követve úgy, hogy mindig a kockán belül maradunk, és nem megyünk bele a `PL` egyik elemébe sem. Ekkor a

```
?- kígyó(K), pozíció(P), irány(I),
   megoldás(K, [P], I, X).
```

kérdésre `P` adja a kezdő pozíciót, `I` a kezdő irányt, és `X` az irányváltoztatásokat.

Ha a kígyónak már csak 1 szakasza van hátra, akkor csak azt kell megnézni, hogy azt le tudjuk-e tenni:

```
megoldás([N], PL, I, [I]) :- ellenőriz(PL, I, N, _).
```

Az `ellenőriz(PL, I, N, PL1)` akkor lesz igaz, ha a `PL` elején levő pozícióról az `I` irányban le tudunk helyezni egy `N` hosszú szakaszt anélkül, hogy kimennénk a kockából vagy érintenénk egy `PL`-ben levő pozíciót, és az így bővült pozíció-lista a `PL1`.

Ezt szavakban elmondani bonyolultabb, mint programban:

```
ellenőriz(PL, _, 1, PL).
ellenőriz([P|M], I, N, PL1) :-
    N > 1, lépés(P, I, P1),
    pozíció(P1), nemtartalmaz(P1, M), N1 is N - 1,
    ellenőriz([P1, P|M], I, N1, PL1).
```

Ha a szakasz 1 kockából áll, akkor nincs további teendőnk, és `PL1 = PL`. Egyébként teszünk a megadott irányban egy lépést, megnézzük, hogy érvényes pozíció-e, nem szerepel-e az eddigi pozíciók listájában, és ha ez mind jó, akkor jöhet a többi lépés – ehhez a `PL`-et kiegészítjük az új `P1` pozícióval, és a készítendő szakasz hosszát csökkentjük 1-el.

Ha a kígyó több szakaszból áll, akkor először ellépünk az első szakasz hosszával az aktuális irányban, utána új irányt választunk, és rekurzióval elvégezzük a maradékot:

```
megoldás([N|M], PL, I, [I|X]) :-
    ellenőriz(PL, I, N, PL1), köv_irány(I, I1),
    megoldás(M, PL1, I1, X).
```

Ezzel a fent ismertetett módon megkapjuk a megoldást, de kicsit nehezen olvasható alakban. Fordítsuk le magyarra az irányokat!

```
fordít([], []).
fordít([i(T, I)|M], [F|FM]) :-
    (  T = x, (I = 1, F = jobbra; I = -1, F = balra)
     ; T = y, (I = 1, F = fel;    I = -1, F = le)
     ; T = z, (I = 1, F = előre;  I = -1, F = hátra)
    ),
    fordít(M, FM).
```

A `fordít` szabály első argumentuma irányok egy listája, a második pedig az ennek megfelelő fordítások listája. Érdemes megfigyelni a zárójelezés és a logikai vagy jelentésű pontosvesszők használatát.

Végül akkor tegyük az egész megoldót egy szabályba!

```
kígyó_kocka(K, P, FI, FX) :-
    kígyó(K), pozíció(P), irány(I),
    megoldás(K, [P], I, X), fordít([I|X], [FI|FX]).
```

Ha kipróbáljuk:

```
?- kígyó_kocka(K, P, FI, FX).
FI = jobbra,
FX = [jobbra, fel, balra, előre, fel, hátra, jobbra,
```

```
      előre, le, balra, fel, hátra, fel, előre, le,
      jobbra, fel],
K = [3, 2, 2, 3, 2, 3, 2, 2, 3, 3, 2, 2, 2, 3, 3, 3,
     3],
P = p(1, 1, 1)
```

... az eredmény már sokkal könnyebben értelmezhető.

A megoldó elég általános ahhoz, hogy más változatokra is használható legyen, pl. a „mean green” feladványra, ha lecseréljük a `kígyó` definícióját:

```
kígyó([3,3,2,3,2,3,2,2,2,3,3,3,2,3,3,3]).
```

Egy másik variáns a „king” feladvány, amelynél 4×4-es kockát kell építeni:

```
méret(4).
kígyó([3,2,3,2,2,4,2,3,2,3,2,3,2,2,2,2,
       2,2,2,2,3,3,2,2,2,2,2,2,3,4,2,2,2,
       4,2,3,2,2,2,2,2,2,2,2,2,2,4,2]).
```

Ennek sokkal több lehetőséget kell megvizsgálnia, ezért kell neki pár perc.

Megfordíthatjuk a kérdést: hogyan lehet egy számsorozatot készíteni, ami olyan kígyót határoz meg, amelyből kocka készíthető? Ehhez a `kígyó`-nak egy új definíciójára lesz szükségünk:

```
kígyó(K) :-
    méret(N), N3 is N^3, N1 is N3 // (N - 1),
    között(N1, N3, H), hossz(K, H),
    kígyó(K, N3).
```

A teljes kígyó hossza a méret köbe (`N3`). A kígyót alkotó szakaszok száma (a kígyónak, mint szakasz-listának a *hossza*, `H`) tehát `N3/(N-1)` és `N3` között lesz, hiszen minden szakasz legfeljebb `N-1` új pozíciót fedhet le. Az egyes szakaszokat ezután a `kígyó/2` segítségével készítjük el:

```
kígyó([], 1).
kígyó([Sz|K], M) :-
    M > 1, méret(N), között(2, N, Sz),
    M1 is M - Sz + 1, kígyó(K, M1).
```

Ha még `M` hosszú kígyót kell csinálni, akkor választunk egy számot 2 és `N` között (egy szakasz hossza csak ilyen lehet), ez lesz a mostani szakaszunk hossza, és a maradék szakaszokkal pedig egy ennyivel rövdebb kígyót csinálunk (pontosabban eggyel hosszabbat, mert egy szakasz utolsó eleme egyben a következő első eleme). Amikor már csak 1 hosszú kígyót kell csinálni, készen vagyunk.

Ezzel a definícióval azt kapjuk, hogy

```
?- kígyó_kocka(K, P, FI, FX).
K = [2, 3, 3, 2, 3, 3, 3, 3, 3, 3, 3, 2, 3, 2, 3],
P = p(1, 2, 2),
FI = le,
FX = [le, jobbra, fel, hátra, le, balra, fel, előre,
      le, jobbra, fel, balra, hátra, le, előre]
```

(Ez jó sokáig tart.) Mivel a szakaszok számával alulról felfele próbálkozunk, az első megoldás a legkevesebb szakaszból álló kígyó (15 db), amit 3×3-as kockába lehet rendezni. Ha a fenti

definícióban a `között(N1, N3, H)` helyett `H = 15`-öt írunk (tehát ha lerögzítjük a szegmensek számát), akkor gyorsan lefut.

A futási idő mindig egy kompromisszum: (aránylag) keveset kellett gondolkodnunk ahhoz, hogy megírjuk ezt a programot. Hatékonyabb programot gyakran nehezebb írni, viszont néha elég a lassabb is, pl. olyan feladatoknál, mint a kígyó-generálás, amikor a lényeg csak az, hogy találjunk egy megoldást (még ha akár napokig is kell a gépnek számolnia). Azért persze törekedjünk a hatékonyságra!

A teljes program

Mivel ez már egy elég összetett program volt, itt van egyben az egész:

```
1  tartalmaz(X, [X|_]).
2  tartalmaz(X, [_|M]) :- tartalmaz(X, M).
3
4  nemtartalmaz(_, []).
5  nemtartalmaz(X, [Y|M]) :- X \= Y, nemtartalmaz(X, M).
6
7  hossz([], 0).
8  hossz([_|M], N) :- hossz(M, N1), N is 1 + N1.
9
10 között(N, M, N) :- N =< M.
11 között(N, M, X) :-
12     N < M, N1 is N + 1,
13     között(N1, M, X).
14
```

```
15 % Standard
16 méret(3).
17 kígyó([3,2,2,3,2,3,2,2,3,3,2,2,2,3,3,3,3]).
18
19 % Mean green
20 % méret(3).
21 % kígyó([3,3,2,3,2,3,2,2,2,3,3,3,2,3,3,3]).
22
23 % King
24 % méret(4).
25 % kígyó([3,2,3,2,2,4,2,3,2,3,2,3,2,2,2,2,
26 %        2,2,2,2,3,3,2,2,2,2,2,3,4,2,2,2,
27 %        4,2,3,2,2,2,2,2,2,2,2,2,2,4,2]).
28
29 % Generáló
30 % méret(3).
31 % kígyó(K) :-
32 %     méret(N), N3 is N^3, N1 is N3 // (N - 1),
33 %     között(N1, N3, H), hossz(K, H),
34 %     kígyó(K, N3).
35 % kígyó([], 1).
36 % kígyó([Sz|K], M) :-
37 %     M > 1, méret(N), között(2, N, Sz),
38 %     M1 is M - Sz + 1, kígyó(K, M1).
39
40 pozíció(p(X, Y, Z)) :-
41     méret(N), között(1, N, X),
42     között(1, N, Y), között(1, N, Z).
43
```

```
44 irány(i(T,I)) :-
45     tartalmaz(T, [x,y,z]),
46     tartalmaz(I, [-1,1]).
47
48 köv_irány(i(T,_), i(T1,I)) :-
49     irány(i(T1,I)), T \= T1.
50
51 lépés(p(X,Y,Z), i(x,I), p(X1,Y,Z)) :- X1 is X + I.
52 lépés(p(X,Y,Z), i(y,I), p(X,Y1,Z)) :- Y1 is Y + I.
53 lépés(p(X,Y,Z), i(z,I), p(X,Y,Z1)) :- Z1 is Z + I.
54
55 ellenőriz(PL, _, 1, PL).
56 ellenőriz([P|M], I, N, PL1) :-
57     N > 1, lépés(P, I, P1),
58     pozíció(P1), nemtartalmaz(P1, M), N1 is N - 1,
59     ellenőriz([P1, P|M], I, N1, PL1).
60
61 megoldás([N], PL, I, [I]) :- ellenőriz(PL, I, N, _).
62 megoldás([N|M], PL, I, [I|X]) :-
63     ellenőriz(PL, I, N, PL1), köv_irány(I, I1),
64     megoldás(M, PL1, I1, X).
65
66 fordít([], []).
67 fordít([i(T, I)|M], [F|FM]) :-
68     (  T = x, (I = 1, F = jobbra; I = -1, F = balra)
69      ; T = y, (I = 1, F = fel;    I = -1, F = le)
70      ; T = z, (I = 1, F = előre;  I = -1, F = hátra)
71     ),
72     fordít(M, FM).
```

```
73
74 kígyó_kocka(K, P, FI, FX) :-
75     kígyó(K), pozíció(P), irány(I),
76     megoldás(K, [P], I, X), fordít([I|X], [FI|FX]).
```

5. lecke

Vezérlés

5.1. Rendezések

Egy klasszikus – és nagyon hasznos – programozási feladat számok listájának növekvő sorrendbe rendezése. Erre rengeteg módszer van, itt csak a legfontosabbak közül nézünk meg néhányat.

Beszúrásos rendezés

Az első algoritmus a *beszúrásos rendezés*. Az ötlet a következő: ha van egy listám, akkor azt szétszedem az első elemre és a maradékára, az utóbbit (rekurzívan) rendezem, és végül az első elemet beszúrom a „megfelelő” helyre.

```
rendez1([], []).
rendez1([X|M], Y) :-
    rendez1(M, M1), beszúr(X, M1, Y).
```

A megfelelő helyre való beszúrásnál pedig egyszerűen addig megyünk, amíg egy legalább akkora elemet nem találunk:

```
beszúr(X, [], [X]).
beszúr(X, [Y|M], [X,Y|M]) :- X =< Y.
beszúr(X, [Y|M], [Y|M1]) :- X > Y, beszúr(X, M, M1).
```

Próbáljuk ki!

```
?- rendez1([4,1,9,2,6,0,3,2,5,1], X).
X = [0, 1, 1, 2, 2, 3, 4, 5, 6, 9]
```

Ennek az algoritmusnak egy nagy előnye, hogy akkor is jól használható, ha az adatokat folyamatosan kapjuk, hiszen ha már van egy rendezett listánk, akkor utána mindig elég a `beszúr` műveletet használni. Egy másik jó tulajdonsága, hogy egy már rendezett sorozatnál nem végez plusz munkát, épp csak ellenőrzi, hogy a lista tényleg jó sorrendben van:

```
?- trace, rendez1([1,2,3], X).
Call:rendez1([1, 2, 3], X)
  Call:rendez1([2, 3], X1)
    Call:rendez1([3], X2)
      Call:rendez1([], X3)
      Exit:rendez1([], [])
      Call:beszúr(3, [], X2)
      Exit:beszúr(3, [], [3])
```

```
    Exit:rendez1([3], [3])
    Call:beszúr(2, [3], X1)
      Call:2=<3
      Exit:2=<3
    Exit:beszúr(2, [3], [2, 3])
  Exit:rendez1([2, 3], [2, 3])
  Call:beszúr(1, [2, 3], X)
    Call:1=<2
    Exit:1=<2
  Exit:beszúr(1, [2, 3], [1, 2, 3])
Exit:rendez1([1, 2, 3], [1, 2, 3])
X = [1, 2, 3]
```

Ugyanakkor ha nincs szerencsénk, nem túl jó hatásfokú: ha a lista n hosszú, akkor kb. n^2 összehasonlítást végez a legrosszabb esetben.

Gyorsrendezés

A gyorsrendezésnél kiválasztunk egy tetszőleges elemet (pl. az elsőt), és a többit két részre osztjuk aszerint, hogy nagyobb-e ennél vagy sem. Ezután a két részt külön–külön rendezzük (rekurzívan), majd az eredmény ezek összefűzéséből adódik:

```
1 rendez2([], []).
2 rendez2([X|M], Y) :-
3     szétoszt(X, M, Kicsi, Nagy),
4     rendez2(Kicsi, K),
5     rendez2(Nagy, N),
6     hozzáfűz(K, [X|N], Y).
```

A szétosztás elég magától értetődő:

```
szétoszt(_, [], [], []).
szétoszt(X, [Y|M], K, [Y|N]) :-
    X =< Y, szétoszt(X, M, K, N).
szétoszt(X, [Y|M], [Y|K], N) :-
    X > Y, szétoszt(X, M, K, N).
```

Ez (nagyon hosszú listákra) bizonyíthatóan sokkal kevesebb ellenőrzést végez, mint a beszúrásos rendezés. A fenti program azonban a hozzáfűzések miatt nem igazán hatékony – a következő leckében majd szó lesz a különbség-listákról, amelyek segítségével sokkal gyorsabbá tehető.

Fésülő rendezés

Utolsónak nézzünk meg egy nagyon hasonló algoritmust: itt is két részre bontjuk mindig a listát, azokat rendezzük, majd összefűzzük, de itt nem a szétválasztás a bonyolultabb, hanem az összefűzés, vagy ez esetben az *összefésülés*.

A listát a felénél kettéosztjuk, és mindkét részt rendezzük (rekurzívan), végül a két rendezett listát egy listává fésüljük össze:

```
rendez3([], []).
rendez3([X], [X]).
rendez3(X, Y) :-
    kettéoszt(X, X1, X2),
    rendez3(X1, Y1),
    rendez3(X2, Y2),
```

```
7       összefésül(Y1, Y2, Y).
8
9   kettéoszt([], [], []).
10  kettéoszt([X], [X], []).
11  kettéoszt([X,Y|M], [X|M1], [Y|M2]) :-
12      kettéoszt(M, M1, M2).
```

Az összefésülésnél a két lista elemét összehasonlítjuk, és a megfelelőt berakjuk a maradékok összefűzéséből kapott listába:

```
1   összefésül([X|Mx], [Y|My], [X|M]) :-
2       X =< Y, összefésül(Mx, [Y|My], M).
3   összefésül([X|Mx], [Y|My], [Y|M]) :-
4       X > Y, összefésül([X|Mx], My, M).
5   összefésül(X, [], X).
6   összefésül([], Y, Y).
```

Itt érdekes módon azt tapasztaljuk, hogy a (helyes) eredményt végtelen sokszor megkapjuk. Ez azért van, mert a `rendez3` harmadik szabálya az egyelemű listákra is meghívódhat. Ezt ki lehetne védeni azzal, hogy megköveteljük, hogy legyen legalább két eleme:

```
1   rendez3(X, Y) :-
2       X = [_,_|_], Y = [_,_|_],
3       kettéoszt(X, X1, X2),
4       rendez3(X1, Y1),
5       rendez3(X2, Y2),
6       összefésül(Y1, Y2, Y).
```

. . . de ez nem a legelegánsabb. (Hasonlóan, az `összefésül` utolsó két szabálya közt is van átfedés.) Nincs valami jobb mód erre?

furcsa rendezések

Rendezési algoritmusokból nincs hiány. Bizonyítható, hogy a gyorsrendezés, a fésülő rendezés (és még sok másik) a lehető leghatékonyabb. . . kivéve egy-két furcsa módszert:

1. Vágjunk (száraz) spagettitésztákat az egyes számoknak megfelelő hosszokra, majd marokra fogva tegyük le az asztalra. Ezután egy kartonlapot rátéve mindig sorban vegyük ki azt, ami hozzáér a laphoz, és így egy csökkenő sorrendezéshez jutunk.

2. Vegyük a lista egy véletlenszerű permutációját, és ellenőrizzük, hogy sorban van-e. Ha igen, készen vagyunk. Ha nem, akkor robbantsuk fel a világot. Feltéve, hogy igaz a párhuzamos univerzumok elmélete, csak az az univerzum fog megmaradni, amelyikben a véletlen permutáció pont a helyes sorbarendezés volt.

5.2. Vágás

Egy szabály törzsében *vágást* eszközölhetünk a `!` segítségével. Ez azt mondja, hogy ha már idáig eljutottunk, akkor vagy sike-

rül teljesíteni a törzs maradék részét, vagy ha nem, akkor úgy vesszük, hogy ezzel a fejjel való egyesítés sikertelen volt, nem próbálunk a `!` előtti kifejezésekre visszamenni, vagy az azonos fejhez tartozó esetleges többi szabályt megnézni.

Nézzük meg például az alábbi szabályokat, ahol `A`, `B`, `C` stb. kifejezéseket jelölnek:

```
1 C :- P, Q, R, !, S, T, U.
2 C :- V.
3
4 A :- B, C, D.
```

Ekkor ha az

```
?- A.
```

kérdést feltesszük, a következő fog történni:

1. Megnézi, hogy `B` teljesül-e (tegyük fel, hogy igen).
2. Megnézi, hogy `P`, `Q`, `R` teljesülnek-e (tegyük fel, hogy igen).
3. Megnézi, hogy `S`, `T`, `U` teljesülnek-e. Tegyük fel, hogy az `S` és `T` teljesül, de az `U` nem. Ekkor szokás szerint visszamegy a `T`-re, és megkeresi annak egy másik megoldását stb.
4. Ha nem sikerült az `S`, `T`, `U`-t teljesíteni, akkor – a vágás miatt – már nem megy vissza, hogy az `R` egy másik megoldását keresse, és nem próbálkozik a `C`-hez tartozó második szabállyal sem, hanem egy szinttel feljebb megy, és a `B`-hez keres egy másik megoldást.

Összefésülés hatékonyabban

Nézzük meg, hogyan lehet ezzel feljavítani az összefésülést!

```
1 összefésül([X|Mx], [Y|My], [X|M]) :-
2     X =< Y, !, összefésül(Mx, [Y|My], M).
3 összefésül([X|Mx], [Y|My], [Y|M]) :-
4     X > Y, !, összefésül([X|Mx], My, M).
5 összefésül(X, [], X) :- !.
6 összefésül([], Y, Y) :- !.
```

A vágások a programot hatékonyabbá teszik: ha az első szabályban láttuk, hogy `X =< Y`, már nem kell ellenőrizni a többi szabályt stb. (Az utolsó sorban a vágás felesleges, csak a szimmetria kedvéért került bele.)

Hasonlóan, a `rendez3` második szabályában:

```
1 rendez3([X], [X]) :- !.
```

Ezekkel a módosításokkal már egyértelmű (*determinisztikus*) lesz a megoldás.

A vágások nem változtatják meg a program értelmét (tehát, hogy milyen kifejezésekre lesz igaz), csak a hatékonyságát.

Maximum

Nézzünk egy másik példát! Az alábbi szabály két szám közül kiválasztja a nagyobbikat:

```
1 max1(X, Y, X) :- X >= Y.
2 max1(X, Y, Y) :- X < Y.
```

A fenti módon ez átírható így:

```
1 max2(X, Y, X) :- X >= Y, !.
2 max2(X, Y, Y) :- X < Y.
```

Felmerül azonban ekkor, hogy a második szabályban szükség van-e egyáltalán az `X < Y` összehasonlításra, hiszen csak akkor juthatunk oda, ha az `X >= Y` nem teljesült. A program akkor is működni látszik, ha elhagyjuk:

```
1 max3(X, Y, X) :- X >= Y, !.
2 max3(_, Y, Y).
```

Teszteljük egy kicsit!

```
?- max3(3, 5, X).
X = 5
?- max3(4, 2, X).
X = 4
?- max3(1, 5, 5).
true
?- max3(5, 1, 1). % Ajjaj...
true
```

Hoppá! Mi történt? Mivel az utolsó példában mindhárom argumentumnak van értéke, és az első és harmadik nem azonos, az első szabállyal nem is próbálkozik, hanem rögtön a másodikra megy, ami pedig most, hogy kivettük a feltételt, teljesül.

A `max3` változatban megváltozott a program deklaratív jelentése, hiszen egy olyan tényt tartalmaz, ami magában nézve nem igaz. Ez általában kerülendő, bár a hatékonysághoz néha

szükséges. Ha ilyen gondolatmenetet alkalmazunk, nagyon óvatosnak kell lenni – itt pl. biztosnak kell lennünk benne, hogy a harmadik argumentum mindig változó.

Hozzáadás halmazhoz

Egy további példaként nézzük meg az alábbi szabályt:

```
% hozzáad(+X, +Halmaz, -Eredmény).
% Hozzáadja X-et a Halmaz listához,
% de csak akkor, ha az még nem tartalmazta.
hozzáad(X, L, L) :- tartalmaz(X, L), !.
hozzáad(X, L, [X|L]).
```

A vágás nélkül itt a második szabályt nehezebb lenne megfogalmazni, kéne hozzá a 3. leckéhez tartozó projektben látott `nemtartalmaz` szabály, és következésképp a hatékonyságból is vesztene.

Cserébe itt is problémába ütközünk, ha a harmadik argumentum nem változó:

```
?- hozzáad(b, [a,b], [b,a,b]).
true
```

Ezt elkerülendő, érdemes a dokumentációval egyértelművé tenni a használatot. A fenti programhoz tartozó megjegyzés is ilyen szellemben íródott. Az első sorában egy egyezményes jelölésrendszert használ, melyben minden argumentum háromféle lehet:

– `+X`: kell, hogy legyen értéke

- `?X`: lehet értéke, de nem muszáj
- `-X`: csak változó lehet

Például:

1. `+X > +Y` [4. lecke]
2. `lapít(+Eredeti, -Lapos)` [3. lecke]
3. `tartalmaz(?Elem,?Lista)` [3. lecke]

Az első esetben mindkét kifejezésnek ismertnek kell lennie; a másodikban a lapítandó lista ismert, és a lapított verzió csak változó lehet; a harmadikban pedig mind a lista, mind a benne tartalmazandó elem lehet változó vagy ismert is.

