Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Deklaratív Programozás

Oktatási segédlet

Bevezetés a logikai programozásba

Szeredi Péter
Benkő Tamás

Számítástudományi
és Információelméleti Tanszék
IQSOFT Rt.

Budapest, 2001. március

Tartalomjegyzék

1. fejezet

Bevezetés

Ez a jegyzet a Deklaratív Programozás (korábban Programozási Paradigmák) tárgy logikai programozás részéhez készült oktatási segédanyag.

A logikai programozás (LP) alapgondolata, hogy programjainkat a (matematikai) logika nyelvén, állítások formájában írjuk meg. Míg a funkcionális nyelvek a matematikai függvényfogalomra, addig a logikai nyelvek a reláció fogalmára építenek. A legismertebb logikai programozási nyelv a Prolog (PROgramming in LOGic, azaz programozás logikában).

A logikai programozás ötlete Robert Kowalskitól származik [3]. Az első Prolog megvalósítást Alain Colmerauer csoportja készítete el a Marseille-i egyetemen 1972-ben [7]. A Prolog Magyarországon is hamar elterjedt, talán azért is mert igény volt egy ilyen magasszintű programozási nyelvre, és a funkcionális nyelveknek nem volt olyan kultúrája, mint pl. az Egyesült Államokban. Az 1975-ben Szeredi Péter által elkészített Prolog interpreter [5] felhasználásával több tucat, igaz többnyire kisérleti jellegű Prolog alkalmazás készült Magyarországon [8]. A Prolog hatékony megvalósítási módszereinek kidolgozása David H. D. Warren nevéhez fűződik, aki 1977-ben elkészítette a nyelv első fordítóprogramját (az ún. DEC-10 Prolog rendszert), majd 1983-ban kidolgozta a máig is legnépszerűbb megvalósítási modellt, a WAM-ot (Warren Abstract Machine) [9].

1981-ben a japán kormány egy nagyszabású számítástechnikai fejlesztési munkát indított el, az ún. „ötödik generációs számítógéprendszerek" projektet, amelynek alapjául a logikai programozást választották. Ez nagy lökést adott a terület kutató-fejlesztő munkáinak, és megjelentek a kereskedelmi Prolog megvalósítások is. Az 1980-as években Magyarországon is több kereskedelmi Prolog megvalósítás készült, az MProlog [1] és a CS-Prolog nyelvcsalád [2].

Bár a japán ötödik generációs projektben nem sikerült elérni a túlzottan ambiciózus célokat, és ez a 90-es évek elején a logikai programozás presztízsét is némileg megtépázta, mára a Prolog nyelv érett és világszerte elfogadott nyelvvé vált. 1995-ben megjelent a Prolog ISO szabványa is, és egyre több ipari alkalmazással is találkozhatunk.

Az elmúlt 10 évben a Prolog mellett újabb LP nyelvek is megjelentek, pl. az elsősorban nagyméretű, ipari alkalmazásokat megcélzó Mercury nyelv, továbbá a CLP (Constraint Logic Programming) nyelvcsalád, amely az operációkutatás ill. a mesterséges intelligencia eredményeit hasznosítva erősebb logikai következtetési mechanizmust biztosít.

A jegyzetben az ISO szabványt is támogató SICStus Prolog rendszert használjuk.[1] A jegyzet első felében bemutatott nyelvi elemek azonban mind olyanok, amelyek más, az ún. Edinburgh-i tradíciót követő megvalósításokban is mind megtalálhatók. A hallgatók rendelkezésére bocsátott SICStus Prolog mellett így gyakorlásra használható a szabadon terjeszthető SWI Prolog illetve a GNU Prolog is.

Ezeknek a megvalósításoknak a kézikönyvei elérhetők a világhálón, mint ahogy számos további információforrás is. Ezekről az 1.1 táblázat ad áttekintést.

A Prolog magyar nyelvű irodalma meglehetősen szerény, az MProlog rendszert ismertető [10] illetve a Pro-

[1] A SICStus kétféle üzemmódban használható: az ISO Prolog kompatibilis `iso` és a korábbi SICStus változattal kompatibilis `sicstus` módban; a két működési mód különbségeire a megfelelő helyeken felhívjuk az olvasó figyelmét.

SWI Prolog	`http://www.swi.psy.uva.nl/projects/SWI-Prolog/`
SICStus Prolog kézikönyv	`http://www.sics.se/ps/sicstus/sicstus_toc.html`
GNU Prolog	`http://pauillac.inria.fr/~diaz/gnu-prolog/`
The WWW Virtual Library: Logic Programming	`http://www.comlab.ox.ac.uk/archive/logic-prog.html`
CMU Prolog Repository	`http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/lang/prolog/0.html`
Prolog FAQ	`http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/faq/prolog.faq`
Prolog Resource Guide	`http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/faq/prg_1.faq` `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/faq/prg_2.faq`

1.1. táblázat: PROLOG INFORMÁCIÓ-FORRÁSOK

log eset-tanulmányokat tartalmazó [4] áll rendelkezésre. Egy rövid Prolog fejezet szerepel a Mesterséges Intelligencia c. monográfiában is [6].

A jegyzet felépítése

A jegyzet 2. fejezete rövid áttekintést ad a logikai ill. deklaratív programozás helyéről a különféle programozási irányzatok között. A 3. fejezet mutatja be a Prolog logikai programozási nyelv alapelemeit: ismerteti a nyelv szintaxisát, az adat- és program-struktúrákat, a végrehajtási mechanizmust. A 4. fejezet a Prolog nyelvhez kapcsolódó programozási módszereket tekinti át, valamint a legfontosabb beépített eljárások használatára mutat példákat.

Az 5. fejezet az ISO Prolog nyelv beépített eljárásait ismerteti, kézikönyv-szerűen. A 6. fejezetben a Prolog nyelv fejlettebb elemeit tárgyaljuk, a 7. fejezet egy nagyobb programpéldát, egy egyszerű fordítóprogramot mutat be, és végül a 8. fejezet a logikai programozás új, a Prolog nyelven túlmutató irányzatairól szól.

Az A függelék a Prolog nyelv fogalom-tárát tartalmazza. A jegyzetben közölt gyakorló feladatok megoldásai a B függelékben találhatók. A C függelék a logikai programozás kialakulásának hátterét és az automatikus tételbizonyítással való kapcsolatát ismerteti. Végül a D függelék a logikai programozás történetét tekinti át.

Jelölések

A jegyzetben a szintaxis-leírásokban BNF jelölést alkalmazunk a következő kiegészítésekkel:

<valami>`@` ... ::=	<valami>-k nem üres sorozata `@` jelekkel elválasztva,
{szöveg}	szöveges magyarázattal leírt szintaktikus elem

Köszönetnyilvánítás

Köszönet illeti Péter Lászlót, Szeredi Tamást és Visontai Mirkót a jegyzet LaTeX változatának elkészítésében végzett munkájukért.

Hibajelentés

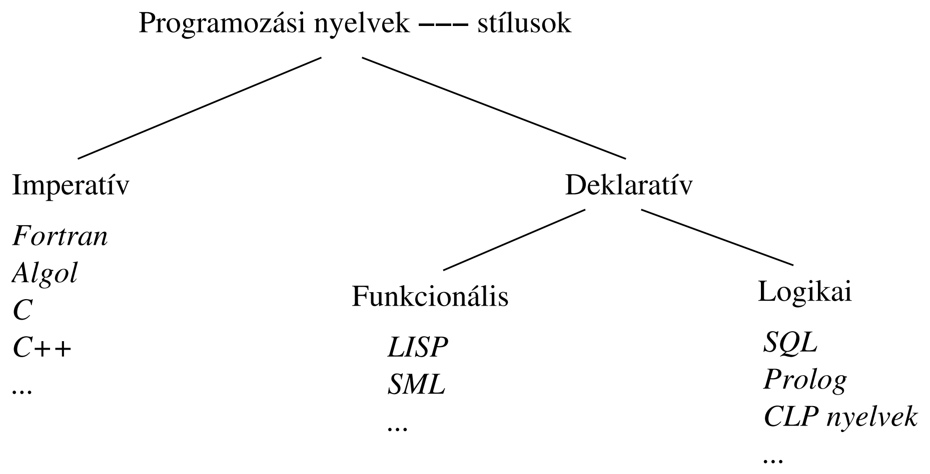
A szerző köszönettel fogad a jegyzettel kapcsolatos bármilyen észrevételt (sajtóhibákat és tartalmi megjegyzéseket egyaránt), a `szeredi@iqsoft.hu` email-címen.

2. fejezet

Deklaratív programozás — logikai programozás

Ez a fejezet röviden bemutatja a logikai, ill. a deklaratív programozás helyét a programozási nyelvek világában.

2.1. Programozási nyelvek osztályozása



2.1. ábra: A programozási nyelvek osztályozása

Mint a fenti ábra mutatja, a programozási nyelveket alapvetően két csoportba sorolhatjuk. A legtöbb nyelv az ún. *imperatív* nyelvek családjába tartozik: ezeket az jellemzi, hogy felszólító módban, parancsok segítségével írjuk le az elvégzendő feladatot. Ezzel szemben a *deklaratív* nyelvekben egyenleteket, állításokat írunk le, azaz alapvetően kijelentő módban programozunk. Míg egy imperatív nyelvű program esetén a hangsúly az algoritmuson van, azaz azon, hogy **hogyan** oldjuk meg a feladatot, addig egy deklaratív programban inkább magát a feladatot írjuk le, azaz azt, hogy **mit** kell megoldani. A deklaratív programozással kapcsolatban sűrün használt jelszó a „MIT és kevésbé HOGYAN" („WHAT rather than HOW"): a cél az, hogy a programozónak inkább azt kelljen leírnia, hogy MIT vár a programtól, és minél kevésbé azt, hogy HOGYAN kell ezt elérni.

Egy másik fontos különbség az imperatív és deklaratív programozási irányzatok között a változó-fogalomban mutatkozik meg. Az imperatív nyelvekben a változó egy adott memóriahelyen tárolt aktuális értéket jelent. Egy imperatív program „lényege" az, hogy egy változónak ismételten új és új értéket adunk. Ezzel szemben a deklaratív programozási nyelvek változói a matematika változó-fogalmának felelnek meg: egyetlen konkrét, bár a programírás idején még ismeretlen értéket jelölnek. A deklaratív nyelvekben nincs értékadás, egy `x=x+1` alakú programelem értelmetlen vagy hamis. Ezért szokás a deklaratív nyelveket az ún. *egyszeres értékadású* nyelvek közé sorolni.

A deklaratív programozási nyelvek általában valamilyen matematikai formalizmusra épülnek. A függvényfogalomra építő közelítésmódot *funkcionális* programozásnak, míg a reláció-fogalomra építőket *logikai* programozásnak nevezzük.

Az első funkcionális nyelv a LISP volt, amit az 1960-as évek elején alkottak meg. Ezt később több más nyelv követte, köztük az SML nyelv, amely a Deklaratív Programozás tárgy keretében oktatott funkcionális nyelv.