36. Feladat. Nézzük meg az alábbi szabályokat!

```
1 p(1).
2 p(2) :- !.
3 p(3).
```

Mit lesz az összes válasz az alábbi kérdésekre:

```
?- p(X).
?- p(X), p(Y).
?- p(X), !, p(Y).
```

37. Feladat. Írd át hatékonyabbra a `szétoszt` szabályt vágások segítségével!

5.3. Tagadás

Hogyan tudjuk megfogalmazni azt, hogy „Csilla szeret minden állatot, kivéve a pókokat”?

```
szereti(csilla, X) :- pók(X), !, fail.
szereti(csilla, X) :- állat(X).
```

(Általában ilyenkor a `false` helyett a `fail`-t szokás használni, de a kettő jelentése azonos.)

Próbáljuk ki!

```
állat(tarantula).
állat(denevér).
pók(tarantula).
```

```
?- szereti(csilla, tarantula).
false
?- szereti(csilla, denevér).
true
```

Ez annyira hasznos, hogy érdemes bevezetni, mint tagadást:

```
nem(P) :- P, !, fail.
nem(_).
```

Figyeljük meg, hogy itt valami olyat csináltunk, amit eddig még soha: egy változót (`P`) a törzsben magában használtunk. Ez feltételezi, hogy a `P`-nek kiszámolható az igazságértéke. Például a fenti példát átírva:

```
szereti(csilla, X) :- állat(X), nem(pók(X)).
```

Itt a `P` értéke a `pók(X)` struktúra, és a `nem` törzsében levő `P` kiértékelésekor ezt mint megoldandó célkifejezést értelmezi. (A szabvány szerint elvileg `call(P)`-t kellene írni, de a legtöbb Prolog rendszerben a `P` is elegendő, erről bővebben lesz még szó a következő fejezetben.)

A „zárt világ” feltétel

A tagadásnak ez a módja nem mindig intuitív. Egy így tagadott kifejezés akkor lesz igaz, ha a kifejezés nem bizonyítható.

Mi történik, ha az előző programban felcseréljük a törzs két tagját?

```
szereti(csilla, X) :- nem(pók(X)), állat(X).
```

Érdekes módon Csilla most már a denevéreket sem szereti! Miért? Azért, mert a `nem(pók(X))`-ben az `X` egy változó, tehát a `pók(X)` bizonyítható az `X = tarantula` helyettesítéssel, így a `nem(pók(X))` nem teljesül. Az érték nélküli argumentumokkal tehát vigyázni kell.

Hasonlóan furcsa lehet, hogy

```
?- nem(állat(kutya)).
true
```

... de persze helyes, hiszen a programban levő szabályok alapján nem bizonyítható, hogy a kutya állat.

Ezek miatt a `nem` (vagy az angol `not`) helyett a `\+` operátort szokás használni, melynek definíciója ugyanaz, csak az elnevezés kevésbé félrevezető:

```
1 szereti(csilla, X) :- állat(X), \+ pók(X).
```

38. Feladat. Milyen kérdéssel lehet a `Jelöltek` listából kiválasztani azokat, akik nem szerepelnek a `Kiesettek` listában? Használd a `tartalmaz` szabályt és a tagadást!

39. Feladat. Készíts szabályt, ami két halmaz különbségét képzi! (A halmaz itt egy olyan lista, amelyben minden elem egyszer fordul elő.)

```
?- különbség([a,b,c,d], [f,d,b,e], X).
X = [a, c]
```

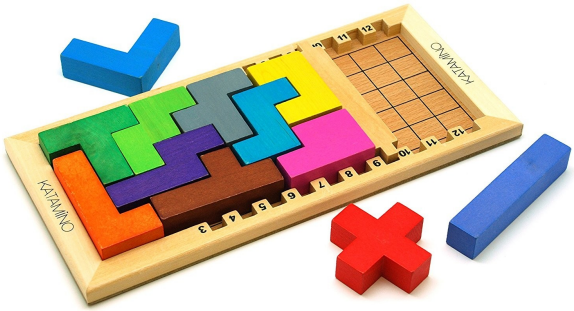
40. Feladat. Készíts szabályt, ami kiválogatja egy listából azokat a kifejezéseket, amelyek egy adott másik kifejezéssel egyesíthetőek!

```
?- egyesíthető([X,b,t(Y)], t(a), L).
L = [X, t(Y)]
```

Figyelj arra, hogy az `X` és `Y` ne kapjon ezáltal értéket!

5.4. Projekt: Katamino

Készítsünk megoldót a Katamino-feladványokhoz!

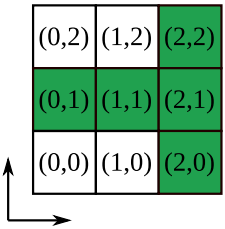


A feladat mindig az, hogy a 12 pentominó egy adott részhalmazából rakjunk ki egy $5 \times n$-es téglalapot.

Reprezentáció

Az első lépés továbbra is az, hogy valahogyan le kell írni a gép számára az adatokat, vagyis a használható alakzatokat. Sokféle leírás elképzelhető; itt most csináljuk a következőt: minden egyes alakzatot a körülírható téglalap bal alsó sarkához képest számolt koordinátáival írjuk le.

Például a T-alakzatnál:



Ha a bal alsó sarok a (0,0) pont, akkor az alakzathoz tartozó pontok a következők: (0,1), (1,1), (2,0), (2,1) és (2,2). A pontokat x-koordináta szerint, és azon belül y-koordináta szerint rendezve írjuk fel.

Az x és y-koordinátáknak összefogására bármilyen struktúrát használhatunk, pl. `p(0,1)`, de a tömörség kedvéért szokás a `-` operátort alkalmazni: `0-1`. Bár az egyes forgatások/tükrözések programból is legenerálhatóak, egyszerűbb mindegyiket külön adatként megadni.

A T esetében pl.:

```
1 alakzat(t, [0-0,0-1,0-2,1-1,2-1]).
2 alakzat(t, [0-0,1-0,1-1,1-2,2-0]).
3 alakzat(t, [0-1,1-1,2-0,2-1,2-2]).
4 alakzat(t, [0-2,1-0,1-1,1-2,2-2]).
```

Hasonlóan lehet definiálni a többi alakzatot, amelyek mind valamilyen 1 betűből álló kódot kaptak: `k`, `p`, `c` stb. (Az összes alakzat, a teljes program forráskódjával együtt, a lecke végén található.)

Megoldó

A programunk fő szabálya a `kirak` lesz, ami egy alakzat-listához megadja, hogy hogyan fog kinézni a tábla. A megoldás módszere nagyon egyszerű: mindig a (balról/alulról) első lyukat próbáljuk betömni.

```
1 kirak(Ak, X) :- hossz(Ak, N), kirak(Ak, N, [], X).
2
```

```
kirak([], _, T, T).
kirak(Ak, N, T, X) :-
    első_lyuk(N, T, P),
    töröl(A, Ak, Ak1),
    letesz(A, P, N, T, T1),
    kirak(Ak1, N, T1, X).
```

A 2-argumentumú változat megnézi, hogy hány alakzatot kapott (tehát milyen hosszú a tábla), és meghívja a 4-argumentumú testvérét. Ez a felhasználható alakzatokon (`Ak`) kívül még megkapja a tábla hosszát (`N`), a tábla jelenlegi állapotát (`T`), és ezáltal kiszámolja a kitöltött táblát (`X`).

Ehhez megkeresi az első lyukat, majd kiválaszt (töröl) egy alakzatot, azt leteszi úgy, hogy lefedje a lyukat, és rekurzívan folytatja a műveletet, amíg mindent le nem rak.

A tábla állapotát egy listában tároljuk, amelynek az elemei `h(X-Y,A)` alakban adják meg, hogy az `X-Y` pozíción az `A` alakzat egy darabja található. Az első lyuk megkeresése így nagyon egyszerű:

```
első_lyuk(N, T, X-Y) :-
    között(1, N, X), között(1, 5, Y),
    \+ tartalmaz(h(X-Y,_), T), !.
```

Ez deklaratív olvasatban azt mondja, hogy az `X-Y` koordináták a megfelelő keretek között vannak, és a tábla ezen a pozíción nem tartalmaz elemet. Mitől fogja ez az elsőt adni? Azért, mert a procedurális olvasatból tudjuk, hogy sorban fog próbálkozni, tehát először az `X = 1` eseteket próbálja végig különböző `Y`

értékekre, aztán az `X = 2`-t stb., és a végén levő vágásnak köszönhetően nem fog további lyukakat megadni akkor sem, ha a keresés visszalépne ide.

Már csak a `letesz(A, X-Y, N, T, T1)` szabály van hátra. Ez megpróbálja az `A` alakzatot lerakni úgy, hogy lefedje az `X-Y` pontot, ne menjen ki az $5 \times n$-es táblából, és ne takarjon olyan pozíciókat, amelyek szerepelnek `T`-ben. Ha sikerül, akkor az alakzat lehelyezésével kapott új tábla a `T1`.

```
letesz(A, X-Y, N, T, T1) :-
    alakzat(A, [_-Dy|Dk]),
    Y1 is Y - Dy, Y1 > 0,
    letesz(A, X-Y1, Dk, N, [h(X-Y,A)|T], T1).
```

Ez tehát az `A` alakzatnak kiválasztja egy forgatását, és megnézi az első pontjának az y-koordinátáját (`Dy`). Az `Y` koordinátánál ennyivel lejjebb kell rakni az alakzatot, hogy az első pont fedje a lyukat (hiszen az első pont az alakzat legbaloldalibb, és azon belül legalsó pontja). Ha ez a módosult `Y1` koordináta nem pozitív, akkor az alakzat kilóg.

A tényleges lerakási kísérletet a `letesz` 6-argumentumú változata végzi:

```
letesz(_, _, [], _, T, T).
letesz(A, X-Y, [Dx-Dy|Dk], N, T, T1) :-
    X1 is X + Dx, között(1, N, X1),
    Y1 is Y + Dy, között(1, 5, Y1),
    \+ tartalmaz(h(X1-Y1,_), T),
    letesz(A, X-Y, Dk, N, [h(X1-Y1,A)|T], T1).
```

Ez plusz argumentumként megkapja a kiválasztott forgatáshoz tartozó pontokat is (az első kivételével, amellyel a lyukat fedtük), és ezeken megy végig rekurzívan. Minden lépésben a (módosított) `X-Y` koordináta alapján kiszámolja, hogy hova kerül a pont, és ellenőrzi, hogy értelmes-e a koordináta és nincs-e már lefedve `T`-ben.

Kiíratás

A lényeggel ugyan már készen vagyunk, de az eredmény nehezen olvasható:

```
?- kirak([l,t,w,k,y,r,z,c,p], X).
  X = [h(9-5, c), h(9-4, c), h(9-3, c), h(8-5, c),
       h(8-3, c), h(9-2, p), h(9-1, p), h(8-2, p),
       h(8-1, p), h(7-1, p), h(8-4, z), h(7-4, z),
       h(7-3, z), h(7-2, z), h(6-2, z), h(7-5, r),
       h(6-5, r), h(6-4, r), h(6-3, r), h(5-4, r),
       h(5-5, w), h(4-5, w), h(4-4, w), h(3-4, w),
       h(3-3, w), h(6-1, y), h(5-2, y), h(5-1, y),
       h(4-1, y), h(3-1, y), h(5-3, k), h(4-3, k),
       h(4-2, k), h(3-2, k), h(2-2, k), h(3-5, t),
       h(2-5, t), h(2-4, t), h(2-3, t), h(1-5, t),
       h(2-1, l), h(1-4, l), h(1-3, l), h(1-2, l),
       h(1-1, l)]
```

Próbáljuk meg kiíratni egy kicsit „grafikusabb” formában! Az elsődleges szabályunk akkor ez lesz:

```
1 katamino(Ak) :-
2     hossz(Ak, N), kirak(Ak, X), kiír(N, X).
```

A `kiír(N, T)` pedig 5 sorba rendezve szépen kiírja az `N` hosszú `T` táblát. A kiíráshoz a `write(X)` kifejezést fogjuk használni, ami kiírja az `X` értékét, új sor nyitásához pedig a `nl`-t (*newline*, újsor).

```
kiír(N, T) :- kiír(N, 1-5, T).

kiír(_, _-0, _).
kiír(N, X-Y, T) :-
    Y =< 5, X > N, Y1 is Y - 1,
    nl, kiír(N, 1-Y1, T).
kiír(N, X-Y, T) :-
    Y =< 5, X =< N,
    tartalmaz(h(X-Y,A), T),
    write(A), write(' '),
    X1 is X + 1,
    kiír(N, X1-Y, T).
```

A 2-argumentumú változat csak átadja a feladatot a 3-argumentumúnak, ami megkapja az éppen kiírandó elem `X-Y` koordinátáját is. Ennek három szabálya van:

1. Ha már a 0. sort kéne kiírni, készen vagyunk.
2. Ha `X > N`, akkor vége az aktuális sornak, új sort kezdünk. Az `Y` koordináta felülről megy lefelé, mert a kiírás is felülről lefelé történik.
3. Egyébként megkeresi, hogy az `X-Y` pozíción melyik alakzat található, és kiírja, utána kiír még egy szóközt, és továbbmegy a következő `X` pozícióra.

Nézzük meg!

```
?- katamino([l,t,w,k,y,r,z,c,p]).
t t t w w r r c c
l t w w r r z z c
l t w k k r z c c
l k k k y z z p p
l l y y y y p p p
true
```

És ez éppen az a megoldás, ami a képen van! Erre persze elég jó esély volt, ugyanis a felhasználandó alakzatok listáját hozzávetőlegesen a képen látható megoldás sorrendjében adtuk meg. Egy másik permutáció más megoldást talál meg először:

```
?- katamino([t,w,l,k,r,z,y,p,c]).
w y y y y l l l l
w w y p p p r r l
t w w p p z z r r
t t t k k z c r c
t k k k z z c c c
true
```

41. Feladat. A Katamino szabályai szerint a kirakandó téglalap magassága mindig 5. Oldjuk fel ezt a korlátot! Írd át a programot úgy, hogy a `katamino` szabály paraméterben kapja meg a magasságot is, és ennek megfelelő megoldást keressen! A program vegye észre rögtön, ha a magasság nem illik a megkapott alakzatok számához (pl. 10 alakzat és 7-es magasság).

Keress megoldást a 3×20, 4×15, 5×12, 6×10 téglalapok kitöltésére! (Tipp: a szabályokban az eddigi `N` paramétert cseréld `N-M` párra.)

A teljes program

```
1  katamino(Ak) :-
2      hossz(Ak, N), kirak(Ak, X), kiír(N, X).
3
4  % Megoldó
5
6  kirak(Ak, X) :- hossz(Ak, N), kirak(Ak, N, [], X).
7
8  kirak([], _, T, T).
9  kirak(Ak, N, T, X) :-
10     első_lyuk(N, T, P),
11     töröl(A, Ak, Ak1),
12     letesz(A, P, N, T, T1),
13     kirak(Ak1, N, T1, X).
14
15 első_lyuk(N, T, X-Y) :-
16     között(1, N, X), között(1, 5, Y),
17     \+ tartalmaz(h(X-Y,_), T), !.
18
19 letesz(A, X-Y, N, T, T1) :-
20     alakzat(A, [_-Dy|Dk]),
21     Y1 is Y - Dy, Y1 > 0,
22     letesz(A, X-Y1, Dk, N, [h(X-Y,A)|T], T1).
23
```

```
24 letesz(_, _, [], _, T, T).
25 letesz(A, X-Y, [Dx-Dy|Dk], N, T, T1) :-
26     X1 is X + Dx, között(1, N, X1),
27     Y1 is Y + Dy, között(1, 5, Y1),
28     \+ tartalmaz(h(X1-Y1,_), T),
29     letesz(A, X-Y, Dk, N, [h(X1-Y1,A)|T], T1).
30
31 % Kiírás
32
33 kiír(N, T) :- kiír(N, 1-5, T).
34
35 kiír(_, _-0, _).
36 kiír(N, X-Y, T) :-
37     Y =< 5, X > N, Y1 is Y - 1,
38     nl, kiír(N, 1-Y1, T).
39 kiír(N, X-Y, T) :-
40     Y =< 5, X =< N,
41     tartalmaz(h(X-Y,A), T),
42     write(A), write(' '),
43     X1 is X + 1,
44     kiír(N, X1-Y, T).
45
46 % Segéd-szabályok
47
48 tartalmaz(X, [X|_]).
49 tartalmaz(X, [_|Maradék]) :- tartalmaz(X, Maradék).
50
51 töröl(X, [X|M], M).
52 töröl(X, [Y|M], [Y|M1]) :- töröl(X, M, M1).
```

```
53
54  hossz([], 0).
55  hossz([_|M], N) :- hossz(M, N1), N is 1 + N1.
56
57  között(N, M, N) :- N =< M.
58  között(N, M, X) :-
59      N < M, N1 is N + 1,
60      között(N1, M, X).
61
62  % Alakzatok
63
64  % kígyó (lila)
65  alakzat(k, [0-0,0-1,0-2,1-2,1-3]).
66  alakzat(k, [0-0,0-1,1-1,1-2,1-3]).
67  alakzat(k, [0-0,1-0,1-1,2-1,3-1]).
68  alakzat(k, [0-0,1-0,2-0,2-1,3-1]).
69  alakzat(k, [0-1,0-2,0-3,1-0,1-1]).
70  alakzat(k, [0-1,1-0,1-1,2-0,3-0]).
71  alakzat(k, [0-1,1-1,2-0,2-1,3-0]).
72  alakzat(k, [0-2,0-3,1-0,1-1,1-2]).
73  % P-betű (rózsaszín)
74  alakzat(p, [0-0,0-1,0-2,1-0,1-1]).
75  alakzat(p, [0-0,0-1,0-2,1-1,1-2]).
76  alakzat(p, [0-0,0-1,1-0,1-1,1-2]).
77  alakzat(p, [0-0,0-1,1-0,1-1,2-0]).
78  alakzat(p, [0-0,0-1,1-0,1-1,2-1]).
79  alakzat(p, [0-0,1-0,1-1,2-0,2-1]).
80  alakzat(p, [0-1,0-2,1-0,1-1,1-2]).
81  alakzat(p, [0-1,1-0,1-1,2-0,2-1]).
```

```
% C-betű (sárga)
alakzat(c, [0-0,0-1,0-2,1-0,1-2]).
alakzat(c, [0-0,0-1,1-0,2-0,2-1]).
alakzat(c, [0-0,0-1,1-1,2-0,2-1]).
alakzat(c, [0-0,0-2,1-0,1-1,1-2]).
% W-betű (világoszöld)
alakzat(w, [0-0,0-1,1-1,1-2,2-2]).
alakzat(w, [0-0,1-0,1-1,2-1,2-2]).
alakzat(w, [0-1,0-2,1-0,1-1,2-0]).
alakzat(w, [0-2,1-1,1-2,2-0,2-1]).
% L-betű (narancssárga)
alakzat(l, [0-0,0-1,0-2,0-3,1-0]).
alakzat(l, [0-0,0-1,0-2,0-3,1-3]).
alakzat(l, [0-0,0-1,1-0,2-0,3-0]).
alakzat(l, [0-0,0-1,1-1,2-1,3-1]).
alakzat(l, [0-0,1-0,1-1,1-2,1-3]).
alakzat(l, [0-0,1-0,2-0,3-0,3-1]).
alakzat(l, [0-1,1-1,2-1,3-0,3-1]).
alakzat(l, [0-3,1-0,1-1,1-2,1-3]).
% Y-betű vagy tengeralattjáró (barna)
alakzat(y, [0-0,0-1,0-2,0-3,1-1]).
alakzat(y, [0-0,0-1,0-2,0-3,1-2]).
alakzat(y, [0-0,1-0,1-1,2-0,3-0]).
alakzat(y, [0-0,1-0,2-0,2-1,3-0]).
alakzat(y, [0-1,1-0,1-1,1-2,1-3]).
alakzat(y, [0-1,1-0,1-1,2-1,3-1]).
alakzat(y, [0-1,1-1,2-0,2-1,3-1]).
alakzat(y, [0-2,1-0,1-1,1-2,1-3]).
% I-betű vagy egyenes (kék)
```

```
111 alakzat(i, [0-0,0-1,0-2,0-3,0-4]).
112 alakzat(i, [0-0,1-0,2-0,3-0,4-0]).
113 % r-betű (szürke)
114 alakzat(r, [0-0,0-1,1-1,1-2,2-1]).
115 alakzat(r, [0-0,1-0,1-1,1-2,2-1]).
116 alakzat(r, [0-1,0-2,1-0,1-1,2-1]).
117 alakzat(r, [0-1,1-0,1-1,1-2,2-0]).
118 alakzat(r, [0-1,1-0,1-1,1-2,2-2]).
119 alakzat(r, [0-1,1-0,1-1,2-1,2-2]).
120 alakzat(r, [0-1,1-1,1-2,2-0,2-1]).
121 alakzat(r, [0-2,1-0,1-1,1-2,2-1]).
122 % V-betű vagy sarok (kék)
123 alakzat(v, [0-0,0-1,0-2,1-0,2-0]).
124 alakzat(v, [0-0,0-1,0-2,1-2,2-2]).
125 alakzat(v, [0-0,1-0,2-0,2-1,2-2]).
126 alakzat(v, [0-2,1-2,2-0,2-1,2-2]).
127 % Z-betű vagy S-betű (kék)
128 alakzat(z, [0-0,0-1,1-1,2-1,2-2]).
129 alakzat(z, [0-0,1-0,1-1,1-2,2-2]).
130 alakzat(z, [0-1,0-2,1-1,2-0,2-1]).
131 alakzat(z, [0-2,1-0,1-1,1-2,2-0]).
132 % pluszjel (piros)
133 alakzat(+, [0-1,1-0,1-1,1-2,2-1]).
134 % T-betű (zöld)
135 alakzat(t, [0-0,0-1,0-2,1-1,2-1]).
136 alakzat(t, [0-0,1-0,1-1,1-2,2-0]).
137 alakzat(t, [0-1,1-1,2-0,2-1,2-2]).
138 alakzat(t, [0-2,1-0,1-1,1-2,2-2]).
```

6. lecke

Haladó eszközök

Ebben a leckében meg fogunk vizsgálni néhány olyan technikát, amelyekkel a programok hatékonyabbá vagy egyszerűbbé tehetőek, illetve szó lesz néhány fontos beépített szabályról is.

6.1. Különbség-listák

Listákat lehet két lista különbségeként ábrázolni, így pl. leírható az `[1,2,3]` lista úgy, mint az `[1,2,3,8]` és `[8]` listák különbsége, vagy az `[1,2,3]` és `[]` listáké, vagy általánosan az `[1,2,3|M]` és `M` különbségeként. A két tagot összekapcsolhatjuk pl. a `-` operátorral: `[1,2,3|M]-M`.

Ennek az ábrázolásnak nagy előnye, hogy közvetlenül tudunk hivatkozni a lista végére, ezért bizonyos műveleteket sokkal ha-

tékonyabban meg lehet valósítani a segítségükkel, mint ha csak sima listákat használnánk.

Egy egyszerű példa erre a hozzáfűzés:

```
1 hozzáfűz_kl(X-Y, Y-Z, X-Z).
```

Ehhez persze az kell, hogy a szabály argumentumként különbséglistákat kapjon:

```
?- hozzáfűz_kl([a,b,c|M]-M, [1,2]-[], L-[]).
M = [1, 2],
L = [a, b, c, 1, 2]
```

Mi is történik itt pontosan? Legjobban talán akkor látszik, ha a 3. leckében a listák bevezetésénél használt prefix jelöléssel írjuk fel:

```
?- hozzáfűz_kl(lista(a, lista(b, lista(c, M)))-M,
               lista(1, lista(2, vége))-vége,
               L-vége).
```

A szabály alapján az `M` és `lista(1, lista(2, vége))` egyesül, tehát a `hozzáfűz_kl` szabályban szereplő `X` értéke

```
lista(a,lista(b,lista(c,lista(1,lista(2,vége)))))
```

lesz, ami éppen a két lista összefűzése. Ugyanez többszörös összefűzésre is használható, pl.

```
?- hozzáfűz_kl([a,b|M1]-M1, [c,d|M2]-M2, L-M),
   hozzáfűz_kl(L-M, [e,f|M3]-M3, X-[]).
L = X, X = [a, b, c, d, e, f],
```

```
M = M2, M2 = [e, f],
M1 = [c, d, e, f],
M3 = []
```

A hozzáfűzésnek ezt a módját – mivel annyira egyszerű – nem szokás így külön szabályként felírni, hanem általában a programba közvetlenül építik be, ahogy azt mindjárt látni fogjuk a gyorsrendezésnél.

Gyorsrendezés

Az előző leckében látott gyorsrendezés is sokkal hatékonyabbá tehető ezzel a technikával. Eredetileg így nézett ki:

```
1 rendez2([], []).
2 rendez2([X|M], Y) :-
3     szétoszt(X, M, Kicsi, Nagy),
4     rendez2(Kicsi, K),
5     rendez2(Nagy, N),
6     hozzáfűz(K, [X|N], Y).
```

Írjuk át különbség-listák használatával!

```
1 rendez2(X, Y) :- rendez2_kl(X, Y-[]).
2
3 rendez2_kl([], X-X).
4 rendez2_kl([X|M], Y-Z) :-
5     szétoszt(X, M, Kicsi, Nagy),
6     rendez2_kl(Kicsi, Y-[X|Y1]),
7     rendez2_kl(Nagy, Y1-Z).
```

Látszik, hogy itt már nincsen szükség hozzáfűzésre, a két rekurzív hívás eleve „jó helyre” készíti a megoldásait. Ezért a varázslatért a változók egyesítése a felelős.

Lista megfordítása

A `fordított` szabállyal már találkoztunk a 3. lecke egyik feladatában. Egy lehetséges megoldás a következő:

```
1 fordított1([], []).
2 fordított1([X|M], Y) :-
3     fordított1(M, M1), hozzáfűz(M1, [X], Y).
```

Ez így nem túl hatékony. Ezt is megpróbálhatjuk vég-rekurziós formára hozni egy plusz (ún. *akkumulátor*) argumentum segítségével:

```
1 fordított2(X, Y) :- fordított2(X, [], Y).
2
3 fordított2([], Y, Y).
4 fordított2([X|M], F, Y) :- fordított2(M, [X|F], Y).
```

Egy másik lehetőség, hogy különbség-listákat használunk:

```
1 fordított3(X, Y) :- fordított_kl(X, Y-[]).
2
3 fordított_kl([], X-X).
4 fordított_kl([X|M], Y-Z) :- fordított_kl(M, Y-[X|Z]).
```

Ha most összehasonlítjuk a két megoldást, látszik, hogy a kettő lényegében megegyezik, csak a vég-rekurziós változat harmadik és második paraméterét összevontuk egy különbség-listává.

42. Feladat. Írd át a 3. lecke feladatai közt szereplő `lapít` szabályt hatékonyabbra különbség-listák használatával!

43. Feladat. Oldd meg Dijkstra *holland zászló* problémáját: piros, fehér és kék színű elemek listáját rendezzétek át úgy, hogy a piros elemek után jöjjenek a fehérek, és végül a kékek, de ezen belül a sorrendjük ne változzon! Például:

```
?- holland([piros(alma),fehér(fal),kék(tenger),
            piros(paprika),fehér(holló)], X).
X = [piros(alma), piros(paprika), fehér(fal),
     fehér(holló), kék(tenger)]
```

A megoldáshoz használj különbség-listákat!

Kitérő: *Dijkstra*

Edsger W. Dijkstra (1930-2002) nevét legtöbben a *Dijkstra-algoritmus* kapcsán ismerik. Ez egy úthálózatban (súlyozott gráfban) megkeresi két pont között a legrövidebb utat.

6.2. Kifejezések vizsgálata

Egy kifejezés típusának megvizsgálására a következő beépített szabályok adottak:

– `var(X)`: `X` változó (és nincs értéke)

- `nonvar(X)`: `X` nem változó vagy van értéke
- `atom(X)`: `X` atom
- `number(X)`: `X` szám
  - `integer(X)`: `X` egész szám
  - `float(X)`: `X` valós szám
- `atomic(X)`: `X` atom vagy szám
- `compound(X)`: `X` összetett struktúra

Egy struktúra ezen kívül szétszedhető az elemeinek listájára (fej + argumentumok), illetve visszaépíthető ezekből az `=..` segítségével:

```
?- f(a,b,c) =.. X.
X = [f, a, b, c]
?- X =.. [f, a, b, c].
X = f(a,b,c)
```

A funktort és aritást, illetve az egyes argumentumokat külön is le lehet kérdezni a `functor` ill. `arg` használatával:

```
?- functor(f(a,b,c), F, N).
F = f,
N = 3
?- arg(1, f(a,b,c), X).
X = a
?- arg(2, f(a,b,c), X).
X = b
```

Ezeknek a segítségével a struktúrákat tudjuk fix hosszú *tömbökként* kezelni, tehát olyan adattárolókként, amelyeknek tetszőleges eleme hatékonyan elérhető (ellentétben a listákkal, ahol egy elem eléréséhez előbb végig kell mennünk az összes előtte levőn). Például a

```
?- functor(T, t, 10), arg(5, T, 42)
```

hatására a `T` egy olyan 10-elemű tömb lesz, amelynek az 5. eleme a 42.

Csere

Hogyan lehetne ezek segítségével megoldani a következő feladatot: egy kifejezésben egy másik (al)kifejezés minden előfordulását le akarjuk cserélni valami másra. Például:

```
?- lecserél(sin(x), t, 2*sin(x)*f(sin(x)), F).
F = 2*t*f(t)
```

Itt az „előfordulás” egyesíthetőséget jelent, tehát

```
?- lecserél(a+b, v, f(a,A+B), F).
A = a,
B = b,
F = f(a, v)
```

Három eset van:

1. A kifejezés és a lecserélendő alkifejezés megegyezik – az egészet cseréljük.

2. A kifejezés atom/szám – nincs mit csinálni.

3. A kifejezés egy struktúra; az argumentumaira kell elvégezni a cserét.

Ez alapján a három szabály:

```
lecserél(X, Y, X, Y) :- !.
lecserél(_, _, Z, Z) :- atomic(Z), !.
lecserél(X, Y, Z, Z1) :-
    Z =.. [F|Arg],
    mindent_lecserél(X, Y, Arg, Arg1),
    Z1 =.. [F|Arg1].
```

A `mindent_lecserél` szabály egy lista minden elemére elvégzi a cserét:

```
mindent_lecserél(_, _, [], []).
mindent_lecserél(X, Y, [Z|M], [Z1|M1]) :-
    lecserél(X, Y, Z, Z1),
    mindent_lecserél(X, Y, M, M1).
```

Ennek segítségével készíthetünk függvénykiértékelőt:

```
?- kiértékel(x*sin((x+y)/2), [x=1,y=2.14], X).
X = 0.9999996829318346
```

A második argumentumban `szimbólum = szám` alakban vannak megadva a helyettesítési értékek.

```
kiértékel(K, L, X) :-
    behelyettesít(K, L, K1), X is K1.
```

```
3
4  behelyettesít(K, [], K).
5  behelyettesít(K, [A=N|M], K2) :-
6      lecserél(A, N, K, K1),
7      behelyettesít(K1, M, K2).
```

44. Feladat. Írj szabályt, ami egy összeadásokat tartalmazó kifejezést egyszerűsít úgy, hogy az ismeretleneket (ha lehet) összevonja és előre rakja, a többi összeadást pedig elvégzi! (Vigyázat, ez egy elég komplex feladat!)

```
?- egyszerűsít(1+1+a, E).
E = a+2
?- egyszerűsít(1+a+4+2+b+c, E).
E = a+b+c+7
?- egyszerűsít(3+x+x, E).
E = 2*x+3
```

6.3. Magasabb rendű szabályok

Vannak olyan szabályok, amelyek argumentumként egy másik szabályt (célt, kérdést) várnak. Ilyen volt például a tagadás, de van még néhány másik is.

Listakezelés

Egy nagyon hasznos szabály a `maplist`, ami egy lista összes elemére megnézi, hogy teljesít-e egy adott szabályt, pl.

```
?- maplist(number, [3,4,5]).
true
?- maplist(number, [3,a,5]).
false
```

A szabály lehet többargumentumú is, ilyenkor több listát kell megadni, egyet minden argumentumhoz. A `mindent_lecserél` például megfogalmazható így:

```
1 mindent_lecserél(X, Y, Z, Z1) :-
2     maplist(lecserél(X, Y), Z, Z1).
```

Itt a `lecserél` első két argumentumát előre megadtuk, a maradék kettőt a `Z` és `Z1` listákból veszi ki.

Összes megoldás

Gyakran előfordul, hogy az összes megoldásra kíváncsiak vagyunk. Ilyenkor ezeket egy listában le lehet kérni a `bagof` („zsákja”), `setof` („halmaza”) vagy `findall` („találd meg az összeset”) segítségével. Nézzük meg sorban őket!

A `bagof(X, P, L)` megkeresi az összes olyan `X`-et, amire `P` igaz, és `L` ezeknek a listája. Például:

```
1 életkor(lica, 11).
2 életkor(mimi, 10).
3 életkor(dusa, 5).
4 életkor(zsófi, 5).
5 életkor(juli, 2).
```

```
?- bagof(X, életkor(X, 5), L).
L = [dusa, zsófi]
?- bagof(X, életkor(X, _), L).
L = [juli]
```

További megoldásokként megkapjuk életkorok szerint csoportosítva a többi gyereket is. Ha azt szeretnénk, hogy egyszerre az összes gyereket megkapjuk, akinek ismert az életkora (és nem csak azokat, akiknek azonos), akkor erre egy speciális jelölést kell használni:

```
?- bagof(X, N^életkor(X, N), L).
L = [lica, mimi, dusa, zsófi, juli]
```

Az `N^` itt azt jelenti, hogy „van olyan `N`, amire igaz, hogy ...”.

A `setof` nagyon hasonló ehhez, de az azonos elemekből csak egyet tart meg, például:

```
?- bagof(N, X^életkor(X, N), L).
L = [11, 10, 5, 5, 2]
?- setof(N, X^életkor(X, N), L).
L = [2, 5, 10, 11]
```

Végül a `findall` olyan, mint a `bagof`, amikor a kifejezés egy változójának értéke sincs lekötve, tehát mintha minden (nem keresett) `V` változóhoz oda lenne írva a `V^`. Például:

```
?- bagof(X, N^életkor(X, N), L).
L = [lica, mimi, dusa, zsófi, juli]
?- findall(X, életkor(X, _), L).
L = [lica, mimi, dusa, zsófi, juli]
```

45. Feladat. Írj szabályt, ami a `bagof` segítségével megkeresi egy halmaz (lista) összes részhalmazát!

6.4. Dinamikus szabályok

Szabályokat a program futása közben automatikusan is hozzá lehet adni az adatbázishoz, illetve ki lehet venni belőle. Ha a szabály már létezik, akkor ehhez az kell, hogy *dinamikusnak* legyen beállítva, pl.:

```
:- dynamic foo/2. % 2 aritású
```

Ezután az alábbi módon lehet a `foo` szabályait módosítani:

- `asserta(foo([], _))`: az adatbázis elejére teszi a `foo([], _).` tényt.
- `assertz(foo(X, Y):- X > Y)`: az adatbázis végére teszi a `foo(X, Y):- X > Y.` szabályt.
- `retract(foo([], _))`: törli a `foo([], _).` tényt.
- `retractall(foo(_,_))`: törli az összes szabályt, aminek a feje egyesíthető a `foo(_,_)`-val.

Ezek segítségével például definiálhatjuk magunk is a `findall` szabályt:

```
összes(X, Cél, L) :-
    Cél, assertz(tároló(X)), fail
  ; assertz(tároló(nincs_több)), összegyűjt(L).
```

```
4
5  összegyűjt(L) :-
6      retract(tároló(X)), !,
7      ( X = nincs_több, !, L = []
8      ; L = [X|M], összegyűjt(M)
9      ).
```

```
?- összes(X, életkor(X, _), L).
L = [lica, mimi, dusa, zsófi, juli]
```

Ez a megoldás feltételezi, hogy a `tároló` szabály még nem létezett, és hogy a keresett értékek közt nem szerepelhet a `nincs_több` atom. Ezt elkerülendő, ezeket a `$` prefix operátorral szokás megkülönböztetni, tehát `tároló(X)` helyett `$tároló(X)` és `nincs_több` helyett `$nincs_több` (vagy a zárójelet kiírva `$(tároló(X))` és `$(nincs_több)`).

Az önmagát módosító programok megértése nehéz, ezért az ilyen jellegű technikákat csak jól elkülönített programrészekben ajánlott alkalmazni.

Polipos játék

A játéktáblán sorokba rendezve kidudorodó gombok találhatóak. A játékosok felváltva lépnek, és egy lépés során kiválasztanak egy sort, és abban a sorban tetszőleges számú (de legalább egy) gombot benyomnak. Az veszt, aki az utolsó gombot nyomja be.

Írjunk programot, ami megtalálja a legjobb lépést! Ehhez először írjuk le a táblát:

```
polip([3,5,2,4,5,5,3]).
```

A lista minden eleme az adott sorban levő gombok számának felel meg. Adjunk hozzá logikát is:

```
nyertes([]) :- !.
nyertes(L) :- nyom(L, _, L1), vesztes(L1), !.
```

Akkor nyer az aktuális játékos, ha (i) már nincsen benyomható gomb, vagy (ii) ha tud vesztes állást előidézni.

A `nyom(L, I-N, L1)` reláció akkor teljesül, ha az `L` állásban az `I`-edik sorban `N` gombot benyomva az `L1` állás jön létre. A sorok számlálásához felveszünk egy második paramétert, ami 1-ről indul:

```
nyom(L, V, L1) :- nyom(L, 1, V, L1).
```

A lépéskor három lehetőség van:

1. Benyomjuk az egész első sort.
2. Benyomunk valamennyit az első sorból (de nem mindet).
3. Másik sorban nyomunk.

Ennek megfelel az alábbi három szabály:

```
nyom([X|M], I, I-X, M).
nyom([X|M], I, I-Y, [Y1|M]) :-
    X1 is X - 1, között(1, X1, Y), Y1 is X1 - Y.
nyom([X|M], I, V, [X|M1]) :-
    I1 is I + 1, nyom(M, I1, V, M1).
```

A vesztes állás az egyszerűen a nem nyertes állás:

```
vesztes(L) :- \+ nyertes(L).
```

Végül pedig nyerő lépés az, ami után vesztes állás jön létre:

```
nyerő_lépés(L, V) :- nyom(L, V, X), vesztes(X), !.
```

Például a kezdőállásból egy nyerő lépést így kaphatunk meg:

```
?- polip(P), nyerő_lépés(P, X).
P = [3, 5, 2, 4, 5, 5, 3],
X = 1-3. % Az első sorban nyomjuk be az összeset (3)
```

De mi köze mindennek a dinamikus szabályokhoz? Ha valaki kipróbálja a fenti programot, bizony elég sokat kell várnia, mert nagyon lassú. Fel lehetne gyorsítani azzal, ha a **vesztes** már kiszámolt értékeit (tehát, hogy mely állások vesztesek) eltárolnánk, és amikor újra szükség van rájuk, akkor egyszerűen csak elő tudnánk húzni az eredményt a tarsolyunkból. Ezt a módszert – a korábban kiszámolt eredmények eltárolását – *memoizálás*nak vagy *táblázás*nak szokás nevezni.

A legtöbb Prolog rendszernek van erre valamilyen beépített módszere, pl. SWI-Prologban mindössze annyit kell írni, hogy

```
:- table vesztes/1.
```

... de ez nem szabványos. A dinamikus szabályok segítségével viszont ez nagyon könnyen megoldható:

```
:- dynamic(vesztes/1).
vesztes(L) :-
```

```
3       \+ nyertes(L), !,
4       asserta((vesztes(L) :- !)).
5   vesztes(L) :-
6       asserta((vesztes(L) :- !, fail)),
7       fail.
```

Ha egy állásról kiderül, hogy vesztes, akkor ezt, mint tényt, elrakjuk. Ha meg az derül ki, hogy nem, akkor egy olyan szabályt rakunk el, ami ezt biztosítja; ennek az utóbbi szabálynak is `fail` kell a végére, különben a `vesztes(L)` elsőre mindig igaz lenne.

Így a program már nagyon gyors. Ez természetesen egy kompromisszum: a sebességért cserébe tárhellyel fizetünk. Ha sokkal több megjegyzendő adatunk van, akkor a program könnyen kifuthat a memóriából.

6.5. Vezérlés

A program folyásának vezérlésére már láttunk néhány módszert, mint a vágás (`!`) vagy a mindig igaz ill. hamis célok (`true`, `false` ill. `fail`). Itt van néhány további:

1. Ha egy szabály több megoldást is vissza tud adni, de mi csak az elsőt szeretnénk, a vágással le tudjuk tiltani a továbbiakat. Ez elég gyakori ahhoz, hogy van rá egy beépített szabály, a `once` („egyszer”):

```
1   once(P) :- P, !.
```

2. Amikor egy `P` változót célként használunk, mint a `once` vagy a tagadás definíciójában, akkor valójában a háttérben a `call(P)` („hív”) hívódik meg; a szabvány szerint ezt ki is kéne írni, de a legtöbb implementációban elhagyható. Ennek ellenére érdemes lehet kiírni, hogy egyértelműbb legyen, mi történik. Amikor a `call`-nak több argumentuma van, ezeket a célhoz kapcsolandó további paraméterekként értelmezi, tehát pl.:

```
?- P = hozzáfűz([a,b]),
   call(P, [c,d], X),
   call(P, [x,y], Y).
P = hozzáfűz([a, b]),
X = [a, b, c, d],
Y = [a, b, x, y].
```

3. A `(P -> Q; R)` jelentése: ha `P`, akkor `Q`, különben `R`. Tehát pl. az alábbi kettő ekvivalens:

```
1 implikáció(X) :- foo(X) -> bar(X); baz(X).
```

és

```
1 implikáció(X) :- foo(X), !, bar(X).
2 implikáció(X) :- baz(X).
```

4. A felhasználóval való kommunikációhoz gyakran van szükség végtelen ciklusra, ezt segíti a `repeat` („ismétel”), amit így lehet definiálni:

```
repeat.
repeat :- repeat.
```

Ez tehát mindig igaz, akárcsak a `true`, de ezt az igaz értéket végtelenszer generálja. Egy példa a használatára az alábbi program, ami a `read` segítségével kér be a felhasználótól számokat, és kiírja a négyzetüket, egészen amíg `stop`-ot nem kap:

```
négyzetes :-
    repeat, read(X),
    ( X = stop, !
    ; Y is X * X, write(Y), fail
    ).
```

6.6. Projekt: mankala

Készítsünk mesterséges intelligenciát a *kalah* mankala-játékhoz!

A játéknak számos variánsa létezik, itt most egy egyszerű változatot fogunk megvalósítani. A táblán 12 kisebb és 2 nagyobb lyuk található; az egyes játékosokhoz a hozzájuk közelebb levő 6 lyuk és a jobb kéz felőli nagyobb lyuk (a *kalah*) tartozik. Kezdetben minden (kis) lyukban 6 kő van, bár szokás 4–4 kővel is játszani.

A játékosok felváltva lépnek. Egy lépés abból áll, hogy választanak egy saját lyukat, amiben van kő, és a benne levő köveket óramutató járásával ellenkező irányban egyesével beleszórják a következő lyukakba – amikor elfogynak a sajátok, akkor egy kerül a saját kalahba, utána az ellenfél lyukaiba, és ha azok elfogytak, akkor megint a sajátokba (az ellenfél kalahját ki kell hagyni).

Ha az utolsó kő a (saját) kalahba esik, akkor még egyszer lehet lépni (ez bárhányszor ismételhető). Ha az utolsó kő egy olyan saját lyukba kerül, ami előtte üres volt, és a szemben levő lyukban van kő, akkor a szóró játékos mindkét lyuk tartalmát megkapja és a kalahjába teszi.

A játékot az nyeri meg, aki megszerzi a köveknek több, mint a felét. A játéknak akkor is vége van, ha egy lépés végeztével az egyik térfél teljesen kiürül – ilyenkor a másik játékos megkapja a saját oldalán levő köveket.

Keretprogram

Ez már egy elég komplex program lesz, amit apránként fogunk felépíteni felülről lefelé, tehát először egy magas szinten fogalmazzuk meg, és utána kitöltjük a részleteket.

A Kalah az absztrakt táblajátékok általános sémáját követi;

erre írhatunk egy olyan keretprogramot, ami később akár más hasonló jellegű játékokra is használható lesz:

```
kalah :-
    alapbeállítás(Állás, Játékos),
    kirajzol(Állás, Játékos),
    játék(Állás, Játékos).

játék(Állás, Játékos) :-
    játék_vége(Állás, Játékos, Eredmény), !,
    bejelent(Eredmény).
játék(Állás, Játékos) :-
    lépést_választ(Állás, Játékos, Lépés),
    lép(Lépés, Állás, Állás1),
    következő_játékos(Játékos, Játékos1), !,
    kirajzol(Állás1, Játékos1),
    játék(Állás1, Játékos1).
```

Az `alapbeállítás` felállítja a tábla kezdő állapotát, és meghatározza a kezdő játékost. Ezt az állást kirajzoljuk, és utána kezdődik a tényleges `játék/2`. Ez először ellenőrzi, hogy vége van-e a játéknak, és ha igen, közli az eredményt. Egyébként a soron következő játékos választ egy lépést, ezt a lépést le is játsza, majd a következő játékos számára kirajzolja a (módosult) állást és a játék megy tovább.

A két játékosunk lehet az `ember` és a `számítógép`, és ezek váltakoznak, tehát

```
következő_játékos(ember, számítógép).
következő_játékos(számítógép, ember).
```

Az eredmény lehet egyszerűen a nyertes megnevezése, vagy a `döntetlen`:

```
bejelent(ember) :- kiír('Nyertél, gratulálok!').
bejelent(számítógép) :- kiír('Nyertem.').
bejelent(döntetlen) :- kiír('Döntetlen lett!').
```

Itt a `kiír` a `write` olyan változata, ami utána még egy újsort is kiír (`nl`):

```
kiír(X) :- write(X), nl.
```

Reprezentáció

A legfontosabb feladat itt is, mint mindig, a *reprezentáció* (adatábrázolás) ügyes megválasztása. Hogyan érdemes eltárolnunk az állást? A cél, hogy később kényelmesen le tudjuk írni a lépéseket.

Számozzuk be a lyukakat mindkét játékosnak balról jobbra 1–6-ig. A lépés tehát mindkét játékos számára egy 1 és 6 közötti szám lesz, vagyis pontosabban ezeknek egy listája, ugyanis ha az utolsó kő a kalahba kerül, akkor a játékos újra léphet, és így egy lépést a kiválasztott lyukak listájával lehet definiálni.