A legegyszerűbb logikai nyelvnek a relációs adatbázisok lekérdező nyelve, az SQL tekinthető. A „valódi" logikai nyelvek között a Prolog nyelv a legelterjedtebb, de újabban egyre nagyobb jelentőséggel bírnak a Prolog ún. korlát (constraint) alapú kiterjesztései, a CLP rendszerek (CLP = Constraint Logic Programming).

2.2. Egy egyszerű példaprogram

Ebben a fejezetben egy példa segítségével hasonlítjuk össze a különböző programozási irányzatokat. Példánk egy egyszerű adatbázis: adott gyermek–szülő kapcsolatok esetén meg kell határozni egy személy nagyszüleit. Példa-adatbázisunk a következő lesz:

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

C nyelvű megoldás

```
struct gysz {
  char *gyerek, *szulo;
} szulok[] = {
  "Imre", "István",
  "Imre", "Gizella",
  "István", "Géza",
  "István", "Sarolt",
  "Gizella", "Civakodó Henrik",
  "Gizella", "Burgundi Gizella",
  NULL, NULL
};

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

A fenti C programban az adatbázist egy struktúrákból álló tömbben tároljuk. Ezután definiáljuk a `nagyszuloi` függvényt, amely egy paraméterként kapott személy nagyszüleit írja ki. Vegyük észre, hogy az adatbázis bejárása egy kétszeresen egymásba skatulyázott ciklus segítségével történik.

Egy SML megoldás

```
fun szulo "Imre"    = ["István", "Gizella"]
  | szulo "István"  = ["Géza", "Sarolt"]
  | szulo "Gizella" = ["Civakodó Henrik",
                       "Burgundi Gizella"]
  | szulo _         = []

fun nagyszulok g = List.concat (map szulo (szulo g))
```

Az SML funkcionális nyelvű programban maga az adatbázis is egy függvény, amely a személyekhez a szüleik listáját rendeli (`[Elem1,Elem2, ...,ElemN]` egy `N` $\geq$ 0 elemű listát jelöl). Erre építve definiáljuk a `nagyszulok` függvényt, amely egy személyhez a nagyszülei listáját kell rendelje. SML-ben a függvény meghívását egyszerűen a függvény nevének és az argumentumnak az egymás után írásával jelöljük, pl. a `(szulo g)` a `szulo` függvényt hívja meg `g`-re. Kövessük nyomon a `nagyszulok "Imre"` kiértékelésében történő függvényhívásokat!

```
- szulo "Imre";
> val it = ["István", "Gizella"] : string list
```

Először a legmélyebb hívás történik meg, a `szulo "Imre"`. A > jellel kezdődő sor a rendszer válasza, vegyük észre, hogy a sor végén megjelenik az érték *típusa*, esetünkben `string list`, azaz füzérek listája.

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

A kapott eredményt a `map` függvényben használjuk, ez az első argumentumában kapott függvényt, esetünkben a `szulo`-t, alkalmazza a második argumentumában kapott lista minden elemére, és az így előálló értékekből képez listát. Példánkban az eredmény tehát egy olyan lista lesz, amelynek elemei listák: az első elem az apai nagyszülők, míg a második az anyai nagyszülők listája.

```
- List.concat (map szulo (szulo "Imre"));
> val it =
    ["Géza", "Sarolt", "Civakodó Henrik", "Burgundi Gizella"]
    : string list
```

A végrehajtás utolsó lépésében ezt a listát „lapositjuk” ki, a `List.concat` könyvtári függvény segítségével.

SQL megoldás

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

(...)

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

View created.
```

Az SQL (Structured Query Language) a relációs adatbázis-kezelők szabványos lekérdezési nyelve. Ebben lehetőség van ún. nézetek (view) létrehozására. A fenti példában a `nagyszulok` relációt olyan nézetként definiáljuk, amely két, a `szulok` relációra vonatkozó lekérdezést tartalmaz: `from szulok fiatal, szulok oreg`. Itt a `fiatal` ill. `oreg` jelzők a két szóbanforgó gyerek–szülő-pár megkülönböztetésére szolgál. A nézet létrehozásakor kikötjük, hogy a fiatal szülő legyen azonos az öreg gyerekkel: `where fiatal.szulo = oreg.gyerek`. Az SQL parancs első sorában írjuk elő, hogy a létrehozandó `nagyszulok` nézet-tábla első oszlopa tartalmazza a fiatal gyereket, mig a második az öreg szülőt: `create view nagyszulok as select fiatal.gyerek, oreg.szulo`.

A nézet definiálását követően a `nagyszulok` relációt ugyanúgy kérdezhetjük le, mint a tárolt relációkat:

```
SQL> select * from nagyszulok;

GYEREK                         SZULO
------------------------------ ------------------------------
Imre                           Civakodó Henrik
Imre                           Burgundi Gizella
Imre                           Géza
Imre                           Sarolt

SQL>
```

Prolog nyelvű megoldás

```
szülője('Imre', 'István').
szülője('Imre', 'Gizella').
szülője('István', 'Géza').
szülője('István', 'Sarolt').
szülője('Gizella', 'Civakodó Henrik').
szülője('Gizella', 'Burgundi Gizella').

nagyszülője(Gyerek, Nagyszülő) :-
    szülője(Gyerek, Szülő),
    szülője(Szülő, Nagyszülő).
```

A Prolog program ponttal lezárt állításokból épül fel. A fenti példában az első hat ún. tényállítás, azaz feltétel nélkül igaz állítás. Például, a legelső azt fejezi ki, hogy `'Imre'`-nek `szülője` `'István'`. Ezekkel a tényállításokkal tehát a gyerek–szülő adatbázisunkat írjuk le. Az utolsó állítás egy ún. szabály, amelynek jelentése:

> `Gyerek`-nek `nagyszülője` `Nagyszülő`, ha van olyan `Szülő`, hogy `Gyerek`-nek `szülője` `Szülő`, és `Szülő`-nek `szülője` `Nagyszülő`.

Itt `Gyerek`, `Szülő` és `Nagyszülő` Prolog változók, mivel nagybetűvel kezdődnek. (Az adatbázisban szereplő személyek neveit jelző névkonstansokat, pl. `'Imre'`-t azért kellett aposztrofok közé tenni, mert enélkül azokat is változónak tekintené a Prolog rendszer.)

A fenti Prolog programot például a következőképpen hívhatjuk meg:

```
| ?- nagyszülője('Imre', NSz).

NSz = 'Géza' ? ;
NSz = 'Sarolt' ? ;
NSz = 'Civakodó Henrik' ? ;
NSz = 'Burgundi Gizella' ? ;

no
```

A Prolog rendszer először az `NSz = 'Géza'` választ adja, majd az általunk begépelt `;` jel hatására ujabb megoldásokat mutat meg. A negyedik pontosvessző után megjelenő `no` válasz jelzi, hogy nincs több megoldás.

A nagyszülő reláció „visszafelé” is használható, azaz egy nagyszülő ismert unokáinak meghatározására:

```
| ?- nagyszülője(U, 'Géza').

U = 'Imre' ? ;

no
```

Reméljük, hogy az ismertetett példasor meggyőzte az olvasót a deklaratív programozás előnyeiről. Az SML és Prolog programok kódja illetve az SQL nézet-definició sokkal tömörebb, és könnyebben áttekinthető, ellenőrizhető mint a C nyelvű kód. Vegyük észre, hogy amit C-ben kétszeresen skatulyázott ciklussal kellett megvalósítanunk, az a logikai nyelvekben egy egyszerű és-kapcsolattá egyszerűsödött. Ugyanakkor a Prolog és SQL kód többre képes, mint amire terveztük, hiszen nemcsak egyes személyek nagyszüleit képes meghatározni, hanem visszafelé működtetve az unokák előállítására is használható.

2.3. A logikai programozás alapgondolata

A logikai programozás alapgondolata az, hogy a matematikai logika nyelvét, illetve egy a logikán alapuló nyelvet használjunk programozási nyelvként; végrehajtási módszerként pedig logikai következtetési ill. tételbizonyítási eszközöket használjunk.

A Prolog nyelv esetében az elsőrendű logikai nyelvét az ún. *Horn klózokra* szűkítjük, és egy nagyon egyszerű tételbizonyítási módszert használunk (lásd a C függeléket).

Eljárásos értelmezés

Ezek miatt az egyszerűsítések miatt a tételbizonyítási folyamat értelmezhető úgy is, mint logikai értéket adó eljáráshívások végrehajtása, ahol a paraméterek átadása mintaillesztésen alapul, és az eljárások meghiúsúlása ún. *visszalépést* eredményez.

Például a korábbi nagyszülője klóz

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

eljárásos értelmezése a következő:

- `Gy` és `N` formális paraméterek;
- `Sz` lokális változó;
- a `nagyszülője` eljárás végrehajtása abból áll, hogy az `szülője` eljárást kétszer egymás után meghívjuk, a megfelelő aktuális paraméterekkel. Ha a második hívás meghiúsul, akkor visszalépünk az elsőre, és megpróbálunk újabb megoldást keresni rá.

A Horn klózok, mint eljárások több különleges vonással rendelkeznek. Az eljáráshívások mindig egy logikai értéket adnak vissza (tehát valójában Boole-értékű függvények). Az igaz értékkel visszatérő eljárást sikeresnek, a hamissal visszatérőt meghiúsulónak nevezzük. Ha egy eljárás meghiúsul, akkor az adott eljárástörzs további eljárásait nem hajtjuk végre, ehelyett visszalépünk a legutoljára sikeresen lefutott eljáráshoz, és megpróbáljuk azt egy más módon (más változó-behelyettesítésekkel) sikeresen lefuttatni. Ennek sikere esetén az előremenő végrehajtás folytatódik, meghiúsulás esetén pedig újabb visszalépés történik.

Az eljárások paraméter-átvétele kétirányú mintaillesztéssel (egyesítéssel) történik. Ennek folytán a bemenő és kimenő paraméterek nincsenek megkülönböztetve, azaz ugyanaz az eljárás többféleképpen is használható. Például:

- `szülője('István', 'Géza')` — mindkét paraméter bemenő: igaz-e, hogy `'István'` szülője `'Géza'`?
- `szülője('István', Sz)` — első paraméter bemenő, a második kimenő: ki (kik) `'István'` szülője (szülei)?
- `szülője(Gy, 'István')` — első paraméter kimenő, a második bemenő: ki az, akinek `'István'` szülője (ki `'István'` gyermeke)?
- `szülője(Gy, Sz)` — mindkét paraméter kimenő: kik (az ismert) gyermek–szülő párok?

A Prolog két arca

Ha tételbizonyítóként vizsgáljuk a Prolog nyelvet, akkor szinte használhatatlannak találjuk, hiszen a logikai nyelv erősen megszorított és a tételbizonyítási módszer is túlságosan leegyszerűsített.