Az állást tehát a 2×6 lyukban, valamint a kalahokban levő kövek számával tudjuk leírni. A belső ábrázolásunk `tábla(La, Na,Lb,Nb)` alakú lesz, ahol `La` és `Lb` számok 6-elemű listája (a lyukakban levő kövek), `Na` és `Nb` sima számok (a kalahokban levő kövek). Az `a`-végűek az éppen soron következő játékoshoz tartozó adatok, a `b`-végűek az ellenfélhez tartozóak.

A játékot kezdje mindig az emberi játékos:

```
alapbeállítás(Állás, ember) :-
    kövek_száma(K),
    Állás = tábla([K,K,K,K,K,K],0,
                  [K,K,K,K,K,K],0).

kövek_száma(6).
```

A kövek számát érdemes a fent látható módon kivenni külön ténybe, hogy később kényelmesen változtatható legyen.

A Kalah szabályai

Mikor van vége a játéknak? Itt azt az egyszerűsítést alkalmazhatjuk, hogy ha az egyik térfél a lépés végére kiürül, akkor a másik térfélen levő kövek – még a lépés részeként – bekerülnek a megfelelő kalahba. Elég tehát azt a feltételt megnéznünk, hogy valamelyik játékos elvitte már a köveknek több, mint a felét.

```
játék_vége(tábla(_,N,_,N), _, döntetlen) :-
    kövek_száma(K), N =:= 6 * K, !.
játék_vége(tábla(_,N1,_,_), Játékos, Játékos) :-
    kövek_száma(K), N1 > 6 * K, !.
játék_vége(tábla(_,_,_,N2), Játékos, Másik) :-
    kövek_száma(K), N2 > 6 * K,
    következő_játékos(Játékos, Másik).
```

Következik a lépések leírása. Ahogy láttuk, a lépés lyukak listájával van megadva. Ha a lista végére értünk, akkor a másik játékos következik, ezért meg kell „fordítani” a táblát:

```
lép([], Állás, Állás1) :- megfordít(Állás, Állás1).

megfordít(tábla(La,Na,Lb,Nb), tábla(Lb,Nb,La,Na)).
```

Egyébként pedig körbeszórjuk („vetjük”) a köveket:

```
lép([L|M], Állás, Állás2) :-
    kövek(L, Állás, K),
    vet(K, L, Állás, Állás1),
    lép(M, Állás1, Állás2).
```

Itt a `kövek` megadja, hogy egy adott lyukban hány kő van:

```
kövek(I, tábla(L,_,_,_), K) :-
    n_edik(I, L, K), K > 0.

n_edik(N, [_|M], X) :-
    N > 1, !, N1 is N - 1,
    n_edik(N1, M, X).
n_edik(1, [X|_], X).
```

A vetés a játék lelke, és egyben a legbonyolultabb része. Ezt két részre szedjük, a saját oldalon és az ellenfél oldalon való vetésre (utóbbi szükség szerint újra hivatkozik majd a `vet`-re):

```
vet(Kövek, Luk, Állás, Állás2) :-
    vet_saját(Kövek, Luk, Állás, Állás1, Kövek1),
    vet_ellenfél(Kövek1, Állás1, Állás2).
```

Ez tehát azt mondja, hogy a `Luk` lyukból indulva vetünk `Kövek` darab követ, és így jutunk a kezdeti `Állás`-ból a végső

`Állás2`-be. A saját oldali vetés végeztekor egy közbülső `Állás1`-be jutunk, ahol még további `Kövek1` darab követ kell vetnünk az ellenfél oldalára.

Ha a kövek száma nagyobb, mint 7 − `Luk`, akkor átjutunk az ellenfél oldalára (tehát az 1-es lyukból legalább 7 kő kell, a 2-esből 6, ..., a 6-osból legalább 2, hiszen az első a kalahba kerül):

```
vet_saját(Kövek, Luk, tábla(La,Na,Lb,Nb),
          tábla(La1,Na1,Lb,Nb), Kövek1) :-
    Kövek > 7 - Luk, !, % átmegy az ellenfélhez
    felvesz_és_szór(Luk, Kövek, La, La1),
    Na1 is Na + 1, Kövek1 is Kövek + Luk - 7.
```

A lényeget a `felvesz_és_szór` végzi, amit kicsit későbbre hagyunk. Ha nem jutunk át a másik oldalra, akkor a megmaradó kövek száma 0:

```
vet_saját(Kövek, Luk,
          tábla(La,Na,Lb,Nb), Állás, 0) :-
    Kövek =< 7 - Luk,
    felvesz_és_szór(Luk, Kövek, La, La1),
    elfogás(Luk, Kövek, La1, La2, Lb, Lb1, N),
    raktározás(N, Kövek, Luk, Na, Na1),
    vetés_vége(tábla(La2,Na1,Lb1,Nb), Állás).
```

Az `elfogás` megnézi, hogy hány követ ejtettünk foglyul (`N`), és hogy ennek hatására hogyan változik a tábla (új `La2` és `Lb1` értékek). A `raktározás` frissíti a kalah értékét (`Na1`), egyrészt az elfogott kövek, másrészt az aktuális vetés során oda jutó kő figyelembe vételével. Végül a `vetés_vége` ellenőrzi, hogy kiürült-e az

egyik térfél, és ha igen, akkor a fennmaradó köveket a megfelelő kalahba teszi.

Nézzük meg ezeket sorban!

```
elfogás(Luk, Kövek, La, La1, Lb, Lb1, N) :-
    Vége is Luk + Kövek,
    n_edik(Vége, La, 1),
    Szemben is 7 - Vége,
    n_edik(Szemben, Lb, K),
    K > 0, !, % üresbe érkeztünk és van szemben kő
    n_csere(Vége, La, 0, La1),
    n_csere(Szemben, Lb, 0, Lb1),
    N is K + 1.
elfogás(_, _, La, La, Lb, Lb, 0) :- !.

n_csere(1, [_|M], Y, [Y|M]) :- !.
n_csere(N, [X|M], Y, [X|M1]) :-
    N > 1, !, N1 is N - 1,
    n_csere(N1, M, Y, M1).
```

A `Vége` adja meg, hogy melyik lyukba kerül az utolsó kő. Ha ebben 1 kő van, tehát a vetés előtt üres volt, akkor megvizsgáljuk, hogy szemben van-e kő (a szemben levő lyukak számozásának iránya fordított, ezért a megfelelő index a $7 -$ `Vége` lesz). Ha ez is teljesül, akkor kinullázzuk ezt a két lyukat az `n_csere(I, L, X, L1)` használatával, ami az `L` lista `I`-edik elemét `X`-re állítja. Végül a kapott kövek száma a szemben levő kövek száma + 1. Ellenkező esetben a tábla változatlan marad és a kapott kövek száma 0 lesz.

Figyeljük meg, hogy itt az aránylag bonyolult feltételt nem ismételjük meg a második szabályban, hanem a procedurális olvasatra hagyatkozunk: az első szabályban levő vágás miatt a másodikba csak akkor jutunk, ha a feltétel nem teljesül. Ez a deklaratív olvasatot elrontja, de megengedhető, mivel tudjuk, hogy csak olyan környezetben használjuk, ahol az `La1`, `Lb1` és `N` változóknak nincsen értéke. A hatékonyság érdekében sok szabály használja itt ezt a módszert.

A `raktározás` elég magától értetődő. Ha az elfogott kövek száma 0, akkor ellenőrzi, hogy az utolsó kő a kalahba került-e, egyébként csak hozzáadja az eddigiekhez az elfogott köveket:

```
raktározás(0, Kövek, Luk, Na, Na) :-
    Kövek < 7 - Luk, !.
raktározás(0, Kövek, Luk, Na, Na1) :-
    Kövek =:= 7 - Luk, !, Na1 is Na + 1.
raktározás(N, _, _, Na, Na1) :-
    N > 0, !, Na1 is Na + N.
```

A kiürült térfelek kezelésekor a másik térfél köveit összeadjuk, és azt is kiürítjük:

```
vetés_vége(tábla(La,Na,Lb,Nb),
           tábla(La,Na,La,Nb1)) :-
    üres(La), !, összeg(Lb, X), Nb1 is Nb + X.
vetés_vége(tábla(La,Na,Lb,Nb),
           tábla(Lb,Na1,Lb,Nb)) :-
    üres(Lb), !, összeg(La, X), Na1 is Na + X.
vetés_vége(Állás, Állás) :- !.

```

```
9  üres([0,0,0,0,0,0]).
10
11 összeg(L, X) :- összeg(L, 0, X).
12
13 összeg([], A, A).
14 összeg([K|M], A, X) :-
15     A1 is A + K,
16     összeg(M, A1, X).
```

A saját oldali vetéssel így már majdnem készen vagyunk, csak a `felvesz_és_szór` hiányzik:

```
1 felvesz_és_szór(0, K, L, L1) :- % szórás folytatása
2     !, szór(K, L, L1).
3 felvesz_és_szór(1, K, [_|M], [0|M1]) :-
4     !, szór(K, M, M1).
5 felvesz_és_szór(Luk, K, [L|M], [L|M1]) :-
6     Luk > 1, !, Luk1 is Luk - 1,
7     felvesz_és_szór(Luk1, K, M, M1).
```

A `K` itt a szórandó kövek számát adja meg, a `Luk` pedig a lyuknak a száma, ahonnan szórunk. Ha a `Luk` értéke 0, az azt jelenti, hogy egy korábban elkezdődött vetés folytatódik, és az első kőnek az 1-es számú lyukba kell esnie. (A tényleges szórást a `szór` végzi.) Ha a `Luk` száma az 1-es, akkor a kapott lyuk-lista első elemét kinullázzuk (kivesszük belőle a köveket), a többit pedig a `szór` segítségével módosítjuk. Végül ha a `Luk` száma nagyobb, mint 1, akkor a lyuk-lista első eleme megmarad, a maradékot pedig rekurzív hívással tudjuk megadni.

A szórásnál lyukanként haladunk, és mindegyikbe egy kő kerül, amíg vagy nincs több lyuk, vagy nincs több kő:

```
szór(0, L, L) :- !.
szór(N, [L|M], [L1|M1]) :-
    N > 0, !,
    N1 is N - 1, L1 is L + 1,
    szór(N1, M, M1).
szór(_, [], []) :- !.
```

Hátra van még a vetés az ellenfél oldalán. Itt 4 esetet különböztetünk meg:

1. A vetni való kövek száma 0. Ilyenkor nincs teendő.

2. A kövek száma 1 és 6 között van (tehát nem jön vissza hozzánk), és a mi térfelünk nem üres. Ilyenkor egyszerűen végig kell szórni ezeket a köveket.

3. A kövek száma 1 és 6 között van, és a saját térfél üres. Ilyenkor az ellenfél kalahjába bekerülnek a szórandó kövek és az ellenfél térfelén levők is.

4. A kövek száma több, mint 6. Ekkor a kövek végigszórása után 6-al kevesebb kővel egy újabb vetést kell indítani a saját térfél „0-dik” lyukából.

Ennek megfelel az alábbi 4 szabály:

```
vet_ellenfél(0, Állás, Állás) :- !.
vet_ellenfél(Kövek, tábla(La,Na,Lb,Nb),
             tábla(La,Na,Lb1,Nb)) :-
    1 =< Kövek, Kövek =< 6,
    \+ üres(La), !,
```

```
    szór(Kövek, Lb, Lb1).
vet_ellenfél(Kövek, tábla(La,Na,Lb,Nb),
             tábla(La,Na,La,Nb1)) :-
    1 =< Kövek, Kövek =< 6,
    üres(La), !,
    összeg(Lb, X), Nb1 is Nb + Kövek + X.
vet_ellenfél(Kövek, tábla(La,Na,Lb,Nb), Állás) :-
    Kövek > 6, !,
    szór(6, Lb, Lb1),
    Kövek1 is Kövek - 6,
    vet(Kövek1, 0, tábla(La,Na,Lb1,Nb), Állás).
```

Kirajzolás

A nehezén túl vagyunk, de ahhoz, hogy játszani is tudjunk, meg is kell valahogy jeleníteni a táblát, és kommunikálni kell a játékossal.

A tábla így fog kinézni:

```
     6    6    6    6    6    6
0                                  0
     6    6    6    6    6    6
```

A kirajzolásnál mindig a(z ember) játékos sora lesz alul, tehát a számítógép sorát kell elsőnek kiírni. Ezt úgy érjük el, hogy ha a játékos jön, akkor megfordítjuk a táblát a tényleges kirajzolás előtt:

```
kirajzol(Állás, számítógép) :- kirajzol(Állás).
kirajzol(Állás, ember) :-
```

```
3       megfordít(Állás, Állás1),
4       kirajzol(Állás1).
```

A két sor kiírását a `sort_ír` végzi, a kalahokét a `kalahot_ír`. A felső sor számozása jobbról balra történik, tehát ezt a listát meg kell fordítani kiírás előtt:

```
1   kirajzol(tábla(La,Na,Lb,Nb)) :-
2       nl,
3       fordított(La, F),
4       sort_ír(F),
5       kalahot_ír(Na, Nb),
6       sort_ír(Lb).
```

A sorokat 5 szóköznyi behúzással indítjuk, hogy legyen hely a baloldali kalahnak is:

```
1   sort_ír(L) :- behúz(5), lyukat_ír(L).
2
3   lyukat_ír([]) :- nl.
4   lyukat_ír([L|M]) :- köveket_ír(L), lyukat_ír(M).
5
6   köveket_ír(N) :- N < 10, write(N), behúz(4).
7   köveket_ír(N) :- N >= 10, write(N), behúz(3).
8
9   kalahot_ír(N1, N2) :-
10      köveket_ír(N1), behúz(30),
11      write(N2), nl.
12
13  behúz(0) :- !.
```

```
14 behúz(N) :-
15     N > 0, N1 is N - 1,
16     write(' '), behúz(N1).
```

A `köveket_ír` mindig 3 vagy 4 szóközt ír annak függvényében, hogy egy- vagy kétjegyű a kövek száma.

Próba kétjátékosos módban

A mesterséges intelligencia még nincs meg, de a program már majdnem tesztelhető. Csak a `lépést_választ` hiányzik, amit egyelőre írjunk meg úgy, hogy mindig a játékost kérdezi:

```
1 lépést_választ(_, _, Lépés) :-
2     nl, kiír('Melyiket választod?'),
3     read(Lépés), érvényes(Lépés).
4
5 érvényes([]).
6 érvényes([L|M]) :- 0 < L, L < 7, érvényes(M).
```

A lépés érvényességére csak egy minimális ellenőrzés van, azt sem ellenőrzi, hogy nem kéne-e folytatnunk a lépést vagy hogy nem léptünk-e túl sokszor.

Itt egy pár lépés ízelítőnek:

```
?- kalah.

     6    6    6    6    6    6
0                                  0
     6    6    6    6    6    6
```

```
Melyiket választod?
|: [1,5].

     6    7    7    7    7    7
0                                  2
     0    7    7    7    0    8

Melyiket választod?
|: [3].

     7    8    8    0    7    7
1                                  2
     1    8    8    7    0    8

Melyiket választod?
|: [2].

     7    8    8    1    8    8
1                                  3
     1    0    9    8    1    9

Melyiket választod?
|: [1].

     8    9    9    2    9    0
2                                  3
     2    1    9    8    1    9
```

```
Melyiket választod?
|: [6].

     9    10   10   3    10   1
2                                 4
     3    2    9    8    1    0

Melyiket választod?
|: [3].

     10   11   11   0    10   1
2                                 4
     3    2    9    8    1    0

Melyiket választod?
|: [5].

     10   11   11   0    10   0
2                                 6
     3    2    9    8    0    0

Melyiket választod?
|: vége.

false.
```

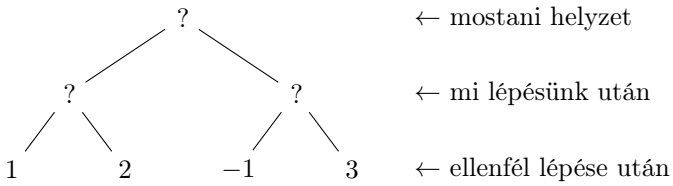
A program jelenleg tetszőleges hibás lépésre (pl. `vége`) leáll. Kipróbálhatjuk egy-egy speciális esetre is, mint például ez:

```
?- Állás = tábla([1,1,1,1,1,1],23,[16,3,1,1,3,3],16),
   kirajzol(Állás, ember), játék(Állás, ember).

     3    3    1    1    3    16
16                                  23
     1    1    1    1    1    1

Melyiket választod?
|: [6,5].

     3    3    1    1    3    0
16                                  41
     1    1    1    1    0    0
Nyertél, gratulálok!
```

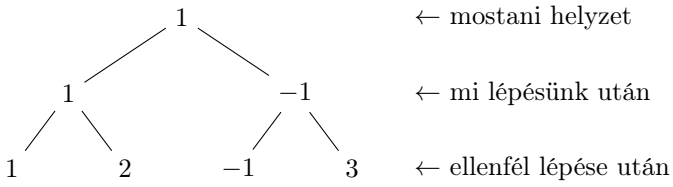
Alfa–béta nyírás keretprogram

A számítógép az ún. *alfa–béta nyírás* algoritmusa szerint fog működni. Ennek a lényege az, hogy minden játékálláshoz tudunk rendelni egy számot, ami annál magasabb, minél kedvezőbb nekünk az állás. Ezután végiggondoljuk az összes lépési lehetőséget (az ellenfél lépéseit is természetesen) valahány lépésre előre: ezt a lépésszámot szokás *mélységnek* nevezni, a programban az *előrelátás* rövidítéseként az `E` betűt fogjuk rá alkalmazni.

Nézzünk egy nagyon egyszerű példát! Az alábbi ábra lépések *fáját* mutatja: a pontok és számok játékállásokat jelölnek, míg a köztük levő szakaszok a lépéseket.

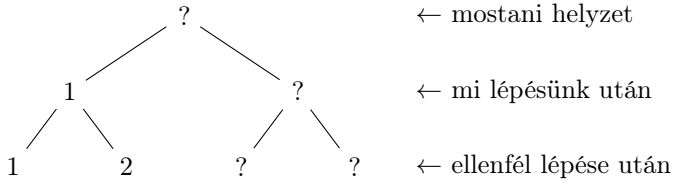


Itt a mélység 2, tehát 2 lépésre előre gondolkodunk, és a legmélyebb szinten kiértékelünk minden állást. Két lépéslehetőségünk van, és ezekre az ellenfélnek 2-2 válasza. Feltesszük, hogy az ellenfelünk jól játszik, tehát ha a baloldali lépést választjuk, arra az ellenfél a saját baloldali (1-es értékelésű) lépését fogja válaszolni, nem pedig a jobboldalit, ami nekünk kedvezőbb állást (2) eredményez. Hasonlóan járunk el a jobboldali lépésnél: az ellenfél két válasza közül feltételezzük a kisebbet (-1). Azt láttuk tehát, hogy a baloldali lépés legrosszabb esetben 1-es, a jobboldali -1-es értékelésű. Mi nyilván a nagyobbat választjuk, tehát a mostani helyzetünk 1-es értékelésű lesz:

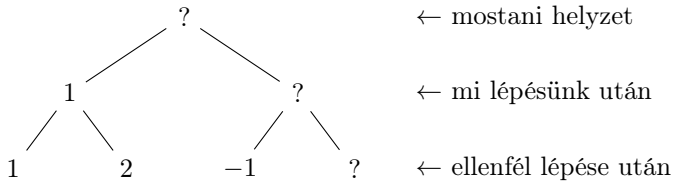


Ha itt kell lépést választani, akkor a baloldali mellett döntünk. Ezt a gondolkodást *minimax* algoritmusnak hívják, mivel az ellenfél lépései közül a minimális értékűt, a saját lépések közül a maximális értékűt választjuk.

Az alfa–béta nyírás ennek egy hatékonyabb változata. Itt mindig számon tartjuk azt, hogy mi az a minimum, amit biztosan el tudunk érni (α, a programban `A`), és mi az a maximum, amit elérhetünk (β, a programban `B`), ezek adják a módszer nevét. Nézzük meg az előbbi példa kiértékelését, amikor már megvizsgáltuk a teljes baloldali ágat!



A jobboldali ággal még egyáltalán nem foglalkoztunk. Azt tudjuk, hogy ha a baloldali lépést választjuk, legrosszabb esetben 1-es (tehát $\alpha = 1$). Hogyan módosul ez a jobboldali lépésre adott baloldali válasz megvizsgálása után?



Azt látjuk, hogy ha a jobboldali lépést választjuk, akkor legjobb esetben -1-re számíthatunk ($\beta = -1$). Lehet, hogy az ellenfélnek van egy még erősebb lépése, és a tényleges maximum

még kisebb, de nagyobb nem lehet. Viszont a baloldali ágon már van egy biztos 1-es, tehát ezt az ágat nem érdemes tovább vizsgálni, le lehet „nyírni”. Általában tehát ha egy ágon a β érték kisebb vagy egyenlő, mint az eddigi legjobb α, akkor nem kell vele foglalkozni.

Az algoritmust teljesen általánosan meg lehet fogalmazni, a konkrét játéktól függetlenül. Csak azt feltételezzük, hogy a következők adottak:

- `lépés(Állás, Lépés)`: adott állásból lehetséges lépés
- `lép(Lépés, Állás, Állás1)`: adott lépés lejátszása
- `értékelés(Állás, Érték)`: adott állás kiértékelése

Ebből egyelőre csak a második van meg, a maradék kettőt a következő részben fogjuk elkészíteni. Most azonban nézzük az általános algoritmust! Hogy ne kelljen külön kezelni a minimum- és maximumkereső eseteket, az értékeléseket minden szintváltáskor negáljuk. Ahhoz, hogy ez működjön, feltételezzük, hogy a keresési mélység kezdetben páros, tehát a 0-ás mélységen a pozitív szám jelenti a nekünk jó állást.

```
alfa_béta(0, Állás, _, _, _-Érték) :-
    értékelés(Állás, Érték).
alfa_béta(E, Állás, A, B, Lépés-Érték) :-
    E > 0, E1 is E - 1,
    A1 is -B, B1 is -A,
    findall(L, lépés(Állás, L), Lépések),
    választ(Lépések, Állás, E1, A1, B1,
            nincs, Lépés-Érték).
```

Ha az előrelátás mélysége 0, akkor az aktuális állást egyszerűen kiértékeljük. (Ilyenkor a legjobb lépést jelölő `Lépés` nem kap értéket.) Ha a mélység legalább 1, akkor eggyel csökkentjük, negáljuk és megcseréljük az alfát és bétát (hogy a másik játékos szemszögébe kerüljünk), és választunk az összes lehetséges lépés közül.

A választást a `választ` végzi:

```
választ([], _, _, A, _, Legjobb, Legjobb-A).
választ([Lépés|M], Állás, E, A, B, Legjobb, X) :-
    lép(Lépés, Állás, Állás1),
    alfa_béta(E, Állás1, A, B, _-Érték),
    Érték1 is -Érték,
    nyír(Lépés-Érték1, E, A, B, M, Állás,
         Legjobb, X).
```

Az utolsóelőtti argumentum az eddigi legjobb lépés (kezdetben `nincs`), az utolsó pedig a végső megtalált legjobb lépés és a hozzá tartozó értékelés.

Ha a lehetséges lépések listája üres, akkor az eddigi legjobb lépést adja vissza (`Legjobb`) és a hozzá tartozó érték az `A` lesz. Ha nem üres, akkor kipróbálja az első lépést: lejátsza, és az így keletkező állást (rekurzívan) kiértékeli az `alfa_béta` szabállyal. Az így kapott `Érték`-et negálni kell, mert egy szinttel feljebb léptünk. Végül a `nyír` az eredeti `Állás` állapotból való lépések közül (rekurzívan) választ, miközben lenyírja azokat az ágakat, amelyeket felesleges kiértékelni:

```
nyír(Lépés-Érték, _, _, B, _, _, _, Lépés-Érték) :-
    Érték >= B.
```

```
nyír(Lépés-Érték, E, A, B, Többi, Állás, _, X) :-
    A < Érték, Érték < B,
    választ(Többi, Állás, E, Érték, B, Lépés, X).
nyír(_-Érték, E, A, B, Többi, Állás, Lépés1, X) :-
    Érték =< A,
    választ(Többi, Állás, E, A, B, Lépés1, X).
```

Itt három esetet különböztetünk meg, a megvizsgált lépés értékelésétől függően:

1. Ha legalább β, akkor nem kell tovább keresni, ennél jobb lépés nem változtat az értékelésen. (Ez megfelel annak, hogy a fenti példában megtaláltuk a -1-et, ami az ellenfél számára negálva 1-es, és ezért nem kell még jobb lépés után néznie, mert a β (a mi negált α-nk) kisebb, -1 értékű.)

2. Ha α és β között van, akkor ez lesz az eddigi legjobb lépés és az értéke az új α, és keresünk egy esetleges jobbat a többi lépésből.

3. Ha kisebb vagy egyenlő, mint α, akkor ez a lépés nem érdekes, a többiekben folytatjuk a keresést.

Kalah-specifikus részek

Hogyan tudjuk az összes lehetséges lépést leírni?

```
lépés(tábla([0,0,0,0,0,0],_,_,_), []).
lépés(Állás, [L|M]) :-
    tartalmaz(L, [1,2,3,4,5,6]),
```

```
4       kövek(L, Állás, K),
5       lépést_folytat(K, L, Állás, M).
```

Ha a térfelünk üres, nincs lehetséges lépés. (Erre azért van szükség, mert a játékot be lehet fejezni úgy, hogy az utolsó követ a kalahba rakjuk.) Egyébként a lépés első eleme egy lyukat jelölő 1 és 6 közti szám; a `kövek` biztosítja, hogy van is benne kő. Azt kell csak ellenőrizni, hogy a kalahba kerül-e az utolsó, amire a feltétel a 13-al való osztási maradékkal számolható:

```
1   lépést_folytat(Kövek, L, _, []) :-
2       Kövek =\= (7 - L) mod 13, !.
3   lépést_folytat(Kövek, L, Állás, Lépések) :-
4       Kövek =:= (7 - L) mod 13, !,
5       vet(Kövek, L, Állás, Állás1),
6       lépés(Állás1, Lépések).
```

Ha a kalahba került, akkor a köveket végigszórjuk, és ebből az állásból rekurzívan további lépéseket keresünk.

Egy állás értékelésére egy nagyon egyszerű definíció a kalahokban levő kövek különbsége:

```
1   értékelés(tábla(_,Na,_,Nb), X) :- X is Na - Nb.
```

Már csak annyi van hátra, hogy átírjuk a `lépést_választ` szabályt. A régi verzió megmarad arra az esetre, amikor az emberi játékos van lépésen; a számítógép esetében pedig az alfa–béta nyírást használjuk:

```
1   lépést_választ(_, ember, Lépés) :-
2       nl, kiír('Melyiket választod?'),
```

```
    read(Lépés), érvényes(Lépés).
lépést_választ(Állás, számítógép, Lépés) :-
    előrelátás(E),
    alfa_béta(E, Állás, -40, 40, Lépés-_),
    nl, write(Lépés), nl.

előrelátás(4).
```

Az $[\alpha, \beta]$ intervallumot kezdetben jó nagyra állítjuk (-40, 40), az előrelátás mélységét pedig a könnyebb módosíthatóság kedvéért egy külön tényként tároljuk.

Tesztjáték

Itt egy 6-os mélységű mesterséges intelligencia ellen játszott 4-köves játék, ahol a gép kezdett:

```
?- kalah.

     4    4    4    4    4    4
0                                 0
     4    4    4    4    4    4

[3,6]

     0    5    5    0    4    4
2                                 0
     5    5    5    5    4    4

Melyiket választod?
```

```
|: [2,1].

     0    5    5    0    4    4
2                                  1
     0    1    7    7    6    6

[4]

     1    6    0    0    4    4
3                                  1
     1    2    7    7    6    6

Melyiket választod?
|: [1].

     1    6    0    0    4    4
3                                  1
     0    3    7    7    6    6

[5]

     2    0    0    0    4    4
4                                  1
     1    4    8    8    6    6

Melyiket választod?
|: [2].
```

```
     2    0    0    0    4    4
4                                  1
     1    0    9    9    7    7

[6]

     0    0    0    0    4    4
5                                  1
     2    0    9    9    7    7

Melyiket választod?
|: [1].

     0    0    0    0    4    4
5                                  1
     0    1    10   9    7    7

[1]

     0    0    1    1    5    0
7                                  1
     0    0    10   9    7    7

Melyiket választod?
|: [5].

     0    1    2    2    6    1
7                                  2
     0    0    10   9    0    8
```

```
[1]

     0    1    2    2    7    0
7                                  2
     0    0    10   9    0    8

Melyiket választod?
|: [6].

     0    2    3    3    8    1
7                                  5
     0    0    10   9    0    0

[5,6,4,6,1]

     0    1    0    3    9    0
11                                 5
     0    0    10   9    0    0

Melyiket választod?
|: [3].

     1    2    1    4    10   1
11                                 6
     0    0    0    10   1    1

[6,5,4]
```

```
     1    1    0    4    10   1
13                                 6
     0    0    0    10   1    1

Melyiket választod?
|: [4].

     0    2    1    5    11   2
13                                 10
     0    0    0    0    2    2

[1]

     0    2    1    6    12   0
13                                 10
     0    0    0    0    2    2

Melyiket választod?
|: [6].

     0    2    1    6    12   1
13                                 11
     0    0    0    0    2    0

[1]

     0    2    1    6    13   0
13                                 11
     0    0    0    0    2    0
```

```
Melyiket választod?
|: [5,6].

     0    0    0    0    0    0
35                                 13
     0    0    0    0    0    0
Nyertem.
```

46. Feladat. Írd át a játékos lépését, hogy hibás lépés esetén kérdezzen újra, és `kilépés`-re lépjen ki!

47. Feladat. Készíts szigorúbb ellenőrzést az emberi játékos lépéseihez, ami (i) nem enged üres lyukat választani, és (ii) akkor és csak akkor enged több lépést, ha az utolsó kő a kalahba kerül!

48. Feladat. Írd át a programot úgy, hogy a bonyolultabb (de izgalmasabb) Oware játék szabályait kövesse! A különbségek:

- A kövek száma általában lyukanként 4
- A szórásnál a kalahokat és a kezdő lyukat ki kell hagyni
- Követ úgy lehet szerezni, hogy ha olyan helyre érkezünk, ami (i) az ellenfél oldalán van és (ii) a szórás befejeztével 2 vagy 3 kő lesz benne. Ha ez teljesül, akkor az ebben levő összes követ megkapjuk, sőt, ha az előzőre is teljesül, akkor az abban levőket is és így tovább, amíg igaz a két feltétel.
- Ha lehet, muszáj olyat lépnünk, hogy az ellenfél oldalán maradjon kő. Ha nem lehet, akkor a maradék köveket megkapjuk, mint a kalahban.

49. Feladat. Írj dámajátékot! A játék kerete legyen ugyanaz, és a számítógép használja a fenti alfa–béta nyírás algoritmust.

A teljes program

```
% Magas szintű keretprogram

kalah :-
    alapbeállítás(Állás, Játékos),
    kirajzol(Állás, Játékos),
    játék(Állás, Játékos).

játék(Állás, Játékos) :-
    játék_vége(Állás, Játékos, Eredmény), !,
    bejelent(Eredmény).
játék(Állás, Játékos) :-
    lépést_választ(Állás, Játékos, Lépés),
    lép(Lépés, Állás, Állás1),
    következő_játékos(Játékos, Játékos1), !,
    kirajzol(Állás1, Játékos1),
    játék(Állás1, Játékos1).

következő_játékos(ember, számítógép).
következő_játékos(számítógép, ember).

bejelent(ember) :- kiír('Nyertél, gratulálok!').
bejelent(számítógép) :- kiír('Nyertem.').
bejelent(döntetlen) :- kiír('Döntetlen lett!').

```

```
25  % Reprezentáció
26
27  alapbeállítás(Állás, ember) :-
28      kövek_száma(K),
29      Állás = tábla([K,K,K,K,K,K],0,
30                    [K,K,K,K,K,K],0).
31
32  % Szabályok
33
34  játék_vége(tábla(_,N,_,N), _, döntetlen) :-
35      kövek_száma(K), N =:= 6 * K, !.
36  játék_vége(tábla(_,N1,_,_), Játékos, Játékos) :-
37      kövek_száma(K), N1 > 6 * K, !.
38  játék_vége(tábla(_,_,_,N2), Játékos, Másik) :-
39      kövek_száma(K), N2 > 6 * K,
40      következő_játékos(Játékos, Másik).
41
42  lép([], Állás, Állás1) :- megfordít(Állás, Állás1).
43  lép([L|M], Állás, Állás2) :-
44      kövek(L, Állás, K),
45      vet(K, L, Állás, Állás1),
46      lép(M, Állás1, Állás2).
47
48  megfordít(tábla(La,Na,Lb,Nb), tábla(Lb,Nb,La,Na)).
49
50  kövek(I, tábla(L,_,_,_), K) :-
51      n_edik(I, L, K), K > 0.
52
53  vet(Kövek, Luk, Állás, Állás2) :-
```

```
54      vet_saját(Kövek, Luk, Állás, Állás1, Kövek1),
55      vet_ellenfél(Kövek1, Állás1, Állás2).
56
57  vet_saját(Kövek, Luk, tábla(La,Na,Lb,Nb),
58            tábla(La1,Na1,Lb,Nb), Kövek1) :-
59      Kövek > 7 - Luk, !, % átmegy az ellenfélhez
60      felvesz_és_szór(Luk, Kövek, La, La1),
61      Na1 is Na + 1, Kövek1 is Kövek + Luk - 7.
62  vet_saját(Kövek, Luk,
63            tábla(La,Na,Lb,Nb), Állás, 0) :-
64      Kövek =< 7 - Luk,
65      felvesz_és_szór(Luk, Kövek, La, La1),
66      elfogás(Luk, Kövek, La1, La2, Lb, Lb1, N),
67      raktározás(N, Kövek, Luk, Na, Na1),
68      vetés_vége(tábla(La2,Na1,Lb1,Nb), Állás).
69
70  felvesz_és_szór(0, K, L, L1) :- % szórás folytatása
71      !, szór(K, L, L1).
72  felvesz_és_szór(1, K, [_|M], [0|M1]) :-
73      !, szór(K, M, M1).
74  felvesz_és_szór(Luk, K, [L|M], [L|M1]) :-
75      Luk > 1, !, Luk1 is Luk - 1,
76      felvesz_és_szór(Luk1, K, M, M1).
77
78  szór(0, L, L) :- !.
79  szór(N, [L|M], [L1|M1]) :-
80      N > 0, !,
81      N1 is N - 1, L1 is L + 1,
82      szór(N1, M, M1).
```

```
83  szór(_, [], []) :- !.
84
85  elfogás(Luk, Kövek, La, La1, Lb, Lb1, N) :-
86      Vége is Luk + Kövek,
87      n_edik(Vége, La, 1),
88      Szemben is 7 - Vége,
89      n_edik(Szemben, Lb, K),
90      K > 0, !, % üresbe érkeztünk és van szemben kő
91      n_csere(Vége, La, 0, La1),
92      n_csere(Szemben, Lb, 0, Lb1),
93      N is K + 1.
94  elfogás(_, _, La, La, Lb, Lb, 0) :- !.
95
96  raktározás(0, Kövek, Luk, Na, Na) :-
97      Kövek < 7 - Luk, !.
98  raktározás(0, Kövek, Luk, Na, Na1) :-
99      Kövek =:= 7 - Luk, !, Na1 is Na + 1.
100 raktározás(N, _, _, Na, Na1) :-
101     N > 0, !, Na1 is Na + N.
102
103 vetés_vége(tábla(La,Na,Lb,Nb),
104             tábla(La,Na,La,Nb1)) :-
105     üres(La), !, összeg(Lb, X), Nb1 is Nb + X.
106 vetés_vége(tábla(La,Na,Lb,Nb),
107             tábla(Lb,Na1,Lb,Nb)) :-
108     üres(Lb), !, összeg(La, X), Na1 is Na + X.
109 vetés_vége(Állás, Állás) :- !.
110
111 üres([0,0,0,0,0,0]).
```

```
112
113 vet_ellenfél(0, Állás, Állás) :- !.
114 vet_ellenfél(Kövek, tábla(La,Na,Lb,Nb),
115              tábla(La,Na,Lb1,Nb)) :-
116     1 =< Kövek, Kövek =< 6,
117     \+ üres(La), !,
118     szór(Kövek, Lb, Lb1).
119 vet_ellenfél(Kövek, tábla(La,Na,Lb,Nb),
120              tábla(La,Na,La,Nb1)) :-
121     1 =< Kövek, Kövek =< 6,
122     üres(La), !,
123     összeg(Lb, X), Nb1 is Nb + Kövek + X.
124 vet_ellenfél(Kövek, tábla(La,Na,Lb,Nb), Állás) :-
125     Kövek > 6, !,
126     szór(6, Lb, Lb1),
127     Kövek1 is Kövek - 6,
128     vet(Kövek1, 0, tábla(La,Na,Lb1,Nb), Állás).
129
130 % Kirajzolás
131
132 kirajzol(Állás, számítógép) :- kirajzol(Állás).
133 kirajzol(Állás, ember) :-
134     megfordít(Állás, Állás1),
135     kirajzol(Állás1).
136
137 kirajzol(tábla(La,Na,Lb,Nb)) :-
138     nl,
139     fordított(La, F),
140     sort_ír(F),
```

```
141     kalahot_ír(Na, Nb),
142     sort_ír(Lb).
143
144 sort_ír(L) :- behúz(5), lyukat_ír(L).
145
146 lyukat_ír([]) :- nl.
147 lyukat_ír([L|M]) :- köveket_ír(L), lyukat_ír(M).
148
149 köveket_ír(N) :- N < 10, write(N), behúz(4).
150 köveket_ír(N) :- N >= 10, write(N), behúz(3).
151
152 kalahot_ír(N1, N2) :-
153     köveket_ír(N1), behúz(30),
154     write(N2), nl.
155
156 % Alfa-béta nyírás
157
158 alfa_béta(0, Állás, _, _, _-Érték) :-
159     értékelés(Állás, Érték).
160 alfa_béta(E, Állás, A, B, Lépés-Érték) :-
161     E > 0, E1 is E - 1,
162     A1 is -B, B1 is -A,
163     findall(L, lépés(Állás, L), Lépések),
164     választ(Lépések, Állás, E1, A1, B1,
165             nincs, Lépés-Érték).
166
167 választ([], _, _, A, _, Legjobb, Legjobb-A).
168 választ([Lépés|M], Állás, E, A, B, Legjobb, X) :-
169     lép(Lépés, Állás, Állás1),
```

```
170     alfa_béta(E, Állás1, A, B, _-Érték),
171     Érték1 is -Érték,
172     nyír(Lépés-Érték1, E, A, B, M, Állás,
173          Legjobb, X).
174
175 nyír(Lépés-Érték, _, _, B, _, _, _, Lépés-Érték) :-
176     Érték >= B.
177 nyír(Lépés-Érték, E, A, B, Többi, Állás, _, X) :-
178     A < Érték, Érték < B,
179     választ(Többi, Állás, E, Érték, B, Lépés, X).
180 nyír(_-Érték, E, A, B, Többi, Állás, Lépés1, X) :-
181     Érték =< A,
182     választ(Többi, Állás, E, A, B, Lépés1, X).
183
184 % Kalah-specifikus rész
185
186 lépés(tábla([0,0,0,0,0,0,0],_,_,_), []).
187 lépés(Állás, [L|M]) :-
188     tartalmaz(L, [1,2,3,4,5,6]),
189     kövek(L, Állás, K),
190     lépést_folytat(K, L, Állás, M).
191
192 lépést_folytat(Kövek, L, _, []) :-
193     Kövek =\= (7 - L) mod 13, !.
194 lépést_folytat(Kövek, L, Állás, Lépések) :-
195     Kövek =:= (7 - L) mod 13, !,
196     vet(Kövek, L, Állás, Állás1),
197     lépés(Állás1, Lépések).
198
```

```
199 értékelés(tábla(_,Na,_,Nb), X) :- X is Na - Nb.
200
201 lépést_választ(_, ember, Lépés) :-
202     nl, kiír('Melyiket választod?'),
203     read(Lépés), érvényes(Lépés).
204 lépést_választ(Állás, számítógép, Lépés) :-
205     előrelátás(E),
206     alfa_béta(E, Állás, -40, 40, Lépés-_),
207     nl, write(Lépés), nl.
208
209 érvényes([]).
210 érvényes([L|M]) :- 0 < L, L < 7, érvényes(M).
211
212 % Beállítások
213
214 kövek_száma(6).
215
216 előrelátás(4).
217
218 % Segéd-szabályok
219
220 kiír(X) :- write(X), nl.
221
222 behúz(0) :- !.
223 behúz(N) :-
224     N > 0, N1 is N - 1,
225     write(' '), behúz(N1).
226
227 tartalmaz(X, [X|_]).
```

```
228 tartalmaz(X, [_|Maradék]) :- tartalmaz(X, Maradék).
229
230 fordított(X, Y) :- fordított(X, [], Y).
231
232 fordított([], Y, Y).
233 fordított([X|M], F, Y) :- fordított(M, [X|F], Y).
234
235 n_edik(N, [_|M], X) :-
236     N > 1, !, N1 is N - 1,
237     n_edik(N1, M, X).
238 n_edik(1, [X|_], X).
239
240 n_csere(1, [_|M], Y, [Y|M]) :- !.
241 n_csere(N, [X|M], Y, [X|M1]) :-
242     N > 1, !, N1 is N - 1,
243     n_csere(N1, M, Y, M1).
244
245 összeg(L, X) :- összeg(L, 0, X).
246
247 összeg([], A, A).
248 összeg([K|M], A, X) :-
249     A1 is A + K,
250     összeg(M, A1, X).
```

7. lecke

Mélyvíz

Most, hogy már tisztában vagyunk a Prolog alapjaival, nézzük meg, hogy hogyan néz ki egy „igazi” Prolog program: az asszociációs struktúrát megvalósító könyvtár SWI-Prologban.

Az alábbiakban bemutatott forráskódban nem szerepelnek az SWI-Prolog-specifikus részek és az angol nyelvű megjegyzések, de ettől (és helyenként formázástól) eltekintve megegyezik az eredetivel.

Mielőtt megnéznénk magát a programot, vizsgáljuk meg, hogy mi is a probléma, amire megoldást ad, és mi ennek az elméleti háttere.

7.1. Asszociációs listák

Programozáskor nagyon gyakran előfordul, hogy adatokat *kulcsokhoz* rendelünk, amelyek szerint később ki akarjuk majd keresni őket. Az a feltételezés, hogy különböző adatokhoz mindig különböző kulcs tartozik. Ilyen kulcs lehet pl. egy bankban a számlaszám, amihez a megfelelő számla adatait rendelik. A kulcs–érték párokat tároló adatstruktúrákat gyakran nevezik *szótáraknak* is, hiszen egy szótárban (enciklopédiában stb.) is egy-egy címszóhoz vannak rendelve a jelentések/magyarázatok.

A szótár legegyszerűbb megvalósítása az *asszociációs lista*, ahol a párokat egy listában tároljuk:

```
1  empty_assoc([]).
2
3  put_assoc(K, A, V, [K-V|A]).
4
5  get_assoc(K, [K-V|_], V) :- !.
6  get_assoc(K, [K1-_|A], V) :-
7      K \= K1, get_assoc(K, A, V).
8
9  del_assoc(K, [K-V|A], V, A1) :-
10     !, del_assoc_others(K, A, A1).
11 del_assoc(K, [K1-V1|A], V, [K1-V1|A1]) :-
12     K \= K1, del_assoc(K, A, V, A1).
13
14 del_assoc_others(_, [], []) :- !.
15 del_assoc_others(K, [K-_|A], A1) :-
```

```
16      !, del_assoc_others(K, A, A1).
17  del_assoc_others(K, [K1-V1|A], [K1-V1|A1]) :-
18      K \= K1, del_assoc_others(K, A, A1).
```

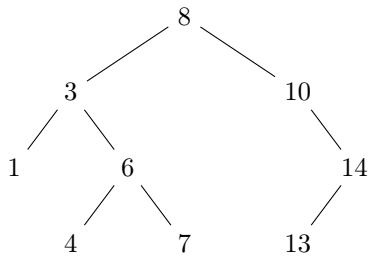
Itt az üres szótár egyszerűen egy üres lista; a `put_assoc` egy asszociációs lista elejére rak be egy kulcs-érték párt; a `get_assoc` megkeresi a listában az első adott kulcsú értéket; a `del_assoc` pedig kitörli ugyanezt. A törlésnél figyelni kell arra, hogy az összes lehetséges előfordulást töröljük, ezért ez mindig a teljes listán végigmegy.

Amíg nincsen nagyon sok adatunk, ez a megoldás elég jól működik. Az egyszerűségnek azonban ára van: mind a keresés, mind a törlés általános esetben az elemek számával arányos. Ezt úgy szokás megfogalmazni, hogy a keresés és törlés komplexitása $O(n)$, ahol n az elemek száma. Ez az $O(n)$ (kiolvasva *ordó enn*) azt mondja, hogy nem biztos, hogy pontosan n művelet, lehet hogy $n+2$, vagy $5n$, de ha az n-et 100-szor akkorára választom, akkor a műveletigény is körülbelül 100-szor akkorára nő. (Egy $O(n^2)$-es algoritmus esetén ilyenkor a műveletigény kb. a 10 000-szeresére változna.)

A következőkben bemutatott módszer olyan, hogy mindhárom művelet (beszúrás, keresés, törlés) egyaránt $O(\log n)$ komplexitású, tehát ha az n a 100-szorosára nő, akkor a műveletigény kb. 6–7-szeresére változik. A beszúrás a fenti egyszerű verzióban gyorsabb – $O(1)$ –, de a keresés és törlés hatékonysága miatt érdemesebb az alábbi adatstruktúrákat alkalmazni.