Tekintsük például az „őse” reláció rekurzív definicióját Prologban:

```
őse(Gy, Ős)  :-
    szülője(Gy, Ős).
őse(Gy, Ős)  :-
    szülője(Gy, Sz), őse(Sz, Ős).
```

Az első klóz jelentése: minden szülő egyben ős is, a másodiké pedig az, hogy a szülők őseit is ősöknek kell tekinteni. Tisztán logikai szempontból ennek a két állításnak a felírási sorrendje nyilván érdektelen, amint érdektelen a második állítás törzsében levő két feltétel sorrendje is. Mégis, ha ez utóbbi két feltételt felcseréljük, akkor a kapott Prolog program az összes megoldás felsorolása után végtelen ciklusba esik. Ha a két állítást is felcseréljük, rögtön végtelen ciklusba esik. (Egyes újabb logikai programozási nyelvek (CLP, Mercury) részben segítenek ezeken a problémákon.)

Ha programozási nyelvként vizsgáljuk a Prologot, akkor egy nagyon magasszintű nyelvnek találjuk, amelynek fő jellemzői:

- általánosított „mintaillesztés”
- visszalépéses eljárásszervezés
- tömör, „többször használható” eljárások

A Prolog különösen jól alkalmazható szimbolikus, keresési feladatokra, gyors prototípus-készítésre.

3. fejezet

A Prolog nyelv alapjai

Ez a fejezet a Prolog nyelv alapelemeit mutatja be. Az első alfejezet a Prolog programok szerkezetét írja le, a második a Prolog vezérlési mechanizmusát ismerteti, míg a harmadik az összetett adatfogalmat mutatja be. Ezt követi a Prolog nyelv szintaxisának pontos leírása, majd a fejezetet a Prolog egy lehetséges típusfogalmának ismertetése zárja.

3.1. A Prolog programok elemei

Míg a funkcionális nyelvek a matematikai függvényfogalomra, addig a logikai nyelvek a reláció (predikátum) fogalmára építenek. Ebben a szakaszban azt mutatjuk be, hogyan definiálhatunk predikátumokat Prologban.

3.1.1. Tényállítások

A legegyszerűbb Prolog állítások a relációs adatbázistáblák sorainak felelnek meg. Példaként tekintsük az alábbi ún. **tényállításokat**:

```
(1)       járat('Budapest', 'Prága', 515).
(2)       járat('Budapest', 'Bécs', 245).
(3)       járat('Bécs', 'Berlin', 635).
(4)       járat('Bécs', 'Párizs', 1265).
```

A tényállítások a definiált predikátum nevével (`járat`) kezdődnek, ezt követik a predikátum argumentumai, zárójelbe zárva és egymástól vesszőkkel elválasztva. A tényállítást, mint a Prolog nyelv minden más elemét, egy sorvégi pontjellel zárjuk le. (A baloldalon levő számok a későbbi hivatkozást szolgálják csak, nem részei a Prolog programnak.)

Az argumentumok, a most vizsgált egyszerű esetben lehetnek szám- vagy névkonstansok. A névkonstansok a hagyományos nyelvek szövegkonstans *(string)* fogalmához állnak közel, ezeket aposztróf-jelek (') közé írt tetszőleges jelsorozattal adhatjuk meg. Az aposztróf-jelek bizonyos esetekben elhagyhatók: ha a név kisbetűvel kezdődő alfanumerikus jelsorozat, vagy ha bizonyos írásjelek sorozata (pl. =<). Valójában a predikátum neve is egy névkonstans, de most nem kell aposztróf-jelek közé tenni, mivel kisbetűs jelsorozat.

A fenti első tényállítással pl. azt a tényt kívánhatjuk a Prolog rendszer tudomására hozni, hogy Budapest és Prága között közlekedik egy autóbuszjárat, amely által megtett út hossza 515 km. A Prolog rendszer számára viszont ez mindössze egy háromargumentumú predikátum, amely bizonyos megadott konstansok között fennáll. A programok olvashatósága érdekében tehát nagyon fontos, hogy az egyes argumentumok jelentését egy megjegyzésben írjuk le, például így:

```
% járat(város1, város2, út hossza km-ben).
```

A százalékjel Prologban egy kommentár kezdetét jelzi, a megjegyzés a sor végéig tart. A predikátum jelentését általánosabban egy olyan mondattal lehet megadni, amely az argumentumok közötti kapcsolatot írja le, pl. így:

```
% járat(A, B, T): Az A és B városok között van járat,
% melynek úthossza T km.
```

Egy ilyen megjegyzés az összetettebb Prolog predikátumok megértéséhez, helyességének eldöntéséhez nagy segítséget tud adni. Ezért minden példánkhoz írunk majd ilyen ún. **fejkommentárt**, és az olvasónak is javasoljuk ezt.

Nézzük most meg, milyen módon használhatjuk a fent definiált `járat` predikátumot! A Prolog megvalósítások többsége interaktív rendszer, azaz egy olyan környezetet ad, amelyben a felhasználó a program betöltése után interaktív módon tehet fel kérdéseket. A feltehető kérdés legegyszerűbb esete az, amikor egy konkrét állítás fennállását kérdezzük:

```
| ?- járat('Budapest', 'Bécs', 245).
yes
```

Az ilyen kérdésre igen vagy nem választ kapunk, a fenti esetben a válasz `yes`, azaz igen. (A `| ?-` jelsorozat a Prolog rendszer promptja, ezzel jelzi, hogy kérdést vár.)

Érdekesebb a kérdés, ha a kérdezett predikátum egyes argumentumaiként ún. **változókat** szerepeltetünk. A változókat Prologban nagybetűvel vagy aláhúzásjellel (_) kezdődő azonosítóval jelöljük:

```
| ?- járat('Budapest', 'Bécs', Táv).
Táv = 245 ?
```

Kérdésünk ilyenkor arra vonatkozik, hogy van-e a változóknak egy olyan behelyettesítése, amelyre a predikátum fennáll. Ehhez a rendszer keres egy olyan tényállítást, amely a kérdéssel illeszthető, azaz változók behelyettesítésével vele azonos alakra hozható. Válaszként, mint a fenti példában is látjuk, megkapjuk a változó(k) szükséges behelyettesítését. A további válaszokat (ha vannak) egy ';' karakter leütésével kaphatjuk meg, mint pl. az alábbi kérdésben:

```
| ?- járat('Bécs', Cél, Táv).
Cél = 'Berlin', Táv = 635 ? ;
Cél = 'Párizs', Táv = 1265 ? ;
no
```

Az utolsó sorban látható `no` jelzi, hogy nincs több válasz.

Most nézzünk egy összetett kérdést: keressünk adatbázisunkban egy olyan Budapestről kiinduló, két szakaszból álló útvonalat amelynek összhossza nagyobb 1000 km-nél.

```
| ?- járat('Budapest', Közben, T1),
     járat(Közben, Cél, T2), T1+T2 > 1000.
T1 = 245, T2 = 1265, Cél = 'Párizs', Közben = 'Bécs' ? ;
no
```

Az összetett kérdésben az egyes rész-kérdések közötti vessző és-kapcsolatot jelöl. Az első két rész-kérdés a `járat` predikátumra, míg a harmadik egy ún. **beépített predikátumra**, a `>` aritmetikai összehasonlító relációra vonatkozik. Ez a beépített predikátum olyan, hogy argumentumaiban aritmetikai kifejezéseket is elfogad.

Nézzük most meg, hogyan is hajtódik végre a fenti összetett kérdés! A Prolog végrehajtási mechanizmusa szerint először a legelső, legbaloldalibb részkérdést kell tekintenünk. Ehhez keresünk egy vele illeszthető tényállítást, példánkban rögtön az (1) tényállítás megfelel, a behelyettesítés:

```
Közben = 'Prága', T1 = 515
```

Ezután az első rész-kérdést elhagyjuk, és az illesztéshez szükséges változó-behelyettesítéseket a fennmaradó kérdésben elvégezzük. A példában az eredmény:

```
járat('Prága', Cél, T2), 515+T2 > 1000.
```

Az illesztés és behelyettesítést együtt redukciós lépésnek nevezzük, ez a Prolog végrehajtás alapeleme. A következő táblázat a most elvégzett redukciós lépés adatait foglalja össze:

```
rész-kérdés:         járat('Budapest', Közben, T1)
illesztett állítás: (1)
behelyettesítés:     Közben = 'Prága', T1 = 515
új kérdés:           járat('Prága', Cél, T2), 515+T2>1000.
```

Most az újonnan első helyre került rész-kérdésre próbálunk egy újabb redukciós lépést elvégezni. Ehhez azonban nem találunk vele illeszthető tényállítást, hiszen miniatűr adatbázisunk nem tartalmaz Prágából induló járatot. A Prolog végrehajtási mechanizmusa ezt az esetet a **visszalépés** segítségével kezeli: visszatér az előző redukciós lépésbeli állapothoz, példánkban az eredeti teljes kérdéshez. Ennek első tagjához keres újabb illeszthető tényállítást, az előző, sikeres illesztést követő állítások közül. Példánkban ez azt jelenti, hogy a (2) állítástól folytatja a keresést. Az illesztés most is azonnal sikerül, a behelyettesítés és az új kérdés a következő:

```
rész-kérdés:         járat('Budapest', Közben, T1)
illesztett állítás: (2)
behelyettesítés:     Közben = 'Bécs', T1 = 245,
új kérdés:           járat('Bécs', Cél, T2), 245+T2>1000.
```

A következő redukciós lépés is sikeres:

```
rész-kérdés:         járat('Bécs', Cél, T2)
illesztett állítás: (3)
behelyettesítés:     Cél = 'Berlin', T2 = 635
új kérdés:           245+635 > 1000.
```

Most jutottunk el a > beépített predikátumra vonatkozó kérdéshez. Ezt a rendszer úgy hajtja végre, mint egy logikai értéket adó eljárást: kiértékeli a benne szereplő kifejezéseket, és ha a baloldali számérték nagyobb a jobboldalinál, akkor igaz, ha nem, akkor hamis eredményt ad. Az igaz eredmény annak felel meg, mintha sikeresen illesztettünk volna egy tényállítással, tehát elhagyjuk a részkérdést (változó-behelyettesítés most nincs). Ha viszont hamis az eredmény, akkor ugyanúgy mintha nem találtunk volna illeszthető tényállítást, visszalépés történik.