7.2. Bináris keresőfák

Tegyük fel, hogy a kulcsok sorbarendezhetőek (Prologban tetszőleges két kifejezés sorbarendezhető a `@<` operátorral). Ekkor a kulcsokat egy olyan (általában fejjel lefele ábrázolt) fa alakú struktúrába lehet szervezni, ami mindig legfeljebb kétfelé ágazik el (ezért *bináris fának* hívják). Nézzünk egy példát, ahol a kulcsok számok:



Itt a fa *gyökere* a 8, belső *csúcsok* a 3, 10, 6 és 14, és a fa *levelei* az 1, 4, 7 és 13. Amikor egy csúcsból csak egy ág megy tovább (pl. 10, 14), olyankor is az ág vagy balra, vagy jobbra megy. Egy ilyen fát könnyen le tudunk írni Prologban, például egy `fa` struktúrával, amelynek az első argumentuma a kulcs, a második és harmadik pedig a bal- és jobboldali ág (a nem létező ágakat jelölje mondjuk a `-`):

```
fa(8, fa(3, fa(1, -, -),
            fa(6, fa(4, -, -),
                  fa(7, -, -))),
      fa(10, -,
```

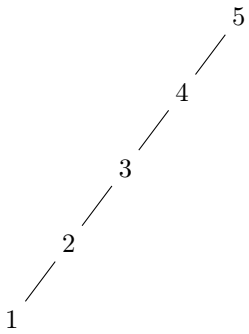
```
5               fa(14, fa(13, -, -),
6                      -)))
```

(Egy másik lehetőség, hogy a levelekre egy külön `levél` funktort használunk stb.)

A fenti példában szereplő fának van egy különleges tulajdonsága: egy csúcs alatti baloldali ágon (és az abból kijövő ágakon stb.) minden elem kisebb, a jobboldali ágon pedig mindegyik nagyobb, mint a csúcsban levő érték. Az ilyen tulajdonságú fákat *keresőfának* nevezik.

Ha meg akarunk keresni egy elemet, akkor elindulunk a gyökértől, és aszerint, hogy a keresett kulcs kisebb, vagy nagyobb, balra ill. jobbra megyünk tovább. Ezt addig folytatjuk, amíg meg nem találjuk a keresett elemet, vagy egy levélhez nem érünk. A keresés műveletigénye tehát a fa magasságával arányos. A beszúrásról és törlésről hasonlóan megmutatható, hogy a fa magasságától függ a komplexitásuk. Ha az adatok szépen egyenletesen helyezkednek el, akkor ez hozzávetőlegesen $\log n$ lesz (2-es alapú logaritmussal).

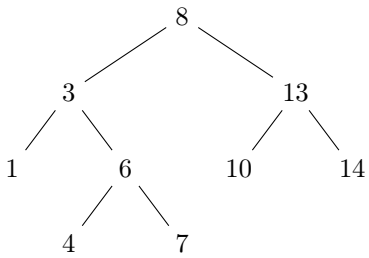
Sajnos azonban ez nem feltétlenül teljesül – pl. ez is egy keresőfa:



. . . de itt a fa magassága az elemek számával azonos.

7.3. AVL-fák

Egy lehetséges megoldása ennek a problémának az, hogy megköveteljük, hogy a fa mindig *kiegyensúlyozott* legyen, tehát minden csúcsnál a bal- és jobb ághoz tartozó részfa magassága legfeljebb 1-el különbözhet. A fenti példában a 8-as alatti két részfa magassága egyaránt 3, a 3-as alatti két részfa magassága 1 és 2, de pl. a 10-es alatti két részfa magassága 0 és 2, tehát ez nem kiegyensúlyozott. Ha a 10-14-13 hármast „átforgatjuk”, akkor kiegyensúlyozottá válik:



A beszúrás és törlés műveletekbe ilyen jellegű forgatásokat épít be az *AVL-fa*, hogy biztosítja a kiegyensúlyozottságot. Ezt a módszert 1962-ben publikálta két szovjet matematikus, Adelszon-Velszkij és Landisz, az ő vezetéknevükből származik az adastruktúra elnevezése.

Az alább vizsgált program AVL-fát használ a szótár megvalósítására – a pontos részleteket majd útközben megbeszéljük. Kalandra fel!

7.4. A program

```
1  /*  Part of SWI-Prolog
2
3      Author:        R.A.O'Keefe, L.Damas, V.S.Costa, Glenn Burgess,
4                     Jiri Spitz and Jan Wielemaker
5      E-mail:        J.Wielemaker@vu.nl
6      WWW:           http://www.swi-prolog.org
7      Copyright (c)  2004-2018, various people and institutions
8      All rights reserved.
9
10     Redistribution and use in source and binary forms, with or without
11     modification, are permitted provided that the following conditions
12     are met:
```

Kitérő: *asszociatív tárolók*

Egy másik, kicsit bonyolultabb keresőfa a *piros-fekete fa*, ami az egyes elemekhez (piros vagy fekete) színt rendel, és ennek segítségével még hatékonyabb beszúrást/törlést tesz lehetővé, mint az AVL-fa. Cserébe viszont a keresés egy kicsit lassabb lehet.

A kulcs szerinti keresés problémájára még egy nagyon érdekes megoldást adnak a *hash táblák*, melyeknek rengeteg variánsa létezik. Ezek általában sokkal gyorsabbak, mint a keresőfák ($O(1)$ átlagosan), de időnként lassabbak is lehetnek ($O(n)$ legrosszabb esetben), valamint az elemeket nem rendezetten tárolják.

Prologban nincs mód egy adat *megváltoztatására*, csak egy módosított változat készítésére (tehát az adatok *perzisztensek*). Ilyen megkötések mellett a hatékonysághoz időnként trükkök kellenek – erről szól Chris Okasaki könyve,[†] nagyon érdekes olvasmány.

[†]Ch. Okasaki, *Purely Functional Data Structures*, Cambridge, 1996.

```
13
14      1. Redistributions of source code must retain the above copyright
15         notice, this list of conditions and the following disclaimer.
16
17      2. Redistributions in binary form must reproduce the above copyright
18         notice, this list of conditions and the following disclaimer in
19         the documentation and/or other materials provided with the
20         distribution.
21
22      THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
23      "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
24      LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
25      FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
26      COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
27      INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
28      BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
29      LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
30      CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
31      LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
32      ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
33      POSSIBILITY OF SUCH DAMAGE.
34  */
```

Ez (szó szerint) a „kisbetűs rész”. A programok elején szokás felsorolni a szerzőket, illetve meghatározni, hogy a program milyen feltételekkel terjeszthető. A `/*` és `*/` közti rész megjegyzésnek számít, olyan, mint ha minden sor elején lenne egy `%` szimbólum.

Mielőtt rátérnénk a lényegre, érdemes még pár szót ejteni a *modulok*ról. Prologban a programokat könyvtárakba vagy modulokba lehet szervezni. Ezek megadásának módja nem szerepelt az eredeti ISO-szabványban, és bár egy későbbi kiegészítésbe belekerült, ezt kevés Prolog rendszer követi, mindegyik kicsit máshogyan működik.

A modulok mindig tartalmaznak egy listát azon *exportált* szabályokról, amelyek „kívülről” (más programfájlokból) is látszanak. Ezzel el lehet rejteni azokat a szabályokat, amelyek

nem tartoznak hozzá a könyvtár lényegi funkcióihoz (az *interfész*hez). A fenti kezdetleges asszociációs listánknál például a `del_assoc_others` volt egy ilyen lokális szabály, amit a `del_assoc` ugyan használ, de a könyvtárat használó más program már jobb, ha nem lát.

A kompatibilitás kedvéért a modul definícióját kihagyjuk, de az alábbiakban az exportált szabályok könnyen felismerhetőek lesznek az őket megelőző megjegyzésről, ahogy máris látni fogjuk.

```
35 % empty_assoc(?Assoc) [semidet]
36 empty_assoc(t).
```

A *semidet* egyike a szabályok öt lehetséges kategóriájának:

- *det* (determinisztikus): mindig pontosan egyszer teljesül, pl. `összeg(+L, -Ö)`
- *semidet* (félig determinisztikus): legfeljebb egyszer teljesül, pl. `maximum(+L, -M)` [üres listára sikertelen]
- *multi* (többszörös): legalább egyszer teljesül, ilyen pl. a `permutáció(+L, -P)`
- *nondet* (nemdeterminisztikus): többször teljesülhet, de lehet sikertelen is, pl. `tartalmaz(?E,?L)`
- *failure* (sikertelen): sosem teljesül, pl. `fail`

Ezeket mind úgy kell érteni, hogy „ha a dokumentációjának megfelelően adjuk meg a paramétereket”.

Visszatérve az AVL-fára, az üres fát itt a `t` atom fogja jelölni (nem a `-`, mint fent a bináris fánál).

```
37 % assoc_to_list(+Assoc, -Pairs) [det]
38 assoc_to_list(Assoc, List) :-
39     assoc_to_list(Assoc, List, []).
40
41 assoc_to_list(t(Key,Val,_,L,R), List, Rest) :-
42     assoc_to_list(L, List, [Key-Val|More]),
43     assoc_to_list(R, More, Rest).
44 assoc_to_list(t, List, List).
```

Az `assoc_to_list` szabály egy AVL-fából párok listáját hozza létre. Akkumulátoros megoldás (ld. 6. lecke), tehát egy üres akkumulátor paraméterrel meghívja a 3-argumentumú változatot. Az AVL-fa egy csúcsát a `t(K,V,B,L,R)` struktúra írja le, ahol `K` és `V` a kulcs és a hozzá tartozó érték, `B` a kiegyensúlyozáshoz használt szimbólum (`-` ha a két részfa azonos magasságú, `<` ill. `>` ha a baloldali ill. jobboldali magasabb), `L` és `R` pedig a bal- és jobboldali részfa.

```
45 % assoc_to_keys(+Assoc, -Keys) [det]
46 assoc_to_keys(Assoc, List) :-
47     assoc_to_keys(Assoc, List, []).
48
49 assoc_to_keys(t(Key,_,_,L,R), List, Rest) :-
50     assoc_to_keys(L, List, [Key|More]),
51     assoc_to_keys(R, More, Rest).
52 assoc_to_keys(t, List, List).
53
54 % assoc_to_values(+Assoc, -Values) [det]
55 assoc_to_values(Assoc, List) :-
```

```
56      assoc_to_values(Assoc, List, []).
57
58  assoc_to_values(t(_,Value,_,L,R), List, Rest) :-
59      assoc_to_values(L, List, [Value|More]),
60      assoc_to_values(R, More, Rest).
61  assoc_to_values(t, List, List).
```

Ez a két szabály gyakorlatilag ugyanaz, mint az előző, csak nem kulcs-érték párokat gyűjtenek ki listába, hanem rendre kulcsokat illetve értékeket.

```
62  % is_assoc(+Assoc) [semidet]
63  is_assoc(Assoc) :-
64      is_assoc(Assoc, _Min, _Max, _Depth).
65
66  is_assoc(t,X,X,0) :- !.
67  is_assoc(t(K,_,-,t,t),K,K,1) :- !, ground(K).
68  is_assoc(t(K,_,>,t,t(RK,_,-,t,t)),K,RK,2) :-
69      !, ground((K,RK)), K @< RK.
70
71  is_assoc(t(K,_,<,t(LK,_,-,t,t),t),LK,K,2) :-
72      !, ground((LK,K)), LK @< K.
73
74  is_assoc(t(K,_,B,L,R),Min,Max,Depth) :-
75      is_assoc(L,Min,LMax,LDepth),
76      is_assoc(R,RMin,Max,RDepth),
77      compare(Rel,RDepth,LDepth),
78      balance(Rel,B),
79      ground((LMax,K,RMin)),
```

```
80      LMax @< K,
81      K @< RMin,
82      Depth is max(LDepth, RDepth)+1.
83
84  balance(=,-).
85  balance(<,<).
86  balance(>,>).
```

Az `is_assoc` ellenőrzi, hogy a paraméterben kapott kifejezés egy helyes AVL-fa-e. Az első szabály csak meghívja a 4-argumentumú verziót. A többi az 5 lehetséges esetet kezeli:

1. Az *üres fa* egy helyes AVL-fa. (A többi paraméter értéke itt érdektelen, de azért valami értelmesre vannak beállítva.)

2. A *levél* minimuma és maximuma is a levélben szereplő kulcs, a mélysége pedig 1. A `ground` azt ellenőrzi, hogy a kulcsban nem szerepel ismeretlen változó. (Ez nem teljesen szabványos, de a gyakorlatban minden Prolog implementáció támogatja.)

3. Ha csak a *baloldali részfa üres*, a jobboldali részfa alatti részfák üresek kell, hogy legyenek, és az egész fa mélysége 2. Ellenőrzi, hogy a kulcsokban nincsenek ismeretlenek, és a jobboldali részfa kulcsa nagyobb, mint a gyökéré.

4. Ha csak a *jobboldali részfa üres*, akkor ugyanez fordítva.

5. Ha *egyik részfa sem üres*, akkor rekurzívan ellenőrzi mindkettőt. A fa minimuma a baloldali részfa minimuma, a

maximuma pedig a jobboldali részfa maximuma lesz. A részfák mélységét összehasonlítja a `compare` segítségével (szintén egy csak *de facto* szabványos szabály), ami az első argumentumban `<`, `=` vagy `>` lesz. Ellenőrzi, hogy ennek megfelel-e a fában szereplő `B` szimbólum (a `balance` az `=` jelből - jelet csinál), és hogy a kulcs a baloldali maximum és a jobboldali minimum közé esik. A mélység eggyel több, mint a két részfa mélysége közül a nagyobbik.

```
87 % gen_assoc(?Key, +Assoc, ?Value) [nondet]
88 gen_assoc(Key, Assoc, Value) :-
89     (   ground(Key)
90     ->  get_assoc(Key, Assoc, Value)
91     ;   gen_assoc_(Key, Assoc, Value)
92     ).
93
94 gen_assoc_(Key, t(_,_,_,L,_), Val) :-
95     gen_assoc_(Key, L, Val).
96 gen_assoc_(Key, t(Key,Val,_,_,_), Val).
97 gen_assoc_(Key, t(_,_,_,_,R), Val) :-
98     gen_assoc_(Key, R, Val).
```

A `gen_assoc` egy olyan keresés, ami fordítva is tud működni: meg tud keresni egy adott értékhez tartozó kulcsot (természetesen ez nem lesz hatékony), vagy ha az érték sincsen megadva, akkor végigmegy az összes kulcs-érték páron. Ha a kulcs meg van adva, akkor ugyanazt csinálja, mint a `get_assoc`.

Mivel először a baloldali részfát vizsgálja, utána a gyökérben levő értéket, és végül a jobboldali részfát (*infix* bejárás), a kulcsokat növekvő sorrendben fogja végigvenni.

```
99  % get_assoc(+Key, +Assoc, -Value) [semidet]
100 get_assoc(Key, t(K,V,_,L,R), Val) :-
101     compare(Rel, Key, K),
102     get_assoc(Rel, Key, V, L, R, Val).
103
104 get_assoc(=, _, Val, _, _, Val).
105 get_assoc(<, Key, _, Tree, _, Val) :-
106     get_assoc(Key, Tree, Val).
107 get_assoc(>, Key, _, _, Tree, Val) :-
108     get_assoc(Key, Tree, Val).
```

Megadott kulcs alapján keres.

```
109 % get_assoc(+Key, +Assoc0, ?Val0, ?Assoc, ?Val)
110 %                                        [semidet]
111 get_assoc(Key, t(K,V,B,L,R), Val,
112           t(K,NV,B,NL,NR), NVal) :-
113     compare(Rel, Key, K),
114     get_assoc(Rel, Key, V, L, R, Val,
115               NV, NL, NR, NVal).
116
117 get_assoc(=, _, Val, L, R, Val, NVal, L, R, NVal).
118 get_assoc(<, Key, V, L, R, Val, V, NL, R, NVal) :-
119     get_assoc(Key, L, Val, NL, NVal).
120 get_assoc(>, Key, V, L, R, Val, V, L, NR, NVal) :-
121     get_assoc(Key, R, Val, NR, NVal).
```

Ez a változat arra használható, hogy egy már létező kulcshoz tartozó értéket lecseréljünk. A harmadik argumentum a kulcshoz tartozó régi érték, a negyedik az így keletkező AVL-fa, és az

utolsó az új érték.

```
% list_to_assoc(+Pairs, -Assoc) [det]
list_to_assoc(List, Assoc) :-
    (  List = [] -> Assoc = t
    ;  keysort(List, Sorted),
       length(Sorted, N),
       list_to_assoc(N, Sorted, [], _, Assoc)
    ).

list_to_assoc(1, [K-V|More], More,
              1, t(K,V,-,t,t)) :- !.
list_to_assoc(2, [K1-V1,K2-V2|More], More,
              2, t(K2,V2,<,t(K1,V1,-,t,t),t)) :- !.
list_to_assoc(N, List, More,
              Depth, t(K,V,Balance,L,R)) :-
    N0 is N - 1,
    RN is N0 div 2,
    Rem is N0 mod 2,
    LN is RN + Rem,
    list_to_assoc(LN, List, [K-V|Upper], LDepth, L),
    list_to_assoc(RN, Upper, More, RDepth, R),
    Depth is LDepth + 1,
    compare(B, RDepth, LDepth), balance(B, Balance).
```

Az `assoc_to_list` fordítottja. A fa hatékony építéséhez először sorbarakja az elemeket a kulcsok szerint a `keysort` szabály segítségével. (Ez megint egy nem teljesen szabványos, de általánosan elfogadott szabály – szükség esetén könnyen megírható az

összefűzéses keresés mintájára. Ugyanez igaz a `lengthre`, amit mi `hossz` néven definiáltunk.)

A fa építését az 5-argumentumú verzió végzi. Az első argumentum az elemek száma, a második és harmadik elem együtt a rendezett elemek különbség-listája, a negyedik a fa mélysége, és az utolsó a készített AVL-fa. Az elemek száma alapján:

1. Ha *1 elem van*, akkor egy 1 mélységű, egy levélből álló fa az eredmény.

2. Ha *2 elem van*, akkor egy 2 mélységű, egy bal-levéllel rendelkező fa az eredmény.

3. Ha *legalább 3 elem van*, akkor rekurzívan elkészít két részfát, amelyek a lista első ill. második felét tartalmazzák. Kicsit pontosabban, a két részfa összesen $n-1$ elemet tárol (mivel 1 a gyökérbe kerül), és ha ez nem páros, akkor a baloldaliban lesz több elem. A különbség-lista itt lesz hasznos: az első `LN` elemből elkészül az `L` fa, és a `List` listából fel nem használt maradékot a `[K-V|Upper]` listával egyesíti. Ezáltal a gyökérhez tartozó kulcs-érték pár és a jobboldali fa építéséhez szükséges `Upper` lista is rögtön adott. Mivel a baloldali részfában van több elem, a mélység eggyel több lesz, mint a baloldali mélység. Végül a bal- és jobb mélység alapján kiszámolja a gyökérhez tartozó `B` értéket (`<`, `-` vagy `>`).

```
144 % ord_list_to_assoc(+Pairs, -Assoc) [det]
145 ord_list_to_assoc(Sorted, Assoc) :-
146     (  Sorted = [] -> Assoc = t
```

```
147     ;  length(Sorted, N),
148        list_to_assoc(N, Sorted, [], _, Assoc)
149     ).
```

Ugyanez, csak már feltételezi, hogy a lista elemei rendezettek.

```
150 % map_assoc(:Pred, +Assoc) [semidet]
151 map_assoc(Pred, T) :-
152     map_assoc_(T, Pred).
153
154 map_assoc_(t, _).
155 map_assoc_(t(_,Val,_,L,R), Pred) :-
156     map_assoc_(L, Pred),
157     call(Pred, Val),
158     map_assoc_(R, Pred).
```

Infix bejárással végigmegy az elemeken, és mindegyikre meghívja az első argumentumban kapott szabályt.

```
159 % map_assoc(:Pred, +Assoc0, ?Assoc) [semidet]
160 map_assoc(Pred, T0, T) :-
161     map_assoc_(T0, Pred, T).
162
163 map_assoc_(t, _, t).
164 map_assoc_(t(Key,Val,B,L0,R0), Pred,
165            t(Key,Ans,B,L1,R1)) :-
166     map_assoc_(L0, Pred, L1),
167     call(Pred, Val, Ans),
168     map_assoc_(R0, Pred, R1).
```

Ez a változat két argumentumot ad a kapott szabályhoz: az első (`Val`) az éppen vizsgált érték, mint az előző verzióban, a második (`Ans`) pedig egy változó, amire az adott elemet lecserélve egy új fát épít. Ezzel tehát lehet olyat csinálni, hogy minden értéket a fában négyzetre emelünk stb.

```
169 % max_assoc(+Assoc, -Key, -Value) [semidet]
170 max_assoc(t(K,V,_,_,R), Key, Val) :-
171     max_assoc(R, K, V, Key, Val).
172
173 max_assoc(t, K, V, K, V).
174 max_assoc(t(K,V,_,_,R), _, _, Key, Val) :-
175     max_assoc(R, K, V, Key, Val).
```

Megkeresi a legnagyobb kulcsot a fában, és a hozzá tartozó értéket.

```
176 % min_assoc(+Assoc, -Key, -Value) [semidet]
177 min_assoc(t(K,V,_,L,_), Key, Val) :-
178     min_assoc(L, K, V, Key, Val).
179
180 min_assoc(t, K, V, K, V).
181 min_assoc(t(K,V,_,L,_), _, _, Key, Val) :-
182     min_assoc(L, K, V, Key, Val).
```

Ugyanez a legkisebb kulcsra.

```
183 % put_assoc(+Key, +Assoc0, +Value, -Assoc) [det]
184 put_assoc(Key, A0, Value, A) :-
185     insert(A0, Key, Value, A, _).
```

```
186
187 insert(t, Key, Val, t(Key,Val,-,t,t), yes).
188 insert(t(Key,Val,B,L,R), K, V,
189        NewTree, WhatHasChanged) :-
190     compare(Rel, K, Key),
191     insert(Rel, t(Key,Val,B,L,R), K, V,
192            NewTree, WhatHasChanged).
193
194 insert(=, t(Key,_,B,L,R), _, V, t(Key,V,B,L,R), no).
195 insert(<, t(Key,Val,B,L,R), K, V,
196        NewTree, WhatHasChanged) :-
197     insert(L, K, V, NewL, LeftHasChanged),
198     adjust(LeftHasChanged, t(Key,Val,B,NewL,R),
199            left, NewTree, WhatHasChanged).
200 insert(>, t(Key,Val,B,L,R), K, V,
201        NewTree, WhatHasChanged) :-
202     insert(R, K, V, NewR, RightHasChanged),
203     adjust(RightHasChanged, t(Key,Val,B,L,NewR),
204            right, NewTree, WhatHasChanged).
```

A beszúrásnál először az `insert/5` szabály hívódik meg, aminek utolsó argumentuma azt mondja meg, hogy nőtt-e a fa mélysége. Ez az üres fa esetét lekezeli, egyébként pedig a felelősséget az `insert/6` szabályra hárítja. Ennek az argumentumai:

1. A gyökérben levő kulcs hogyan viszonyul a beszúrandó kulcshoz (`<`, `=`, `>`).
2. Az eredeti AVL-fa.
3. A beszúrandó kulcs.