Példánkban most egy hamis állításhoz jutottunk, tehát visszalépés történik a megelőző állapotba, ahol a (4) állítással folytatódik a keresés:

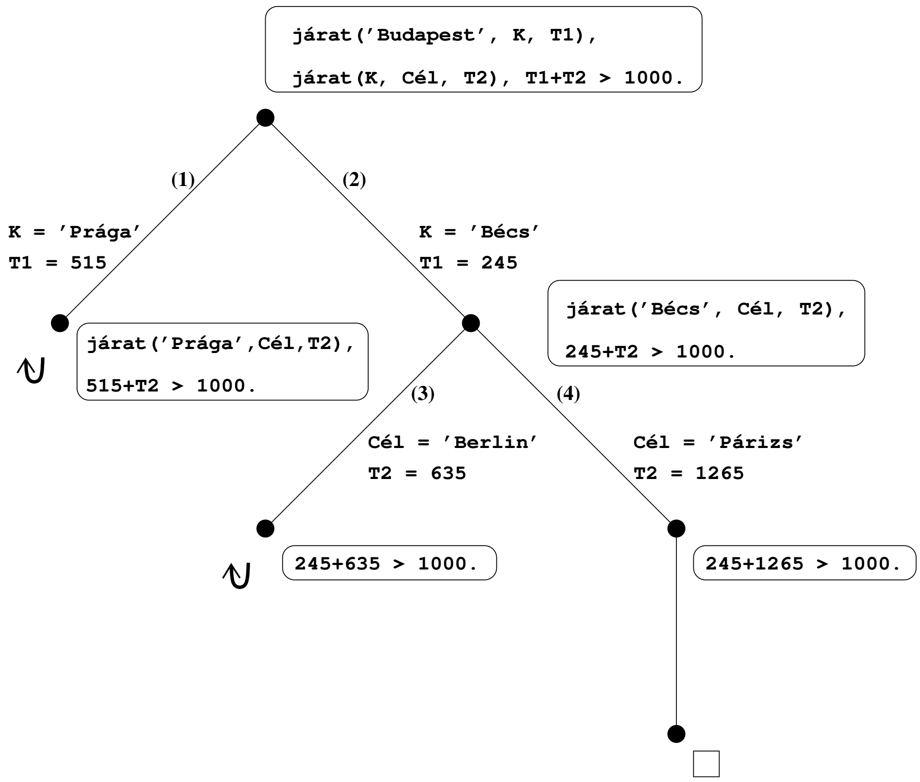
```
rész-kérdés:         járat('Bécs', Cél, T2)
illesztett állítás: (4)
behelyettesítés:     Cél = 'Párizs', T2 = 1265
új kérdés:           245+1265 > 1000.
```

Most a beépített > predikátum igaz értéket ad, ami által az utolsó rész-kérdést is elhagyhatjuk, tehát sikeresen megoldottuk a kérdést. A változó-behelyettesítések mostani állapota jelenti a kérdés megoldását, azaz:

```
Közben = 'Bécs', T1 = 245, Cél = 'Párizs', T2 = 1265.
```

Ha további megoldások keresésére kérjük a rendszert, akkor az visszalépéssel folytatja a keresést. Példánkban ez azt jelenti, hogy visszatér a legutolsó sikeres illesztett részkérdéshez, ahol a (4) állítással illesztettünk, és további illesztéseket keres. Mivel összesen négy állításunk van, így nem találhatunk újabb illesztést. Ez újabb visszalépést okoz, most már a legelső rész-kérdéshez, ahol is a (3)-tól kezdve keres illeszthető állítást. Miután itt sem talál újabb megoldást, megállapítja, hogy az eredeti kérdésnek nincs több megoldása, és kiírja a `no` választ.

A 3.1 ábra mutatja azt a keresési fát (VAGY-fát), amelyet a Prolog végrehajtási mechanizmusa bejár. A fa csomópontjait kérdésekkel címkéztük meg, a fa gyökerében a teljes eredeti kérdés áll. A fa éleit viszont az illesztett állítás sorszámával és az elvégzett változó-behelyettesítésekkel címkéztük meg. A fa levelei vagy visszalépési pontoknak felelnek meg, ezeket a zsákutca jele jelzi, vagy pedig megoldásoknak, amelyeket az üres kérdés-doboz címkéz meg. Vegyük észre, hogy a Prolog rendszer végrehajtási mechanizmusa nem más, mint egy mélységi keresés ezen a fán.



3.1. ábra: A KÉRDÉS KERESÉSI FÁJA

3.1.2. A Prolog rendszer használata

Mielőtt a Prolog nyelv elemeinek bemutatását folytatnánk, néhány praktikus, a Prolog rendszerek használatára vonatkozó tudnivalót adunk meg. Mi itt ténylegesen a SICStus Prolog rendszert mutatjuk be, de az elmondottak szinte minden Prolog rendszerre érvényesek.

A Prolog programot általában egy állományból töltjük be, erre a

`consult(`Állománynév`)` és `compile(`Állománynév`)`

beépített eljárások szolgálnak. Az előbbi kisebb helyigényű, de lassúbb kódot, míg az utóbbi terjedelmesebb, de gyorsabb kódot eredményez. A `consult(`Állománynév`)` helyett írható a rövidebb `[`Állománynév`]` alak is. Az állománynév, egy névkonstans, kiterjesztése alaphelyzetben `.pl`.

Álljon itt most egy teljes párbeszéd a Prolog rendszerrel, feltételezve, hogy a négy tényállításból álló példa-adatbázisunk a `jarat.pl` állományban van.

```
> sicstus
SICStus 3.8 (sparc-solaris-5.5.1): Thu Oct  7 14:58:41 MET DST 1999
| ?- consult(jarat).
{consulting /home/user/jozsi/jarat.pl...}
{/home/user/jozsi/jarat.pl consulted, 0 msec 688 bytes}

yes
| ?- járat('Bécs', Cél, Táv).

Cél = 'Berlin',
Táv = 635 ? ;

Cél = 'Párizs',
Táv = 1265 ? ;

no
| ?- járat(Honnan, 'Párizs', Táv).

Táv = 1265,
Honnan = 'Bécs' ?

yes
| ?- halt.
>
```

Az első kérdésnél az összes lehetséges választ elkértük a rendszertől. Ha nem kívánjuk, hogy a rendszer az összes választ felsorolja, akkor ezt úgy jelezhetjük, hogy a pontosvessző helyett újsort ütünk le. A második kérdés esetén ez történt, a rendszer ilyenkor egy `yes` választ ad. A Prolog rendszerből való kilépés a `halt` beépített eljárással történt. Ehelyett alkalmazható az operációsrendszer-függő állományvégjelző karakter is (`^D` Unix, ill. `^Z` DOS alatt).

Kivételes esetben a programot, vagy annak egy részét a konzolról is bevihetjük, a `consult(user)` beépített eljárás meghívásával. Például:

```
| ?- consult(user).
| végállomás('Budapest').
| végállomás('Párizs').
| end_of_file.
{user consulted, 0 msec 424 bytes}

yes
| ?-
```

A bevitel végét jelző `end_of_file.` kifejezés helyett itt is alkalmazható az állományvégjelző karakter.

Végül megjegyezzük, hogy a SICStus Prologot az Emacs szövegszerkesztő programhoz is illesztették. Egy Emacs puffernek vagy tartománynak a Prolog rendszerbe való bevitelére egy adott billentyű-kombináció is használható, a Prolog szerkesztési mód alkalmazása esetén (lásd a `prolog.el` Emacs könyvtárat).

3.1.3. Szabályok

Tegyük fel, hogy a fenti programunk továbbfejlesztéseként szeretnénk egy olyan predikátumot definiálni, amely `Kezdet` és `Cél` városok között keres összeköttetést, két kapcsolódó járat igénybevételével. Természetesen, az új predikátumot a korábbi `járat` predikátumra szeretnénk visszavezetni. Ezt a Prolog nyelv második állítás-fajtája, a **szabály** segítségével tehetjük meg:

```
% járat2(Kezdet, Cél, Táv): Kezdet városból vezet egy
%     két szakaszból álló,  Táv km hosszú út Cél-ba.
járat2(Kezdet, Cél, Táv) :-
    járat(Kezdet, Közben, T1),
    járat(Közben, Cél, T2),
    Táv is T1+T2.
```

A szabály egy ún. **fej**- és **törzs**-részből áll, amelyeket a `:-` jelkombináció választ el egymástól. A fej az előző szakaszban ismertetett tényállításhoz hasonló alakú, de az argumentumok változók is lehetnek. A törzs viszont a korábbi kérdéssel azonos szerkezetű. Példánkban a törzs utolsó tagjaként az `is` beépített predikátum áll: ez a jobboldalán levő aritmetikai kifejezést kiértékeli és az eredményt a baloldali változóba helyettesíti (pontosabban: azzal illeszti).

Vegyük észre, hogy a Prolog változó-fogalma nem azonos az algoritmikus nyelvek változó-fogalmával. Egy Prolog változó kezdetben nem bír értékkel, majd egy illesztés során helyettesítődik egy másik Prolog objektummal (eddigi példáinkban egy szám- vagy névkonstanssal). A behelyettesítés pillanatában a változó megszűnik létezni, minden rá való hivatkozás helyettesítődik a változó értékével. Egy már értékkel bíró változó nem módosítható: például az `X is X+1` hívás mindig sikertelen: a rendszer kiszámolja `X+1` értékét, majd megkísérli illeszteni `X`-szel, ami mindig meghiúsul. Egy változó csak úgy kaphat újabb értéket, ha a visszalépés során visszaáll az üres állapotba, és ezután egy másik ágon újra behelyettesítődik. Azoknak, akik már hozzászoktak ahhoz, hogy a programozás során változtatható cellákban gondolkozzanak, ez a közelítésmód valószínűleg szokatlan, de el kell ismerni, hogy logikailag egyszerűbb és tisztább.

A szabályokat első közelítésben makró-definíciónak tekinthetjük: ha a `járat2` predikátumra vonatkozó kérdésre kell válaszolnunk, akkor a kérdést a szabály törzsével kell helyettesítenünk, természetesen a megfelelő paraméter-behelyettesítés után. Például:

```
| ?- járat2('Budapest', Cél, T), T > 1000.
T = 1510, Cél = 'Párizs' ? ;
no
```

Itt a végrehajtás első lépése az, hogy a fenti `járat2` kérdést helyettesítjük a rá vonatkozó szabály törzsével, és ezután ezt az előző szakaszban leírt módon megválaszoljuk.

```
járat('Budapest', Közben, T1), járat(Közben, Cél, T2),
T is T1+T2, T > 1000.
```

Az út összhosszának vizsgálata itt két lépésben zajlik le (először kiszámoljuk a két rész-távolság összegét, majd külön műveletben vizsgáljuk, hogy ez 1000-nél nagyobb-e).