4. A beszúrandó érték.
5. Az új AVL-fa.
6. Nőtt-e a fa mélysége.

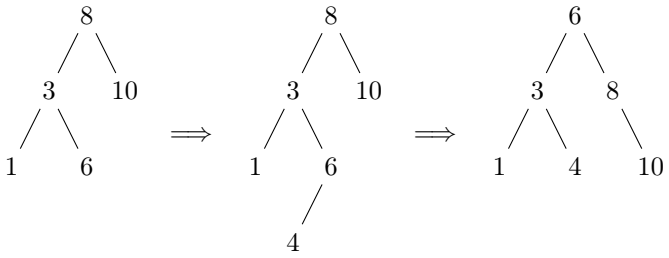
Ha a kulcsok megegyeznek, akkor a hozzá tartozó értéket a megadottra lecseréli. Ha a beszúrandó kulcs a kisebb, akkor a baloldali részfán végez rekurzívan beszúrást (az `insert/5`-tel), majd az `adjust` szabállyal (ld. lent) biztosítja a kiegyensúlyozottságot; ha a beszúrandó kulcs a nagyobb, akkor ugyanez a jobboldali részfával.

```
205 adjust(no, Oldree, _, Oldree, no).
206 adjust(yes, t(Key,Val,B0,L,R), LoR,
207        NewTree, WhatHasChanged) :-
208     table(B0, LoR, B1,
209           WhatHasChanged, ToBeRebalanced),
210     rebalance(ToBeRebalanced, t(Key,Val,B0,L,R), B1,
211               NewTree, _, _).
212
213 table(-, left , <, yes, no ) :- !.
214 table(-, right, >, yes, no ) :- !.
215 table(<, left , -, no , yes) :- !.
216 table(<, right, -, no , no ) :- !.
217 table(>, left , -, no , no ) :- !.
218 table(>, right, -, no , yes) :- !.
```

Az `adjust` első argumentuma, hogy történt-e beszúrás. Ha nem, akkor a kulcsok változatlanok, tehát a kiegyensúlyozottság továbbra is teljesül. A második argumentum a módosított AVL-fa,

a harmadik azt mondja meg, hogy a bal vagy jobb részfát módosítottuk (`left` ill. `right`), a negyedik a kiegyensúlyozás után kapott AVL-fa, az utolsó pedig azt jelzi, hogy nőtt-e a fa mélysége.

A `table` táblázatból könnyen kiolvasható, hogy a 6 lehetséges esetben mi történik: mi lesz az új egyensúly-szimbólum (`<`, `-` vagy `>`), megnövekedik a mélység, illetve szükség van-e forgatásra. Látszik, hogy csak két esetben van szükség forgatásra. Nézzük meg, mi történik az alábbi AVL-fával a 4-es kulcs beszúrásakor!



A 4-es beszúrása után a 6-os `<` típusú lesz, a 3-as `>` típusú, de probléma csak a legfelső szinten jelentkezik, ahol a 8-as már eleve `<` típusú volt. Itt tehát az egész fára fog meghívódni a forgató `rebalance` operáció, melynek eredménye jobboldalt látszik.

```
% del_min_assoc(+Assoc0, ?Key, ?Val, -Assoc)
%                                          [semidet]
del_min_assoc(Tree, Key, Val, NewTree) :-
    del_min_assoc(Tree, Key, Val,
                  NewTree, _DepthChanged).
```

```
224
225 del_min_assoc(t(Key,Val,_B,t,R), Key, Val,
226                R, yes) :- !.
227 del_min_assoc(t(K,V,B,L,R), Key, Val,
228                NewTree, Changed) :-
229     del_min_assoc(L, Key, Val, NewL, LeftChanged),
230     deladjust(LeftChanged, t(K,V,B,NewL,R),
231                left, NewTree, Changed).
```

Kitörli a legkisebb kulcsú elemet. `Val` a hozzá tartozó érték, és az utolsó argumentum a törléssel keletkezett AVL-fa. Ha a baloldali részfa üres, akkor a keresett kulcs a gyökérben van, és az új fa a jobboldali részfa lesz. Egyébként a baloldali részfában végezzük rekurzívan a törlést, és utána kiegyensúlyozzuk a `deladjust` szabály segítségével (ld. lent).

```
232 % del_max_assoc(+Assoc0, ?Key, ?Val, -Assoc)
233 %                                        [semidet]
234 del_max_assoc(Tree, Key, Val, NewTree) :-
235     del_max_assoc(Tree, Key, Val,
236                   NewTree, _DepthChanged).
237
238 del_max_assoc(t(Key,Val,_B,L,t), Key, Val,
239                L, yes) :- !.
240 del_max_assoc(t(K,V,B,L,R), Key, Val,
241                NewTree, Changed) :-
242     del_max_assoc(R, Key, Val, NewR, RightChanged),
243     deladjust(RightChanged, t(K,V,B,L,NewR),
244                right, NewTree, Changed).
```

Ugyanez, csak a legnagyobb kulcsú elemmel és jobboldali rekurzióval.

```
245 % del_assoc(+Key, +Assoc0, ?Value, -Assoc) [semidet]
246 del_assoc(Key, A0, Value, A) :-
247     delete(A0, Key, Value, A, _).
248
249 delete(t(Key,Val,B,L,R), K, V,
250        NewTree, WhatHasChanged) :-
251     compare(Rel, K, Key),
252     delete(Rel, t(Key,Val,B,L,R), K, V,
253            NewTree, WhatHasChanged).
254
255 delete(=, t(Key,Val,_B,t,R), Key, Val, R, yes) :- !.
256 delete(=, t(Key,Val,_B,L,t), Key, Val, L, yes) :- !.
257 delete(=, t(Key,Val,>,L,R), Key, Val,
258        NewTree, WhatHasChanged) :-
259     del_min_assoc(R, K, V, NewR, RightHasChanged),
260     deladjust(RightHasChanged, t(K,V,>,L,NewR),
261               right, NewTree, WhatHasChanged),
262     !.
263 delete(=, t(Key,Val,B,L,R), Key, Val,
264        NewTree, WhatHasChanged) :-
265     del_max_assoc(L, K, V, NewL, LeftHasChanged),
266     deladjust(LeftHasChanged, t(K,V,B,NewL,R),
267               left, NewTree, WhatHasChanged),
268     !.
269
270 delete(<, t(Key,Val,B,L,R), K, V,
```

```
271         NewTree, WhatHasChanged) :-
272     delete(L, K, V, NewL, LeftHasChanged),
273     deladjust(LeftHasChanged, t(Key,Val,B,NewL,R),
274                 left, NewTree, WhatHasChanged).
275 delete(>, t(Key,Val,B,L,R), K, V,
276         NewTree, WhatHasChanged) :-
277     delete(R, K, V, NewR, RightHasChanged),
278     deladjust(RightHasChanged, t(Key,Val,B,L,NewR),
279                 right, NewTree, WhatHasChanged).
```

Általános törlő operáció. Nézzük végig a `delete/6` egyes eseteit!

1. Keresett kulcs a gyökérben, baloldali részfa üres. Eredmény a jobboldali részfa.
2. Keresett kulcs a gyökérben, jobboldali részfa üres. Eredmény a baloldali részfa.
3. Keresett kulcs a gyökérben, és a jobboldali részfa mélyebb. Ekkor a jobboldali részfából kiveszi a legkisebb kulcsú elemet, és berakja a gyökérbe, majd kiegyensúlyozza a fát.
4. Keresett kulcs a gyökérben, és a jobboldali részfa nem mélyebb. Ekkor a baloldali részfából kiveszi a legnagyobb kulcsú elemet, és berakja a gyökérbe, majd kiegyensúlyozza a fát.
5. Keresett kulcs kisebb a gyökér kulcsánál. Rekurzív törlés a bal részfában, utána kiegyensúlyozás.
6. Keresett kulcs nagyobb a gyökér kulcsánál. Rekurzív törlés a jobb részfában, utána kiegyensúlyozás.

```
deladjust(no, OldTree, _, OldTree, no).
deladjust(yes, t(Key,Val,B0,L,R), LoR,
          NewTree, RealChange) :-
    deltable(B0, LoR, B1,
             WhatHasChanged, ToBeRebalanced),
    rebalance(ToBeRebalanced, t(Key,Val,B0,L,R), B1,
              NewTree, WhatHasChanged, RealChange).

deltable(-, right, <, no , no ) :- !.
deltable(-, left , >, no , no ) :- !.
deltable(<, right, -, yes, yes) :- !.
deltable(<, left , -, yes, no ) :- !.
deltable(>, right, -, yes, no ) :- !.
deltable(>, left , -, yes, yes) :- !.
```

Teljesen hasonló a beszúrás utáni `adjust` szabályhoz, csak törlés után.

```
rebalance(no, t(K,V,_,L,R), B, t(K,V,B,L,R),
          Changed, Changed).
rebalance(yes, OldTree, _, NewTree,
          _, RealChange) :-
    avl_geq(OldTree, NewTree, RealChange).
```

Elérkeztünk az AVL-fák lelkéhez, a forgató `rebalance` szabályhoz. Az első argumentum azt mondja meg, hogy szükség van-e forgatásra. Ha ez `no`, akkor a fa lényegileg nem változik, csak az egyensúly-szimbólumot állítja be a harmadik argumentumban kapott értékre (ami a megfelelő táblázatból lett kiolvasva). Az utolsó argumentum azt mutatja, hogy a fa mélysége

megváltozott-e, és ha nem történik forgatás, akkor csak átmásolja a beszúrás/törlés során megállapított értéket.

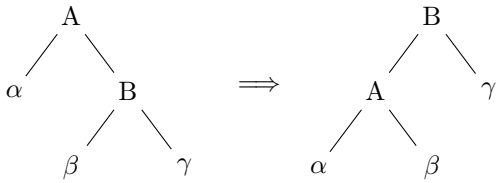
A tényleges forgatást az `avl_geq` végzi, aminek mindössze 3 argumentuma van: a régi fa, a forgatás után keletkező új fa, és hogy a forgatás során változott-e a fa mélysége.

```
299 avl_geq(t(A,VA,>,Alpha,t(B,VB,>,Beta,Gamma)),
300         t(B,VB,-,t(A,VA,-,Alpha,Beta),Gamma),
301         yes) :- !.
302 avl_geq(t(A,VA,>,Alpha,t(B,VB,-,Beta,Gamma)),
303         t(B,VB,<,t(A,VA,>,Alpha,Beta),Gamma),
304         no) :- !.
305 avl_geq(t(B,VB,<,t(A,VA,<,Alpha,Beta),Gamma),
306         t(A,VA,-,Alpha,t(B,VB,-,Beta,Gamma)),
307         yes) :- !.
308 avl_geq(t(B,VB,<,t(A,VA,-,Alpha,Beta),Gamma),
309         t(A,VA,>,Alpha,t(B,VB,<,Beta,Gamma)),
310         no) :- !.
311 avl_geq(t(A,VA,>,Alpha,
312             t(B,VB,<,t(X,VX,B1,Beta,Gamma),Delta)),
313         t(X,VX,-,t(A,VA,B2,Alpha,Beta),
314             t(B,VB,B3,Gamma,Delta)),
315         yes) :-
316     !,
317     table2(B1, B2, B3).
318 avl_geq(t(B,VB,<,
319             t(A,VA,>,Alpha,t(X,VX,B1,Beta,Gamma)),
320             Delta),
321         t(X,VX,-,t(A,VA,B2,Alpha,Beta),
```

```
322             t(B,VB,B3,Gamma,Delta)),
323         yes) :-
324     !,
325     table2(B1, B2, B3).
326
327 table2(< ,- ,> ).
328 table2(> ,< ,- ).
329 table2(- ,- ,- ).
```

Nézzük végig az egyes eseteket!

1. `>/>`: a jobboldali részfa túl mély, és annak a jobboldali részfája a mélyebb. A forgatás hatására a mélység csökken.

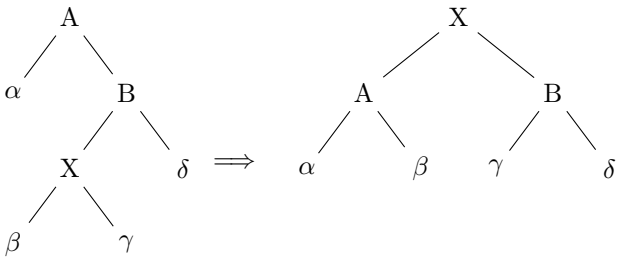


2. `>/-`: a jobboldali részfa túl mély, és az egyensúlyban van. Ez csak baloldali törlés után jöhet létre. A forgatás megegyezik az előzővel, de ilyenkor a mélység nem változik.

3. `</<`: az 1-es tükrözve.

4. `</-`: a 2-es tükrözve.

5. `>/<`: a jobboldali részfa túl mély, és annak a baloldali részfája a mélyebb. A forgatás hatására a mélység csökken. Az `A`-nál és `B`-nél levő egyensúly-szimbólumokat az `X`-nél levő alapján a `table2` táblázat adja meg.



6. `</>`: az 5-ös tükrözve (ilyen volt a fenti beszúrásos példa).

Itt érdemes megjegyezni, hogy a mélységcsökkenést jelző utolsó argumentumot csak a `deladjust` veszi figyelembe, az `adjust` nem. Ennek az az oka, hogy ha beszúráskor nőne a mélység, akkor a forgatás azt mindig kompenzálja, tehát végül az eredeti állapothoz képest a mélység nem változik. Ezzel szemben törlésnél ha forgatásra van szükség, akkor csak ettől függ, hogy a mélység csökken-e.

Ezzel vége a könyvtár forráskódjának. Ez egy szép, jól érthető program, ami kevés külső definíciót használ, ugyanakkor rendkívül tanulságos.

7.5. Projekt: prioritásos sor

Egy másik gyakran szükséges adatszerkezet a *prioritásos sor*, amiben minden elemhez egy prioritást rendelünk, és mindig a legmagasabb prioritásút vesszük ki belőle. A következő szabályokkal kezelhető:

- `üres_sor(?Sor)` [semidet]
- `sorba_tesz(+Sor, +Prioritás,?Elem, -Sor1)` [det]
- `maximum_elem(+Sor,?Elem)` [semidet]
- `maximumot_kivesz(+Sor, -Prioritás,?Elem, -Sor1)` [semidet]

A kényelmes használat kedvéért még érdemes listából/listává átváltó szabályokat is készíteni (ahol a lista `Prioritás-Elem` párokból áll):

- `listából_sor(+Lista, -Sor)` [det]
- `sorból_lista(+Sor, -Lista)` [det]

Ennek egy egyszerű megoldása az, hogy a betevés sorrendjében egy listában tároljuk az elemeket – ebben az esetben a maximum megkeresése ill. a sorból kivétel $O(n)$ műveletigényű lesz. Egy másik lehetőség, hogy az elemeket mindig csökkenő sorrendben tároljuk; ekkor az új elem betevése lesz $O(n)$-es komplexitású.

Egy hatékonyabb módszert kapunk, ha itt is egy fát használunk. Ebben a fában a csúcsokból nem csak kettő, hanem

több ág is indulhat, és csak azt várjuk el, hogy a csúcsban levő prioritás legalább akkora legyen, mint az alatta levő részfák maximum prioritása, így a maximum mindig a gyökérben lesz. Ezt „párosító kupacnak" nevezik.

Két ilyen fa egybeolvasztása nagyon egyszerű: megnézzük, hogy melyik gyökerének nagyobb a prioritása, és az marad a gyökér, a másik pedig ez alá kerül új részfaként. A beszúrás is ennek egy speciális esete, ahol a beszúrt elem egy olyan fa, aminek csak gyökere van.

Az egyetlen bonyolultabb – $O(\log n)$ komplexitású – művelet a maximális prioritású elem kivétele. Ilyenkor az összes alatta levő részfát egybe kell olvasztani, amíg csak egy nem marad. Ez sokféleképp megtehető, és a különböző módszerek más jellegű fákat eredményeznek. A klasszikus megoldás az, hogy a részfákat először balról jobbra páronként egybeolvasztjuk (innen a nevében a *párosító*), és utána a jobboldalitól elindulva a már egybeolvasztott párokat egyenként hozzáolvasztjuk. (Ez rekurzív módon nagyon egyszerűen megfogalmazható.)

Nézzünk egy példát! Jelölje `P-[P1,P2,..,Pn]` azt a fát, aminek a gyökerében `P`, az alatta levő részfák gyökerében pedig `Pi` prioritások vannak (a további, érdektelen részfákat `..` jelöli):

```
10-[5,8,2,8,10,1,3]
```

Maximumkivétel ⇒ 7 különálló fa:

```
5-[..] 8-[..] 2-[..] 8-[..] 10-[..] 1-[..] 3-[..]
```

Párosítás balról jobbra ⇒ 4 különálló fa:

```
8-[5,..] 8-[2,..] 10-[1,..] 3-[..]
```

Egyesítés jobbról balra, amíg csak 1 marad:

```
8-[5,..] 8-[2,..] 10-[3,1,..]

8-[5,..] 10-[8-[2,..],3,1,..]

10-[8-[5,..],8-[2,..],3,1,..]
```

50. Feladat. Készíts egy prioritásos sor adatszerkezetet párosító kupaccal!

Könyvajánló

Magyar nyelven nem sok minden jelent meg a Prologról, bár fontos megemlíteni a Szeredi Péter által fejlesztett MPROLOG (*Modular Prolog*) rendszerről szóló könyvet:

Zs. Farkas, I. Futó, T. Langer, P. Szeredi, *Mprolog programozási nyelv*, Műszaki Könyvkiadó, 1989.

Angolul már sokkal nagyobb a választék. Elsőként az Előszóban említett két könyvet ajánlanám: az *Art of Prolog* egy mélyebb, technikaibb tárgyalása a nyelvnek, míg a Bratko-féle tankönyv számos mesterséges intelligenciabeli alkalmazást mutat be.

A programozási készség elsajátításához a legfontosabb a gyakorlás; rengeteg érdekes feladat található (megoldásokkal!) az alábbiakban:

H. Coelho, J. C. Cotta, *Prolog by Example—How to Learn, Teach and Use It*, Springer, 1988.

B. Demoen, Ph-L. Nguyen, T. Schrijvers, R. Tronçon, *The First 10 Prolog Programming Contests*, Belgium, 2005.

Hatékony programok írásához praktikus tanácsokkal szolgál ez a könyv:

R. A. O'Keefe, *The Craft of Prolog*, MIT Press, 1990.

Végül a beépített szabályokról, fájlkezelésről és sok más hasznos dologról egy jó referencia az ISO szabvány informális leírása:

P. Deransart, A. Ed-Dbali, L. Cervoni, *Prolog: The Standard*, Springer, 1996.

Bár a Prolog nyelvhez nem kapcsolódik közvetlenül, de egy rendkívül olvasmányos és változatos áttekintést ad a számítástechnikáról a Turing Omnibus:

A. K. Dewdney, *The (New) Turing Omnibus*, Freeman / Holt, 1993.

Végül pedig azoknak, akik komolyabban érdeklődnek a téma iránt, egy jó kiindulási pont a *SICP* avagy a „varázslós könyv":

H. Abelson, G. J. Sussman, J. Sussman, *Structure and Interpretation of Computer Programs*. MIT Press, 1996.

A feladatok megoldásai

1. feladat

```
?- szülő(huszajn, X).
false

?- szülő(X, huszajn).
X = ali ;
X = fátima

?- szülő(ámna, X), szülő(X, fátima).
X = mohamed

?- szülő(ámna, X), szülő(X, Y), szülő(Y, haszan).
X = mohamed,
Y = fátima
```

2. feladat

```
?- szülő(X, ali).
?- szülő(umáma, X).
?- szülő(X, zajnab), szülő(Nagyszülő, X).
```

4. feladat

```
boldog(X) :- vangyereke(X).
kétgyerekes(X) :- szülő(X, Y), fivér(Y, _).
```

5. feladat

```
unoka(X, Y) :- nagyszülő(Y, X).
```

6. feladat

```
nővér(X, Y) :-
    nő(X),
    szülő(Z, X), szülő(Z, Y),
    X \= Y.
nagynéni(X, Y) :- nővér(X, Z), szülő(Z, Y).
```

```
?- nagynéni(zajnab, X), férfi(X).
```

7. feladat

Jó a definíció; X őse Z-nek, ha (i) X szülője Z-nek, vagy (ii) X őse Z szülőjének.

8. feladat

Változó; atom; atom; változó; atom; struktúra; szám; (hibás); struktúra; (hibás).

9. feladat

Erre a feladatra nincs *egyetlenegy* jó megoldás; különböző alkalmazásokhoz más és más leírási módok lehetnek kényelmesebbek.

- Egy tengelyekkel párhuzamos téglalapot le lehet (pl.) írni a bal felső és jobb alsó sarkával, tehát `téglalap(BalFelső, JobbAlsó)`, ahol `BalFelső` és `JobbAlsó` pontok: `pont(X, Y)`. Ha a téglalap a tengelyekkel nem párhuzamos, akkor a legegyszerűbb talán mind a négy csúcspontot felsorolni (bár ez redundáns). Egy másik lehetőség, hogy egy irányított szakasszal és egy hosszal írjuk le: `téglalap( szakasz(P1,P2),Hossz)`; ez alapján a téglalap úgy szerkeszthető meg, hogy a szakasz lerajzolása után derékszögben balra fordulunk, és onnan felmérjük a hosszt, a negyedik csúcs pedig már adódik.

- Egy négyzetet mindig le lehet írni a bal felső és jobb alsó sarokkal, de lehet pl. a középponttal és egy csúcsponttal is; tengelyekkel párhuzamos esetben elég a középpont és a csúcsok ettől való távolsága is.

- Egy kört is reprezentálhat a befoglaló négyzete, de megadható a középpontja és a sugara által is: `kör(Középpont, Sugár)`, pl. `kör(pont(1,2),3)`.

10. feladat

```
vízszintes(szakasz(pont(_,Y),pont(_,Y))).
```

```
?- vízszintes(S), függőleges(S).
S = szakasz(pont(_X,_Y),pont(_X,_Y)).
```

Tehát a ponttá degenerált szakasz ilyen.

11. feladat

Igen (`A=1,B=2`); nem; nem; igen (`D=2,E=2`); nem; igen (`P1=pont(-1,0),P2=pont(1,0),P3=pont(0,Y)`).

12. feladat

A megoldás függ a választott reprezentációtól. Ha a négy csúcsponttal írtuk le:

```
tengelytéglalap(téglalap(A,B,_,_)) :-
    függőleges(szakasz(P1,P2)).
tengelytéglalap(téglalap(_,B,C,_)) :-
    függőleges(szakasz(P2,P3)).
```

Ha az irányított szakasz és hossz megoldást választottuk:

```
tengelytéglalap(téglalap(S,_)) :- függőleges(S).
tengelytéglalap(téglalap(S,_)) :- vízszintes(S).
```

Természetesen sok más megoldás is elképzelhető.

13. feladat

```
1 kiolvas(1, egy).
2 kiolvas(2, kettő).
3 kiolvas(3, három).
```

14. feladat

```
?- f(s(1), A).
A = kettő.
```

Csak a második szabállyal egyesíthető.

```
?- f(s(s(1)), kettő).
false.
```

Nincsen egyesíthető szabály.

```
?- f(s(s(s(s(s(s(1)))))), C).
C = egy.
```

Nézzük ezt meg lépésenként. Az eredeti kérdés csak a negyedik szabállyal egyesíthető; egyesítés után az

```
?- f(s(s(s(1))), C).
```

kérdést kell megválaszolni, ami ismét a negyedik szabállyal egyesíthető, tehát:

```
?- f(1, C).
```

Ez pedig az első szabály alapján adja az eredményt.

```
?- f(D, három).
D = s(s(1)) ;
D = s(s(s(s(s(1))))) ;
D = s(s(s(s(s(s(s(s(s(1)))))))))) ;
...
```

A harmadik szabály adja az első megoldást. A negyedik szabály szerint, ha

```
?- f(X, három).
```

teljesül, akkor `D = s(s(s(X)))`, ez alapján minden olyan `D` megoldás lesz, ahol $3n + 2$ db. `s` szerepel.

15. feladat

```
trace, nagy(X), sötét(X), notrace.
Call: nagy(_5078)
Exit: nagy(medve)
Call: sötét(medve)
  Call: fekete(medve)
  Fail: fekete(medve)
Redo: sötét(medve)
  Call: barna(medve)
  Exit: barna(medve)
Exit: sötét(medve)
X = medve
```

16. feladat

A nyomkövetés a `fekete(X)` kielégítésétől indul. A `notrace` hozzáadásával a nyomkövetés csak addig tart, amíg el nem jut a `sötét` második szabályához:

```
?- sötét(X), nagy(X).
  Call: fekete(_4284)
  Exit: fekete(macska)
Exit: sötét(macska)
Call: nagy(macska)
Fail: nagy(macska)
Redo: sötét(_4284)
X = medve.
```

17. feladat

Amikor további bizonyítást keresünk, a második szabály alapján Fátimának egy másik ősét (nem Mohamedet) fogja keresni, akiről aztán majd be akarja látni, hogy Abdulla gyermeke.

Ekkor azonban eljutunk a második szabályhoz úgy, hogy az `X` ismeretlen, és a szabály szerint az

```
?- ős3(X, fátima).
```

kérdés megválaszolásához először meg kell válaszolni az

```
?- ős3(Y, fátima).
```

kérdést – ez viszont nyilvánvalóan egy végtelen rekurzió.

18. feladat

```
kivesz3(X, Y) :- hozzáfűz(Y, [_,_,_], X).
```

19. feladat

```
kivesz33(X, Y) :- kivesz3(X, [_,_,_|Y]).
```

20. feladat

Hozzáfűzéssel:

```
utolsó(X,Y) :- hozzáfűz(_,[Y],X).
```

Anélkül:

```
utolsó(X, [X]).
utolsó(X, [_|M]) :- utolsó(X, M).
```

21. feladat

```
páros_hosszú([]).
páros_hosszú([_|M]) :- páratlan_hosszú(M).
páratlan_hosszú([_|M]) :- páros_hosszú(M).
```

22. feladat

```
fordított([], []).
fordított([X|M], F) :-
    fordított(M, L), hozzáfűz(L, [X], F).
```

23. feladat

```
palindróma(X) :- fordított(X, X).
```

24. feladat

```
forgat([X|M], Y) :- hozzáfűz(M, [X], Y).
```

25. feladat

```
részhalmaz([], []).
részhalmaz([X|M], [X|R]) :- részhalmaz(M, R).
részhalmaz([_|M], R) :- részhalmaz(M, R).
```

26. feladat

```
ugyanolyan_hosszú([], []).
ugyanolyan_hosszú([_|M], [_|N]) :-
    ugyanolyan_hosszú(M, N).
```

27. feladat

```
lapít([], []).
lapít([X|M], Y) :-
    lapít(X, X1), lapít(M, M1),
    hozzáfűz(X1, M1, Y).
lapít(X, [X]) :- X \= [], X \= [_|_].
```

28. feladat

```
?- t(0+1, A).
A = 1+0.
?- t(0+1+1, B).
B = 1+1+0
?- t(1+0+1+1+1, C).
C = 1+1+1+1+0
?- t(D, 1+1+1+0).
D = 1+1+0+1 ;
D = 1+0+1+1 ;
D = 0+1+1+1
```

29. feladat

```
max(A, B, A) :- A >= B.
max(A, B, B) :- A < B.
```

30. feladat

```
maximum([X], X).
maximum([X,Y|M], Z) :-
    maximum([Y|M], X1), max(X, X1, Z).
```

31. feladat

```
összeg([], 0).
összeg([X|M], N):- összeg(M, N1), N is N1 + X.
```

32. feladat

```
növekvő([_]).
növekvő([A,B|M]) :- A < B, növekvő([B|M]).
```

33. feladat

```
részösszeg(L, N, X) :-
    részhalmaz(L, X), összeg(X, N).
```

34. feladat

```
között(A, B, A) :- A =< B.
között(A, B, X) :-
    A < B, A1 is A + 1,
    között(A1, B, X).
```

35. feladat

```
:- op(600, xfx, :=).
:- op(700, xfx, egyébként).
:- op(800, xfx, akkor).
:- op(900, fx, ha).
ha X > Y akkor Z := V egyébként _ :- X > Y, Z = V.
ha X > Y akkor _ egyébként Z := V :- X =< Y, Z = V.
```

36. feladat

```
?- p(X).
X = 1 ;
X = 2
?- p(X), p(Y).
X = 1, Y = 1 ;
X = 1, Y = 2 ;
X = 2, Y = 1 ;
X = 2, Y = 2
?- p(X), !, p(Y).
X = 1, Y = 1 ;
X = 1, Y = 2
```

37. feladat

```
szétoszt(_, [], [], []) :- !.
szétoszt(X, [Y|M], K, [Y|N]) :-
    X =< Y, !, szétoszt(X, M, K, N).
szétoszt(X, [Y|M], [Y|K], N) :-
    X > Y, !, szétoszt(X, M, K, N).
```

38. feladat

```
?- tartalmaz(X, Jelöltek),
   \+ tartalmaz(X, Kiesettek).
```

39. feladat

```
különbség([], _, []).
különbség([X|M], Y, Z) :-
    tartalmaz(X, Y), !, különbség(M, Y, Z).
különbség([X|M], Y, [X|Z]) :- különbség(M, Y, Z).
```

40. feladat

```
egyesíthető([], _, []) :- !.
egyesíthető([X|M], Y, L) :-
    \+(X = Y), !, egyesíthető(M, Y, L).
egyesíthető([X|M], Y, [X|L]) :-
    egyesíthető(M, Y, L).
```

Ha tagadás nélkül ellenőriznénk az egyesíthetőséget, akkor el is végezné az egyesítést:

```
egyesíthető_rossz([], _, []) :- !.
egyesíthető_rossz([X|M], Y, [X|L]) :-
    X = Y, !, egyesíthető_rossz(M, Y, L).
egyesíthető_rossz([X|M], Y, L) :-
    egyesíthető_rossz(M, Y, L).
```

```
?- egyesíthető_rossz([X,b,t(Y)], t(a), L).
X = t(a),
Y = a,
L = [t(a), t(a)].
```

41. feladat

```
katamino(M, Ak) :-
    hossz(Ak, H), N is H * 5 // M,
    H * 5 =:= N * M,
    kirak(N-M, Ak, X), kiír(N-M, X).

kirak(N-M, Ak, X) :- kirak(Ak, N-M, [], X).

kirak([], _, T, T).
kirak(Ak, N-M, T, X) :-
    első_lyuk(N-M, T, P),
    töröl(A, Ak, Ak1),
    letesz(A, P, N-M, T, T1),
    kirak(Ak1, N-M, T1, X).

első_lyuk(N-M, T, X-Y) :-
    között(1, N, X), között(1, M, Y),
    \+ tartalmaz(h(X-Y,_), T), !.

letesz(A, X-Y, N-M, T, T1) :-
    alakzat(A, [_-Dy|Dk]),
    Y1 is Y - Dy, Y1 > 0,
    letesz(A, X-Y1, Dk, N-M, [h(X-Y,A)|T], T1).

letesz(_, _, [], _, T, T).
letesz(A, X-Y, [Dx-Dy|Dk], N-M, T, T1) :-
    X1 is X + Dx, között(1, N, X1),
    Y1 is Y + Dy, között(1, M, Y1),
```

```
28      \+ tartalmaz(h(X1-Y1,_), T),
29      letesz(A, X-Y, Dk, N-M, [h(X1-Y1,A)|T], T1).
30
31  kiír(N-M, T) :- kiír(N-M, 1-M, T).
32
33  kiír(_, _-0, _).
34  kiír(N-M, X-Y, T) :-
35      Y =< M, X > N, Y1 is Y - 1, nl,
36      kiír(N-M, 1-Y1, T).
37  kiír(N-M, X-Y, T) :-
38      Y =< M, X =< N,
39      tartalmaz(h(X-Y,A), T),
40      write(A), write(' '),
41      X1 is X + 1,
42      kiír(N-M, X1-Y, T).
```

(A program további része változatlan.)

```
?- katamino(3, [k,p,c,w,l,y,i,r,v,z,+,t]).
c c + i i i i i k k k r t w y y y y z v
c + + + p p l k k r r r t w w y z z z v
c c + p p p l l l l r t t t w w z v v v
true

?- katamino(4, [k,p,c,w,l,y,i,r,v,z,+,t]).
p p z v v v c c c w w t t t y
p p z z z v c r c + w w t y y
p k k k z v r r + + + w t l y
k k i i i i i r r + l l l l y
true
```

```
?- katamino(5, [k,p,c,w,l,y,i,r,v,z,+,t]).
c c c r l l l l y y y y
c k c r r w t l + y z z
k k r r w w t + + + z v
k p p w w t t t + z z v
k p p p i i i i i v v v
true

?- katamino(6, [k,p,c,w,l,y,i,r,v,z,+,t]).
p p y y y y z v v v
p p l l y + z z z v
p k l w + + + r z v
k k l w w + r r r t
k c l c w w r t t t
k c c c i i i i i t
true
```

42. feladat

```
1 lapít(X, Y) :- lapít_kl(X, Y-[]).
2
3 lapít_kl([], Y-Y).
4 lapít_kl([X|M], Y-A) :-
5     lapít_kl(M, A1-A), lapít_kl(X, Y-A1).
6 lapít_kl(X, [X|A]-A) :- X \= [], X \= [_|_].
```

43. feladat

```
holland(X, PFK) :- holland_kl(X, PFK-FK, FK-K, K-[]).

holland_kl([], P-P, F-F, K-K).
holland_kl([piros(X)|M], [piros(X)|P]-P1, F, K) :-
    holland_kl(M, P-P1, F, K).
holland_kl([fehér(X)|M], P, [fehér(X)|F]-F1, K) :-
    holland_kl(M, P, F-F1, K).
holland_kl([kék(X)|M], P, F, [kék(X)|K]-K1) :-
    holland_kl(M, P, F, K-K1).
```

44. feladat

A programban a `k`-végű változónevek listákat jelölnek, például `Xk` („`X`-ek”).*

```
egyszerűsít(Kifejezés, Megoldás) :-
    változók(Kifejezés, Változók, Konstansok),
    számol(Változók, Számolt),
    összeg(Konstansok, N),
    felépít(Számolt, N, Megoldás).

összeg([], 0).
összeg([X|Xk], N):- összeg(Xk, N1), N is N1 + X.

felépít([], N, N) :- !.
felépít(Számolt, N, Megoldás) :-
```

*Angolul erre az `s` végződést használják.

```
12      felépít(Számolt, Érték),
13      ( N =:= 0, !, Megoldás = Érték
14      ; N =\= 0, Megoldás = Érték + N
15      ).
16
17  % Felépíti a kifejezés szimbolikus részét
18  felépít([v(X,1)], X) :- !.
19  felépít([v(X,K)], K*X) :- K > 1, !.
20  felépít([V1,V2|Vk], E2+E1) :-
21      felépít([V1], E1),
22      felépít([V2|Vk], E2).
23
24  % változók(+Kifejezés, -Változók, -Konstansok)
25  %   Szétválogatja a változókat és konstansokat.
26  változók(X, [X], []) :- atom(X), !.
27  változók(X, [], [X]) :- number(X), !.
28  változók(Xk+X, [X|Vk], Kk) :-
29      atom(X), !, változók(Xk, Vk, Kk).
30  változók(Xk+X, Vk, [X|Kk]) :-
31      number(X), változók(Xk, Vk, Kk).
32
33  % Megszámolja a változók előfordulásait,
34  % pl. [a,a,b,a,c,b] => [v(a,3),v(b,2),v(c,1)]
35  számol([], []).
36  számol([X|Xk], [v(X,N)|Zk]) :-
37      számol_és_töröl(X, Xk, N, Yk),
38      számol(Yk, Zk).
39
40  % számol_és_töröl(+X, +L, -N, -L1)
```

```
41 %   Megszámolja az X előfordulásait L-ben (N),
42 %   és kitörli őket, ennek eredménye az L1.
43 számol_és_töröl(_, [], 1, []) :- !.
44 számol_és_töröl(X, [X|Yk], N, Zk) :-
45     !, számol_és_töröl(X, Yk, N1, Zk),
46     N is N1 + 1.
47 számol_és_töröl(X, [Y|Yk], N, [Y|Zk]) :-
48     X \= Y, számol_és_töröl(X, Yk, N, Zk).
```

45. feladat

```
1 bagof(C, részhalmaz(X, C), L).
```

46. feladat†

Elég az alábbi szabályokat megváltoztatni.

```
1  % A második szabály egy új kiértékel/3 szabályt hív
2  játék(Állás, Játékos) :-
3      játék_vége(Állás, Játékos, Eredmény), !,
4      bejelent(Eredmény).
5  játék(Állás, Játékos) :-
6      lépést_választ(Állás, Játékos, Lépés),
7      kiértékel(Lépés, Állás, Játékos).
8
9  kiértékel(kilépés, _, _) :-
10     kiír('Viszlát!').
```

†Varga Csilla megoldása.

```
11 kiértékel(Lépés, Állás, Játékos) :-
12     érvényes(Lépés), !,
13     lép(Lépés, Állás, Állás1),
14     következő_játékos(Játékos, Játékos1), !,
15     kirajzol(Állás1, Játékos1),
16     játék(Állás1, Játékos1).
17 kiértékel(_, Állás, Játékos) :-
18     kiír('Érvénytelen lépés. Próbáld újra!'),
19     játék(Állás, Játékos).
20
21 % Itt nincs szükség az érvényesség ellenőrzésére
22 % (a lépés_választ/3 másik szabálya változatlan).
23 lépést_választ(_, ember, Lépés) :-
24     nl, kiír('Melyiket választod?'),
25     read(Lépés).
```

47. feladat[‡]

A trükk az, hogy az érvényesség ellenőrzése helyett ténylegesen elvégezzük a lépést, és ha sikerül, akkor jó, ha nem, akkor hibás. A változtatások az előző feladathoz képest értendők; csak a `kiértékel/3` második szabálya módosult.

```
1 kiértékel(Lépés, Állás, Játékos) :-
2     lépés(Állás, Lépés), !,
3     lép(Lépés, Állás, Állás1),
4     következő_játékos(Játékos, Játékos1), !,
```

[‡]Varga Csilla megoldása.

```
5       kirajzol(Állás1, Játékos1),
6       játék(Állás1, Játékos1).
```

48. feladat[§]

Itt a legtöbb problémát annak a kezelése okozza, hogy a kezdő lyukat az esetleges visszatérő vetés során ki kell hagyni. Ennek érdekében megváltoztatjuk a reprezentációt, és sorszámozzuk a lyukakat.

Szintén érdekes annak a biztosítása, hogy ha lehet, akkor hagyjunk követ az ellenfél oldalán. Ehhez megvizsgálunk minden lépéslehetőséget, és csak akkor hagyunk meg olyanokat, amelyek nem hagynak követ az ellenfélnek, ha nincsen más.

A változtatások az előző feladathoz képest értendők.

```
1   kövek_száma(4).
2
3   % A lyukak meg vannak sorszámozva.
4   alapbeállítás(Állás, ember) :-
5       kövek_száma(K),
6       Állás = tábla([1-K,2-K,3-K,4-K,5-K,6-K],0,
7                     [1-K,2-K,3-K,4-K,5-K,6-K],0).
8
9   % Hozzáad/levesz sorszámokat.
10  sorszámoz([K1,K2,K3,K4,K5,K6],
11            [1-K1,2-K2,3-K3,4-K4,5-K5,6-K6]).
12
```

[§]Varga Csilla megoldása alapján.

```
13 % Kezeli a sorszámokat.
14 kövek(I, tábla(L,_,_,_), K) :-
15     n_edik(I, L, _-K), K > 0.
16
17 % A táblákból ki kell venni a sorszámokat
18 % kirajzolás előtt.
19 kirajzol(tábla(La,Na,Lb,Nb)) :-
20     nl,
21     sorszámoz(La1, La),
22     fordított(La1, F),
23     sort_ír(F),
24     kalahot_ír(Na, Nb),
25     sorszámoz(Lb1, Lb),
26     sort_ír(Lb1).
27
28 % Figyel arra, hogy csak akkor hagyja üresen az
29 % ellenfél térfelét, ha nem tehet mást.
30 lépés(Állás, X) :-
31    findall(L, lehetőség(Állás, L), Lépések),
32    szétválaszt(Lépések, Üres, Nemüres),
33    ( Nemüres = [] -> tartalmaz(X-_, Üres)
34    ; tartalmaz(X-_, Nemüres)
35    ).
36
37 % Megfelel a régi lépés/2 szabálynak,
38 % de nincsen sosem a lépésnek folytatása,
39 % és nincsen szükség az üres tábla esetre sem.
40 lehetőség(Állás, L-Állás1) :-
41     tartalmaz(L, [1,2,3,4,5,6]),
```

```
42      kövek(L, Állás, _),
43      lép(L, Állás, Állás1).
44
45  % Különválogatja azokat a lépéseket, amelyek
46  % üresen hagyják az ellenfél térfelét azoktól,
47  % amelyek nem.
48  szétválaszt([], [], []).
49  szétválaszt([L-A|M], [L-A|Üres], Nemüres) :-
50      A = tábla(_,_,Lb,_), sorszámoz(Lb1, Lb),
51      üres(Lb1), !,
52      szétválaszt(M, Üres, Nemüres).
53  szétválaszt([L-A|M], Üres, [L-A|Nemüres]) :-
54      szétválaszt(M, Üres, Nemüres).
55
56  % A lépés nem lista, mert nincsen ismételt lépés.
57  lép(L, Állás, Állás2) :-
58      kövek(L, Állás, K),
59      vet(K, L, L, Állás, tábla(La,Na,Lb,Nb)),
60      söpör(tábla(La,Na,Lb,Nb), tábla(La1,Na1,Lb,Nb)),
61      megfordít(tábla(La1,Na1,Lb,Nb), Állás2).
62
63  % Ha kiürült az ellenfél oldala, minden a miénk.
64  söpör(tábla(La,Na,Lb,Nb), tábla(Lb,Na1,Lb,Nb)) :-
65      sorszámoz(Lb1, Lb),
66      üres(Lb1), !,
67      sorszámoz(La1, La),
68      összeg(La1, X),
69      Na1 is Na + X.
70  söpör(Y, Y).
```

```
71
72  % A kiinduló lyukat (KLuk) is megkapja,
73  % hogy azt ki tudjuk hagyni.
74  vet(Kövek, Luk, KLuk, Állás, Állás2) :-
75      vet_saját(Kövek, Luk, KLuk, Állás,
76                Állás1, Kövek1),
77      vet_ellenfél(Kövek1, KLuk, Állás1, Állás2).
78
79  % Amikor már nem először vetünk a saját oldalon,
80  % akkor a kihagyott kezdő lyuk miatt
81  % eggyel kevesebb kő kell.
82  vet_saját(Kövek, Luk, KLuk, tábla(La,Na,Lb,Nb),
83            tábla(La1,Na,Lb,Nb), Kövek1) :-
84      ( Luk = KLuk -> N = 6 ; N = 5 ),
85      Kövek > N - Luk, !,
86      felvesz_és_szór(Luk, KLuk, Kövek, La, La1),
87      Kövek1 is Kövek + Luk - N.
88  vet_saját(Kövek, Luk, KLuk, tábla(La,Na,Lb,Nb),
89            tábla(La1,Na,Lb,Nb), 0) :-
90      felvesz_és_szór(Luk, KLuk, Kövek, La, La1).
91
92  % Lényegében változatlan, csak kezeli a sorszámokat,
93  % és átadja a kiindulási lukat a szór/4 szabálynak.
94  felvesz_és_szór(0, KLuk, K, L, L1) :-
95      !, szór(K, KLuk, L, L1).
96  felvesz_és_szór(1, KLuk, K, [X-_|M], [X-0|M1]) :-
97      !, szór(K, KLuk, M, M1).
98  felvesz_és_szór(Luk, KLuk, K, [L|M], [L|M1]) :-
99      Luk > 1, !, Luk1 is Luk - 1,
```

```
100     felvesz_és_szór(Luk1, KLuk, K, M, M1).
101
102 % Figyel arra, hogy kihagyja a kiinduló lukat.
103 szór(0, _, L, L) :- !.
104 szór(N, KLuk, [X-L|M], [X-L|M1]) :-
105     N > 0, X = KLuk, !,
106     szór(N, KLuk, M, M1).
107 szór(N, KLuk, [X-L|M], [X-L1|M1]) :-
108     N > 0, X \= KLuk, !,
109     N1 is N - 1, L1 is L + 1,
110     szór(N1, KLuk, M, M1).
111 szór(_, _, [], []) :- !.
112
113 % Ha az ellenfél oldalán fejeződik be a vetés,
114 % akkor ellenőrzi, hogy elfogunk-e köveket.
115 vet_ellenfél(0, _, Állás, Állás) :- !.
116 vet_ellenfél(Kövek, _, tábla(La,Na,Lb,Nb),
117              tábla(La, Na2, Lb2, Nb)) :-
118     1 =< Kövek, Kövek =< 6,
119     szór_ellenfél(Kövek, Lb, Lb1),
120     elfog(Kövek, Lb1, Na, Lb2, Na2).
121 vet_ellenfél(Kövek, KLuk,
122              tábla(La,Na,Lb,Nb), Állás) :-
123     Kövek > 6, !, Kövek1 is Kövek - 6,
124     szór_ellenfél(6, Lb, Lb1),
125     vet(Kövek1, 0, KLuk, tábla(La,Na,Lb1,Nb), Állás).
126
127 % Ez megegyezik a régi szór/3 szabállyal,
128 % csak kezeli a sorszámokat is.
```

```
129 szór_ellenfél(0, L, L) :- !.
130 szór_ellenfél(N, [X-L|M], [X-L1|M1]) :-
131     N > 0, !,
132     N1 is N - 1, L1 is L + 1,
133     szór_ellenfél(N1, M, M1).
134 szór_ellenfél(_, [], []) :- !.
135
136 % Megfordítja az ellenfél köveinek listáját,
137 % hogy könnyen végig lehessen menni rajta
138 % rekurzióval, majd a végén a módosult listát
139 % visszafordítja.
140 elfog(Kövek, Lb1, Na, Lb2, Na2) :-
141     fordított(Lb1, Lb1f),
142     fog(Kövek, Lb1f, Na, Lb2f, Na2),
143     fordított(Lb2f, Lb2).
144
145 % Megkeresi a lyukat, ahová az utolsó kő esett,
146 % és ha ez nem 2 vagy 3, akkor nem történik semmi.
147 % Egyébként a saját kalahjába teszi, és megy tovább
148 % a lyukakon (visszafelé, mert a lyukak listája meg
149 % lett fordítva).
150 fog(_, [], Na, [], Na) :- !.
151 fog(Kövek, [N-X|M], Na, [N-X|M1], Na2) :-
152     N > Kövek, !, fog(Kövek, M, Na, M1, Na2).
153 fog(Kövek, [N-X|M], Na, [N-X|M], Na) :-
154     N = Kövek, ( X < 2 ; X > 3 ), !.
155 fog(Kövek, [N-X|M], Na, [N-0|M1], Na2) :-
156     N = Kövek, ( X =:= 2 ; X =:= 3 ),
157     Na1 is Na + X, Kövek1 is Kövek - 1,
```

```
158 |     fog(Kövek1, M, Na1, M1, Na2).
```

49. feladat

(Megoldás nélkül.)

50. feladat

Ld. az SWI-Prolog `heaps` modulját.

Tárgymutató