Egy szabályt, bár makróként vezettük be, lehet logikai állításként is értelmezni, mégpedig egy olyan implikációként, amelynek következménye a szabály feje, feltétele pedig a szabály törzse. Példánkban:

```
járat2(Kezdet, Cél, Táv) igaz ha
    járat(Kezdet, Közben, T1) igaz és
    járat(Közben, Cél, T2) igaz és
    Táv is T1+T2 igaz.
```

Ez a logikai állítás minden változó-behelyettesítésre igaz, tehát a benne szereplő változók univerzális kvantorral lekötöttnek tekintendők. Könnyen meggondolható, hogy egy olyan változó esetén, amely csak a törzsben szerepel, a külső univerzális kvantor helyettesíthető a törzsre vonatkozó egzisztenciális kvantorral. Példánkban ilyen a `Közben` változó:

```
tetszőleges Kezdet, Cél, Táv értékek esetén
```

```
járat2(Kezdet, Cél, Táv) igaz ha
    létezik olyan Közben érték, hogy
        járat(Kezdet, Közben, T1) igaz és
        járat(Közben, Cél, T2) igaz és
        Táv is T1+T2 igaz.
```

Bár a fenti formulában a logikai műveleteket magyar nyelven fogalmaztuk meg, a matematikai logikában jártasak számára nyilvánvaló, hogy az elsőrendű predikátumkalkulus egy formuláját írtuk fel. Ugyanakkor, ha a predikátum-hivatkozások helyébe azok fejkommentárjait helyettesítjük, akkor egy értelmes és igaz természetes nyelvű állítást kapunk:

```
tetszőleges Kezdet, Cél, Táv értékek esetén
Kezdet városból vezet egy két szakaszból álló,
Táv km hosszú út Cél-ba, ha
    létezik olyan Közben érték, hogy
        a Kezdet és Közben városok között van járat,
             melynek úthossza T1 km és
        a Közben és Cél városok között van járat,
             melynek úthossza T2 km és
        Táv = T1+T2.
```

(Az utolsó sorban kihasználtuk, hogy a `Táv is T1+T2` beépített predikátum jelentése `Táv = T1+T2`.)

3.1.4. Predikátum definiálása több szabállyal

Folytassuk most példánk kidolgozását: eddig nem foglalkoztunk azzal a kérdéssel, hogy a `járat` relációt célszerű szimmetrikusnak tekintenünk. Pontosabban: ha A városból van járat B-be, akkor célszerű lehet automatikusan úgy tekinteni, hogy B-ből A-ba is van, mégpedig azonos úthosszú járat. Definiáljunk tehát egy `járatszakasz` predikátumot a `járat` predikátum segítségével:

```
% járatszakasz(A, B, H): A és B között, vagy B és A
%    között van járat, amelynek úthossza H.
járatszakasz(Kezdet, Cél, Táv) :-
    járat(Kezdet, Cél, Táv).
járatszakasz(Kezdet, Cél, Táv) :-
    járat(Cél, Kezdet, Táv).
```

Tekintsük egy erre a predikátumra vonatkozó kérdést:

```
| ?- járatszakasz('Bécs', Hová, H).
H = 635, Hová = 'Berlin' ? ;
H = 1265, Hová = 'Párizs' ? ;
H = 245, Hová = 'Budapest' ? ;
no
```

Mint az elvárható, a rendszer először a `járatszakasz` predikátum első szabályát használja, erre kapjuk az első két választ. Ezután, amikor a `járat('Bécs', Cél, Táv)` kérdésre nem talál több választ, alkalmazza a `járatszakasz` predikátum második szabályát, ami által a `járat(Cél, 'Bécs', Táv)` kérdéshez, és így az utolsó megoldáshoz jut.

Vegyük észre, hogy a Prolog a szabályok esetén ugyanolyan visszalépéses keresést valósít, meg mint a tényállításoknál. Valójában a nyelv nem tesz különbséget a két állítástípus között, a tényállítást üres törzsű szabálynak tekinti. Ez azt is jelenti, hogy a tényállítás és a szabály feje szintaktikusan azonos, mindkettő argumentumaiban változók és konstansok egyaránt előfordulhatnak. Arra is van lehetőség, hogy ugyanazon predikátum definiálására tényállításokat és szabályokat vegyesen alkalmazzunk.

Az állítás fogalom szinonimájaként, azaz a szabály és tényállítás fogalmak gyűjtőneveként használatos a **klóz** elnevezés is. Ez a Prolog tételbizonyítási gyökereihez vezet vissza, ahol is a logikai formulák ilyen alakját ún. **Horn klóznak** *(Horn clause)*, vagy **definit klóznak** *(definite clause)* nevezik.

A `járatszakasz` predikátumra építve, a `járat2` általánosításaként, egy olyan rekurzív predikátumot definiálunk most, amely adott számú szakaszból álló útvonalat keres:

```
% útvonal(N, A, B, Táv): A és B között van (pontosan)
% N szakaszból álló útvonal, amelynek összhossza Táv.
útvonal(0, Kezdet, Kezdet, 0).
útvonal(N, Kezdet, Cél, Táv) :-
    N > 0,
    N1 is N-1,
    járatszakasz(Kezdet, Közben, TávA),
    útvonal(N1, Közben, Cél, TávB),
    Táv is TávA+TávB.
```

Az `útvonal` predikátum két állításból áll. Az első egy tényállítás és arra az esetre vonatkozik, amikor 0 szakaszból álló útvonalat keresünk, és azt állítja, hogy tetszőleges `Kezdet` esetén egy 0 összhosszú útvonallal `Kezdet`-be juthatunk. Ez az útvonal-fogalom célszerű kiterjesztése, hasonló ahhoz, ahogy a számok 0. hatványát definiáljuk.

A második állítás egy szabály, eszerint ha `N > 0` szakaszból álló utat keresünk, a kezdőpontunkból van egy járatszakasz egy `Közben`-be, és innen egy `N1 = N-1` szakaszból álló útvonallal eljuthatunk a célunkhoz, akkor létezik egy `N` szakaszból álló útvonal kezdő- és célpontunk között, amelynek hossza a szakasz és a folytatás-útvonal hosszának összege. Vegyük észre azt is, hogy az `N-1` kivonás elvégzését az `is` beépített predikátum segítségével kellett elvégeznünk, mert a felhasználói predikátumok argumentumaiban nem történik meg az aritmetikai kifejezések kiértékelése — ennek magyarázatára később visszatérünk.

3.1.5. Diszjunkciók

Az előző szakaszban két olyan klózzal definiáltuk a `járatszakasz` predikátumot, amelyek feje megegyezett. A Prolog nyelv megengedi ennek a két klóznak az összevonását, az ún. diszjunkció műveletének alkalmazásával. A diszjunkció műveletét pontosvesszővel (`;`) jelöljük:

```
% járatszakasz(A, B, H): A és B között, vagy B és A
%    között van járat, amelynek úthossza H.
járatszakasz(Kezdet, Cél, Táv) :-
    (   járat(Kezdet, Cél, Táv)
    ;   járat(Cél, Kezdet, Táv)
    ).
```

Ez az alak minden tekintetben megegyezik az előző szakaszbeli két-klózos formával, a SICStus Prolog rendszer ténylegesen a több-klózos alakra való visszavezetéssel valósítja meg a diszjunkciókat. Ezért a végrehajtási mechanizmus ismertetésénél nem is kell foglalkoznunk a diszjunkciók kezelésével, úgy tekinthetjük, hogy egy megfelelő több-klózos segéd-predikátummal kiküszöböljük őket a végrehajtás megkezdés előtt.

3.2. Prolog programok végrehajtása

A Prolog és logikai programozás egyik különlegessége az, hogy kétféle értelmezés (szemantika) adható a nyelvhez. A leíró, **deklaratív** értelmezés az, amikor logikai formulaként értelmezzük a Prolog állításait. Eszerint egy kérdésre adott válasz nem más, mint egy olyan behelyettesítés-rendszer, amellyel a programból (mint logikai formulák halmazából) levezethetővé válik a kérdés.

A **procedurális** értelmezés ezzel szemben leírja, hogy milyen algoritmussal hajtódik végre a Prolog program. Természetesen ez utóbbi értelmezésre van szükség ahhoz, hogy pontosan követni tudjuk egy Prolog program futását. A deklaratív értelmezés viszont sokkal olvashatóbbá teszi a programokat, hiszen így a programot alkotó viszonylag kisméretű **független** állítások jelentését kell értelmeznünk. Ha egy Prolog program minden állítását igaznak találjuk a fejkommentárok által adott értelmezés mellett, akkor a program nem adhat hibás eredményt (bár az lehet, hogy végtelen ciklusba esik). Sokszor persze nehéz egy összetettebb Prolog predikátum jelentését (fejkommentárját) megfogalmazni, és vannak a Prolog nyelvnek nem-deklaratív elemei is, amelyek megnehezíthetik az állítások logikai értelmezését. Tehát mindenképpen szükséges a Prolog végrehajtási mechanizmus pontos megértése, de a deklaratív értelmezés egy másik, a program helyességét megerősítő dimenziót ad.

3.2.1. A végrehajtási mechanizmus alapelemei

Az `útvonal` predikátum példáját követve most áttekintjük a Prolog nyelv végrehajtási mechanizmusának alapelemeit. Ehhez többféle szemlélettel is közelíthetünk, tekinthetjük

- tételbizonyítási folyamatnak (SL rezolúciónak),
- célvezérelt keresési folyamatnak, vagy
- általánosított eljáráshívási folyamatnak.

A háromféle szemléletmódhoz különböző szóhasználat is tartozik, ezeket az 3.1 táblázatban foglaljuk össze.

Tételbizonyítás	* klóz, állítás,	* predikátum	pozitív literál	negatív literál		bizonyítandó állítás, * kérdés	rezolúciós lépés
Célvezérelt keresés	* szabály + tényállítás		szabályfej	* cél	részcélok	* célsorozat	* redukciós lépés
Eljárásszervezés	eljárás-változat	* eljárás	* eljárásfej	* eljáráshívás	* eljárástörzs	hívássorozat	eljáráshívási lépés

3.1. táblázat: A VÉGREHAJTÁSI MECHANIZMUS HÁROM SZEMLÉLETE

Bár mindhárom szemlélet jogos, a Prolog nyelv leginkább általános programozási nyelvnek tekinthető. Mi most főleg az eljárásos szemléletet használva ismertetjük a Prolog végrehajtási mechanizmusát, bár lesznek olyan elemek, amelyek a célvezérelt keresés fogalmaival magyarázhatók a legegyszerűbben. A Prolog nyelvre vonatkozó külföldi és magyar nyelvű irodalomban is sokszor vegyesen használják a megnevezéseket: a fenti táblázatban *-gal jelöltük a legelterjedtebb Prolog terminológiát. Bár ez kissé eklektikus, mi is ezt fogjuk használni.

Az eljárásos szemlélet szerint egy Prolog programban szereplő azonos predikátumnevű és argumentumszámú klózok egy eljárást alkotnak. Egy eljárást tehát egy **Név/Argumentumszám** páros határoz meg, ezt a két adatot együtt, egymástól osztásjellel elválasztva az eljárás **funktorának** *(functor)* nevezzük. A fejezet korábbi részében példaként tehát a `járat/3`, `járat2/3`, `járatszakasz/3`, és az `útvonal/4` eljárások szerepeltek. A hagyományos programozási nyelvektől eltérően a Prolog megengedi, hogy egy programban ugyanaz az eljárásnév különböző argumentumszámmal is előforduljon, és ezeket teljesen különböző eljárásoknak tekinti. Bár a programfejlesztés során ez sokszor hibaforrást jelent, szokásos módszer, hogy egy eljárás megírásához használt (tőle eltérő argumentumszámú) segédeljárást ugyanazzal a névvel illetik.

Így tehát egy eljárást egy vagy több állítással definiálunk, ezeket alternatív definícióknak vagy eljárás-változatoknak tekintjük. Például a `járat/3` eljárás négy tényállításból, a `járat2/3` egyetlen szabályból áll. Egy eljárás-változat (eljárás)fejből és (eljárás)törzsből épül fel, tényállítások esetén ez utóbbi üres, és ilyenkor a fejet és törzset elválasztó `:-` is elmarad. Az eljárásfej az eljárásnévvel kezdődik és zárójelbe tett formálisargumentum-listával folytatódik (argumentum nélküli eljárások esetén a zárójelek is elmaradnak). Az eljárástörzs nulla, egy, vagy több, egymástól vesszővel elválasztott eljáráshívásból áll. Egy eljáráshívás a meghívandó eljárás nevéből és a zárójelbe tett aktuálisargumentum-listából épül fel. Az `'is'` beépített eljárás példáján láthattuk, hogy egyes eljáráshívások ún. operátoros formában is írhatók, erre később visszatérünk.

3.2.2. Paraméter-átadás — egyesítés

A hagyományos nyelvek eljárásfogalma szerint az eljárás aktuális argumentumai változók vagy konstansok lehetnek, míg formális argumentumként csak (változó)nevek szerepelhetnek. Prologban más a helyzet, mindkét argumentumfajta lehet változó, vagy konstans (vagy összetett adat, amivel később ismerkedünk meg). A paraméterátadás egy kétirányú minta-illesztés segítségével történik, ezt gyakran egyesítésnek *(unification)* nevezik, a tételbizonyítási szemléletből örökölt kifejezéssel. Az egyesítés során az eljáráshívást és az eljárásfejet azonos alakra kell hoznunk, a változók behelyettesítésével. Példaként nézzük az `útvonal/4` eljárás egy hívását: `útvonal(0, 'Budapest', Cél, Táv)`. Ezt az `útvonal/4` eljárás első klózával az `útvonal(0, Kezdet, Kezdet, 0)` tényállítással próbáljuk először egyesíteni. Az egyesítés sikeres, a szükséges változó-behelyettesítések:

```
Kezdet = 'Budapest', Cél = 'Budapest', Táv = 0
```

Ezeknek a behelyettesítéseknek a segítségével mind a hívás, mind a fej az

```
útvonal(0, 'Budapest', 'Budapest', 0)
```

alakot ölti.

Vegyük észre, hogy a Prolog egyesítési mechanizmusa egyaránt alkalmas ki- és bemenő paraméterátadásra: a fenti példában a második paraméter bemenő, míg a harmadik és negyedik paraméter kimenő jellegű. Kezdő Prolog programozók számára ez gyakran nehezen érthető, ezért például az `útvonal` első klóza helyett egy bonyolultabb szabályt írnak fel:

```
útvonal(N, Honnan, Hová, Táv) :-
    N = 0, Hová = Honnan, Táv = 0.
```

Itt az `A=B` beépített eljárást használják, amely két argumentumát egyesíti egymással: ezzel a formával akarják jelezni, hogy az eljárás-változat alkalmazásához először ellenőrizni kell, hogy az első argumentum 0-e, majd a harmadik argumentumnak értékül a másodikat, a negyediknek pedig 0-t kell adni. Ez logikailag helyes, az eredeti tényállítással azonos jelentésű alak, de végrehajtása annál lassúbb. Általánosan is igaz, hogy ameddig csak logikailag tiszta, deklaratív nyelvi elemeket használunk, addig az `=` művelet felesleges: az egyik oldalán álló változó helyett a másik oldali kifejezést írhatjuk a változó minden más (adott klózbeli) előfordulása helyett. Ha ezt az elvet követjük, akkor a fenti szabály törzséből mindhárom hívást ki tudjuk küszöbölni és a korábbi tényállítás alakot kapjuk.

A ki- és bemenő argumentumok felcserélhetőségére példaként álljon itt egy másik hívás, ahol az 1. és 3. argumentum bemenő, míg a 2. és a 4. kimenő: `útvonal(0, Kezdet, 'Budapest', Táv)`. Ez is sikeresen egyesíthető az első állítással, a `Kezdet = 'Budapest', Táv = 0` behelyettesítésekkel.

Nézzünk most egy újabb hívást: `útvonal(2, 'Budapest', Hová, Táv)`! Ezt a hívást is először az eljárás első klózával próbáljuk illeszteni. Ez nem sikerülhet, mert ebben a tényállításban az első argumentum `0`, a hívásban pedig `2`. Ekkor a második klóz fejének (`útvonal(N, Kezdet, Cél, Táv)`) sikeres illesztése következik, a behelyettesítés:

```
N = 2, Kezdet = 'Budapest', Cél = Hová
```

Ebben a példában új, hogy két változó is egy-egy másik változóval illesztődik. Elvben a két egymással szemben álló változó bármelyikét helyettesíthetjük a másikkal, gyakorlatilag célszerű a fejváltozót helyettesíteni a hívásban szereplővel.

A változó–változó helyettesítés kapcsán teszünk egy másik észrevételt is: az egyesítés mindig a lehető legáltalánosabb behelyettesítést állítja elő, ez az ún. **legáltalánosabb egyesítő** *(most general unifier)*. Utolsó példánkban a fej és hívás azonos alakra hozását elérhettük volna úgy is, hogy a `Cél = Hová` helyett a `Cél = 'Párizs', Hová = 'Párizs'` helyettesítést alkalmazzuk (`Párizs` helyett persze szerepelhetne bármely más konstans). Ez utóbbi helyettesítés azonban speciálisabb mint a fenti, hiszen abból egy további behelyettesítéssel előáll, tehát nem a legáltalánosabb. Bizonyítható, hogy mindig létezik legáltalánosabb egyesítő, és a változóátnevezésektől eltekintve egyértelmű. Ezt a behelyettesítést előállító ún. egyesítési algoritmust később, a Prolog összetett adatstruktúráinak ismertetése után írjuk le.

3.2.3. A redukciós lépés

A Prolog végrehajtási mechanizmusának leírásához először definiáljuk a **redukciós lépés** (eljáráshívási lépés) fogalmát, amely a tételbizonyítási szemlélet rezolúciós lépésének felel meg. Tekintsük hívások egy sorozatát (egy célsorozatot) és egy klózt, mégpedig a sorozat első hívása definíciójának egyik klózát. A redukciós lépésben először elkészítjük a klóz egy másolatát, úgy, hogy a benne szereplő összes változónevet szisztematikusan a célsorozatban nem szereplő változónevekre cseréljük. A változócsere a klózok ismételt felhasználásához fontos, hiszen egy újabb eljáráshíváskor (pl. rekurzió estén) a célsorozatba bekerülő változókat a korábbiaktól különbözőnek kell tekinteni.

Ezután megvizsgáljuk, hogy az így lemásolt klóz feje és az első hívás egyesíthető, azaz változó-behelyettesítéssel azonos alakra hozható-e. Ha igen, akkor a redukciós lépés sikeres, és eredménye egy új célsorozat, amelyet úgy kapunk, hogy az eredeti sorozatra és a klóz törzsére egyaránt alkalmazzuk a legáltalánosabb egyesítő behelyettesítést, majd az első hívást az illesztett klóz törzsére cseréljük le (tényállítás esetén az első hívást elhagyjuk). Ha a klóz feje és a hívás nem egyesíthető, akkor a redukciós lépés sikertelen.

Például tekintsük az `útvonal(2, 'Budapest', Hová, Táv), Táv > 1000` célsorozatot. Ha erre, és az `útvonal/4` első klózára kísérelünk meg egy redukciós lépést, akkor az sikertelen lesz, hiszen a klóz feje és az első hívás nem egyesíthető. Ugyanerre a célsorozatra és a második klózra a redukciós lépés sikeres. Ennek eredménye, a korábban leírt legáltalánosabb egyesítő behelyettesítésnek megfelelően, az alábbi célsorozat:

```
2 > 0, N1 is 2-1, járatszakasz('Budapest', Közben, TávA),
útvonal(N1, Közben, Hová, TávB), Táv is TávA+TávB,
Táv > 1000.
```

3.2.4. A Prolog végrehajtási algoritmusa

A Prolog visszalépéses végrehajtási algoritmusa az általános visszalépéses keresési algoritmus Prologra szabott változata. Az algoritmus leírásához két változót használunk: `CS` jelöli a pillanatnyi célsorozatot, míg `I` a kiválasztott klóz eljáráson belüli sorszámát. Ezeknek az alapváltozóknak az értékeit el kell mentenünk, hogy visszalépés esetén egy korábbi állapotba vissza tudjunk térni. Erre a célra egy olyan vermet használunk amelybe <CS, I> alakú párokat helyezünk.

A leírt algoritmus bemenő adatai: a program (eljárásdefiníciók halmaza) és a célsorozat; eredménye pedig egy, a futás sikeres vagy sikertelen voltát jelentő logikai érték.

1. *(Kezdeti beállítások:)* A verem kezdetben legyen üres, `CS := kezdeti célsorozat`
2. *(Beépített eljárások:)* Tekintsük a `CS` célsorozat első hívását. Ha ez beépített eljárásra vonatkozik, akkor hajtsuk végre az eljárást:
   a. Ha a végrehajtás sikertelen, akkor menjünk a 6. lépésre.
   b. Ha a végrehajtás sikeres, akkor végezzük el a beépített eljárás által esetleg kiváltott behelyettesítéseket a `CS` sorozaton, hagyjuk el az első hívást, és az így kapott célsorozatot tekintsük a továbbiakban `CS`-nek. Ezután folytassuk a 5. lépéssel.
3. *(Klózszámláló kezdőértékezése:)* Legyen `I = 1`.
4. *(Redukciós lépés:)* Itt `CS` első hívása nem beépített. Tekintsük a híváshoz tartozó eljárásdefiníciót, legyen az ebben levő klózok száma `N`.
   a. Ha `I > N`, akkor menjünk a 6. lépésre.
   b. Tekintsük a definíció `I`-edik klózát és kíséreljünk meg egy redukciós lépést erre a klózra és a `CS` célsorozatra.
   c. Ha a redukciós lépés sikertelen, akkor `I := I+1`, és (változatlan `CS` mellett) ismételjük a 4. lépést.
   d. Itt a redukciós lépés sikeres. Ha `I < N` (nem utolsó klóz), akkor mentsük el a verem tetejére a <CS, I> párt.

   e. A redukciós lépés eredményeként kapott célsorozatot tekintjük `CS`-nek.

5. *(Siker:)* Ha `CS` üres sorozat, akkor a végrehajtási algoritmus sikeresen véget ért, egyébként folytassuk a 2. lépésnél.
6. *(Sikertelenség:)* Ha a verem üres, akkor a végrehajtás sikertelenül véget ért.
7. *(Visszalépés:)* Ha a verem nem üres, akkor leemeljük a verem tetején levő `<CS, I>` párt, ebből visszaállítjuk a `CS` és `I` változók értékét. Ezután `I := I+1`, és folytatjuk a 4. lépésnél.

A fenti algoritmus a célsorozat lefutásakor csak a siker tényét jelzi, de a benne levő változók behelyettesítéseit nem adja meg. A következő „trükkel” megkaphatjuk a behelyettesítéseket is.

Az algoritmus elindítását megelőzően a kezdeti célsorozatban szereplő minden olyan `V` változó esetén, amelynek behelyettesítését meg kívánjuk kapni, a célsorozat végére fűzzünk egy `write(V)` hívást. A `write` egy beépített eljárás, amely a paramétereként megadott Prolog kifejezést kiírja. Az alábbi példában a `write` hívások lefutását nem mutatjuk, amikor már csak ezek maradtak meg a célsorozatban, akkor a futást befejezettnek tekintjük. Így végül is a `write` hívások argumentumaiból ki tudjuk olvasni a keresett behelyettesítéseket.

Még egy kiegészítést tehetünk: ha egy sikeres lefutás után további megoldásokat kívánunk kerestetni, akkor visszalépéssel, a 7. pontnál kell folytatnunk a végrehajtást.

3.2.5. Egy végrehajtási példa

A 3.2 táblázatban bemutatjuk az alábbi célsorozat végrehajtásának néhány lépését.

```
útvonal(2, 'Budapest', Hová, Táv), Táv > 1000.
```

Az első oszlop a `CS` célsorozatot, a második az `I` klóz-számláló értékét mutatja, míg a harmadik a verem (ez üresen marad, ha nincs változás az előző sorhoz képest). Az utolsó oszlopban szereplő számok a végrehajtási algoritmus lépéseire utalnak. A célsorozat alatt megadjuk az adott lépés által kiváltott változó-behelyettesítéseket.

A táblázat első sorában a kezdeti célsorozat szerepel, a két `write` hívással kiegészítve (cs1). Ennek első hívását a 4.b. lépés szerint megkíséreljük egyesíteni az `útvonal/4` első klózával. Mivel ez sikertelen, ismételjük a 4. lépést: a második klózzal való redukció sikeres. Mivel az utolsó klózzal illesztettünk, ezért a 4.d. lépés szerint nem kell az állapotot a veremre menteni. Ez azért van így, mert a (cs1) célsorozathoz való visszalépés biztosan nem ad újabb megoldási lehetőséget, hiszen már az utolsó klózt, a keresési fa legjobboldalibb ágát választottuk.

A sikeres redukciós lépés eredménye (cs2) célsorozat. Ezt a táblázatban nem írtuk ki teljesen, a változatlan folytatást csak `...`-tal jelezzük. A (cs3) célsorozathoz a 2.b. lépés (beépített eljárás hívása) kétszeri végrehajtásával jutunk, amelyek közül a második váltja ki a jelzett `N1 = 1` behelyettesítést. Ezt a célsorozatot `járatszakasz` első klózával tudjuk redukálni. Mivel nem utolsó klózzal illesztettünk, a $<$(cs3),1$>$ párt a veremre mentjük, hogy visszalépés esetén ugyanennek a célsorozatnak a második klózzal való redukcióját is meg tudjuk kísérelni.

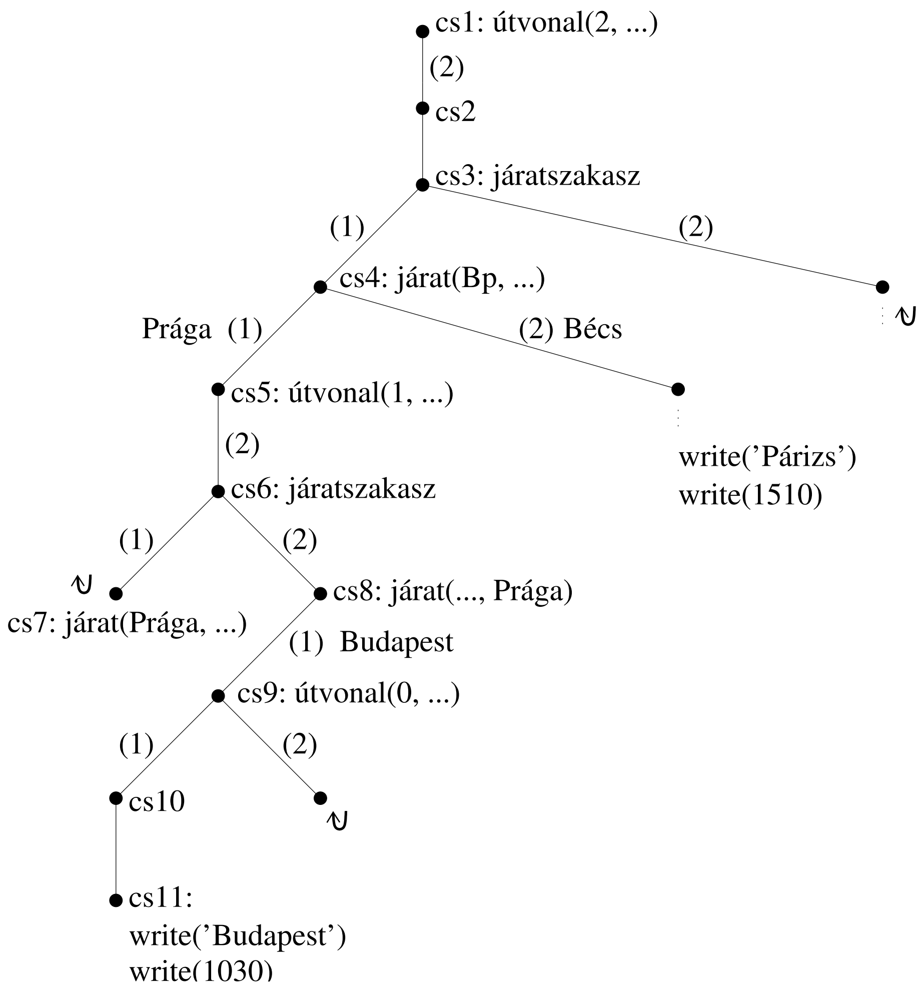
Innen két redukciós lépéssel jutunk el a (cs5) célsorozathoz, az `útvonal` eljárás rekurzív hívásához, és itt ismét a második klózzal illesztünk. Mivel ezt a klózt már alkalmaztuk, a redukciós lépés során szisztematikusan megváltoztatjuk a változóneveket, egy `_1` hozzátoldásával. További lépések segítségével jutunk el a (cs7) célsorozathoz, amelynek első hívása a `járat` eljárás egyik klózával sem illeszthető, ezért a vermet felhasználva visszalépést kell végrehajtanunk.

Vessük össze a verem tartalmát a 3.2 ábrán látható keresési fával (VAGY-fa)! Észrevehetjük, hogy a verem tartalma a gyökértől az adott levélig vezető úton azon csomópontoknak felel meg, amelyek további választási lehetőséggel, azaz még be nem járt ággal rendelkeznek. A veremben levő párok második komponense minden ilyen csomópontra megmondja, hogy a szóban forgó eljárás hányadik klózával történt meg a sikeres egyesítés. Így a verem azt teszi lehetővé, hogy a keresési fában visszalépve a megfelelő helytől folytassuk a keresést.

Leemeljük tehát a verem tetejéről a `<(cs6),1>` párt és a `járatszakasz` 2. klózával folytatjuk az végrehajtást. Mivel utolsó klózzal sikerül illesztenünk, nem rakjuk vissza a veremre a (cs6) célsorozatot. Ezután már

	Célsorozat (CS) *behelyettesítések*	Klóz (I)	Verem	Lépés
(cs1)	útvonal(2, 'Budapest', Hová, Táv), Táv > 1000, write(Hová), write(Táv).	2	üres	4.
(cs2)	2 > 0, N1 is 2-1, járatszakasz('Budapest', Közben, TávA), útvonal(N1, Közben, Hová, TávB), Táv is TávA+TávB, Táv > 1000, ... *N1 = 1*	–		2., 2.
(cs3)	járatszakasz('Budapest', Közben, TávA), útvonal(1, Közben, Hová, TávB), ...	1	<cs3,1>	4.
(cs4)	járat('Budapest', Közben, TávA), útvonal(1, Közben, Hová, TávB), ... *Közben = 'Prága', TávA = 515*	1	<cs4,1> <cs3,1>	4.
(cs5)	útvonal(1, 'Prága', Hová, TávB), Táv is 515+TávB, Táv > 1000, ...	2		4.
(cs6)	járatszakasz('Prága', Közben_1, TávA_1), útvonal(0, Közben_1, Hová, TávB_1), TávB is TávA_1+TávB_1, Táv is 515+TávB, Táv > 1000, ... *N1_1 = 0*	1	<cs6,1> <cs4,1> <cs3,1>	2., 2., 4.
(cs7)	járat('Prága', Közben_1, TávA_1), ...			4., 6., 7.
(cs6)	járatszakasz('Prága', Közben_1, TávA_1), ...	2	<cs4,1> <cs3,1>	4.
(cs8)	járat(Közben_1, 'Prága', TávA_1), útvonal(0, Közben_1, Hová, TávB_1), ... *Közben_1 = 'Budapest', TávA_1 = 515*	1	<cs8,1> <cs4,1> <cs3,1>	4.
(cs9)	útvonal(0, 'Budapest', Hová, TávB_1), TávB is 515+TávB_1, Táv is 515+TávB, Táv > 1000, write(Hová), write(Táv). *Hová ='Budapest', TávB_1 = 0*	1	<cs9,1> <cs8,1> <cs4,1> <cs3,1>	4.
(cs10)	TávB is 515+0, Táv is 515+TávB, Táv > 1000, write('Budapest'), write(Táv). *TávB = 515, Táv = 1030*	–		2., 2., 2.
(cs11)	write('Budapest'), write(1030).			

3.2. táblázat: A PÉLDAPROGRAM VÉGREHAJTÁSA



3.2. ábra: A PÉLDAPROGRAM KERESÉSI FÁJA

visszalépés nélkül, redukciós lépésekkel és beépített eljárások meghívásával eljutunk a (cs11) célsorozathoz, amely a kérdésünkre való választ tartalmazza:

```
(cs11) write('Budapest'), write(1030).
```

A válasz esetleg furcsának tűnhet: Budapestről két szakaszból álló utat kerestünk, és 1030 km megtétele után visszajutottunk Budapestre! Ez természetesen annak „köszönhető”, hogy a korábbi `járat2` példával ellentétben most az oda-vissza utat megengedő `járatszakasz` predikátumot használtuk elemi lépésként. A következő fejezetben megmutatjuk majd, hogyan küszöbölhetők ki az ilyen kört tartalmazó utak.

Nézzük meg mi történik most, ha újabb választ kérve a 7. lépessel folytatjuk a végrehajtási algoritmust! Most a verem tetején levő (cs9) célsorozat, amelynek első hívása `útvonal(0, ...)` alakú, az `útvonal/4` eljárás 2. klózával redukálódik. A törzs behelyettesítése után kapott célsorozat elején a `0 > 0` aritmetikai beépített eljárás hívása áll, ez meghiúsul, ezért folytatjuk a visszalépést. Vegyük észre, hogy milyen fontos az `N > 0` feltétel az `útvonal/4` predikátum második klózában. Ha ezt elhagyjuk, akkor a Prolog végtelen ciklusba esik a Budapest-Prága-Budapest-... szakaszokon ingázva.

A (cs9) célsorozat sikertelenségét követően visszatérünk a (cs8) hívás-sorozathoz, ahol is a 2. `járat` tényállítástól folytatva a keresést további Prágába menő járatot keresünk. Mivel ilyet nem találunk, tovább folytatjuk a visszalépést a `<(cs4),1>` állapothoz, ahol a 2. `járat` tényállítástól kezdve keresünk újabb Budapestről induló járatot. A `járat` 2. klózának, azaz a bécsi járatnak a választása végül is még egy megoldáshoz vezet:

```
| ?- útvonal(2, 'Budapest', Hová, Táv), Táv > 1000.
Táv = 1030, Hová = 'Budapest' ? ;
Táv = 1510, Hová = 'Párizs' ? ;
no
```

3.2.6. Egy aritmetikai példa

Most egy egyszerű számtani feladványt megoldó Prolog programot mutatunk be. A feladvány a következő: keressük azokat a kétjegyű számokat amelyek négyzete háromjegyű és a szám fordítottjával kezdődik.

```
jószám(Szám):-
      első_jegy(A),
      második_jegy(B),
      Szám is A * 10 + B,
      Szám * Szám // 10 =:= B * 10 + A.

első_jegy(1).  első_jegy(2).  első_jegy(3).  első_jegy(4).
első_jegy(5).  első_jegy(6).  első_jegy(7).  első_jegy(8).  első_jegy(9).

második_jegy(0).  második_jegy(1).  második_jegy(2).  második_jegy(3).
második_jegy(4).  második_jegy(5).  második_jegy(6).  második_jegy(7).
második_jegy(8).  második_jegy(9).
```

A feladványmegoldó program futása:

```
| ?- jószám(Szám).
 Szám = 27 ? ;
no
```

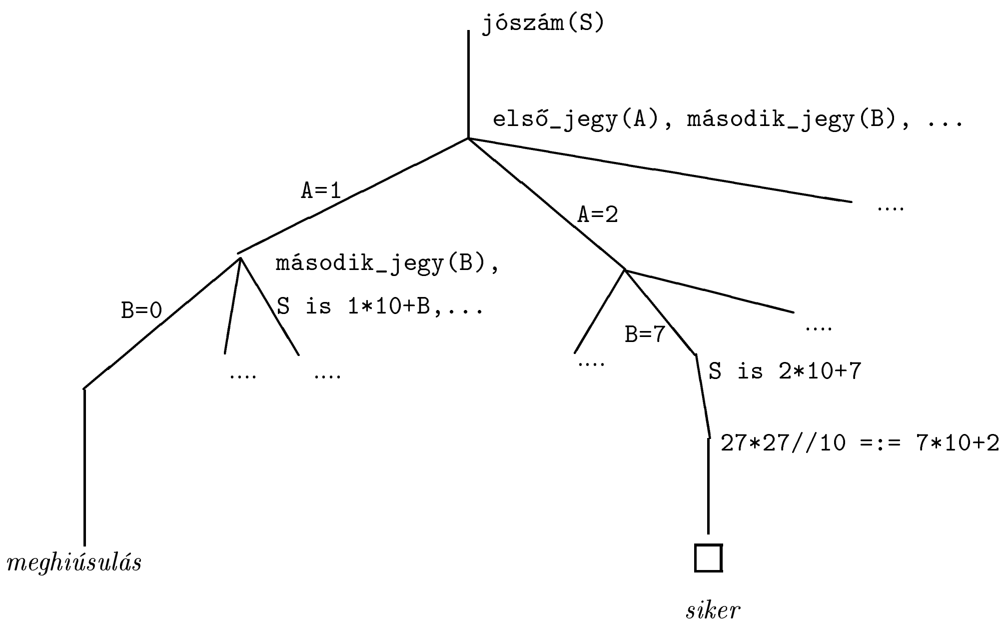
A program végrehajtása során az első redukciós lépés a `jószám` hívást az őt definiáló szabály törzsével helyettesíti. Ezt követően az `első_jegy` hívás redukálódik, először az első tényállítás segítségével, azaz `A=1`, majd a `második_jegy`, ennek első megoldása `B=0`. Ezekkel a behelyettesítésekkel hajtódik végre a

Szám értékét kiszámoló beépített eljárás hívása: `Szám is A * 10 + B`, amely az első esetben a `Szám=10` behelyettesítést eredményezi.

A célsorozat utolsó hívása az `=:=` beépített eljárás, ez mindkét argumentumát aritmetikai kifejezésként értelmezi, kiértékeli és sikerül, ha ezek azonos számértéket adnak (az `//` operátor az egész-osztást jelöli). Példánkban ez az utolsó hívás ellenőrzi, hogy `Szám` négyzetének utolsó jegyét lehagyva tényleg a `BA` kétjegyű számot kapjuk-e. A fenti első választásra ez nem sikerül, tehát visszatérünk az utolsó választási ponthoz, a `második_jegy` híváshoz, és annak újabb megoldásával (`B=1`) folytatjuk. Miután erre sem sikerül az ellenőrzés, újabb visszalépés történik és rendre a `B = 2, ..., 9` értékekkel próbálkozunk. Amikor az utolsó klózzal illesztettük a `második_jegy` hívást (`B=9`), akkor ezzel ezt a választási pontot meg is szüntettük. Mivel erre a behelyettesítésre sem teljesül a feltétel, visszalépünk az `első_jegy` híváshoz, és annak újabb választásával folytatjuk: `A=2`. Ehhez sorra vesszük a `második_jegy` által megengedett értékeket, és végül a `B=7` esetben megkapjuk az első megoldást.

Amikor a `;` leütésével újabb megoldás meglétét kérdezzük, a keresés folyatódik: a Prolog rendszer először még megvizsgálja az `A=2` választás esetén még fennmaradó `B=8` és `B=9` eseteket, majd az `A=3,..., 9` választások mindegyikéhez mind a tíz lehetséges `B` értéket. Mivel ezek egyike sem teljesíti a feltételt, a a `no` választ kapjuk.

A `jószám(Szám)` hívás keresési fáját az alábbi ábra mutatja.



Végül foglalkozzunk azzal, hogyan lehet a számok felsorolására használt `első_jegy` ill. `második_jegy` eljárásokat általánosítani. Tekintsük például az `első_jegy(X)` eljárást, ennek jelentése: `X` egy 1 és 9 közé eső egész szám. Felmerülhet a kérdés, miért nem jó a következő definició:

```
első_jegy(X) :-
      X >= 1, X =< 9.
```

Ez az eljárás alkalmas annak ellenőrzésére, hogy egy adott `X` érték 1 és 9 közé esik-e, arra azonban nem, hogy az összes ilyen egész értéket felsorolja. Ennek fő oka az, hogy a beépített aritmetikai összehasonlító eljárások elvárják, hogy az összehasonlítandó kifejezések a végrehajtás pillanatában konkrét értékkel rendelkezzenek (és természetesen azt a megszorítást sem tartalmazzák, hogy `X` egész legyen).

Adott intervallumba eső egész számok felsorolását rekurzív módon lehet megfogalmazni:

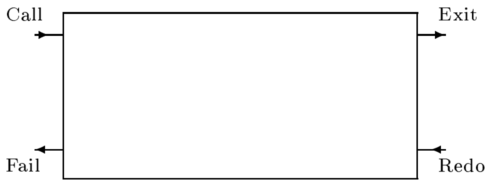
```
% between(N, M, I): I olyan egész, amely az N és M egész
% számok közé esik (N =< I =< M).
between(N, M, N):-
      N =< M.
between(N, M, I):-
      N < M, N1 is N+1,
      between(N1, M, I).
```

A `between` eljárás első két argumentuma csak bemenő lehet, az utolsó lehet bemenő és kimenő is. Ha az utolsó argumentum bemenő, a `between` eljárás nem hatékony, hiszen pl. a `between(1, 1000, 1000)` vizsgálatot 1000 rekurzív lépésben hajtja végre.

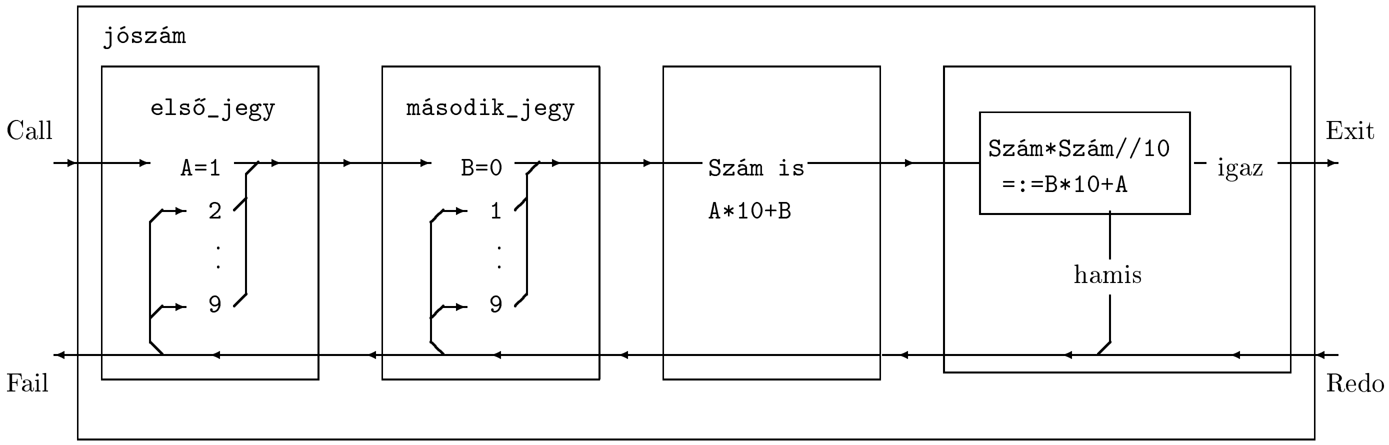
3.3. A Prolog végrehajtás eljárás-doboz modellje — nyomkövetés

A keresési fa mellett van a Prolog végrehajtásnak egy másik megjelenítési formája, az ún. eljárás-doboz modell (procedure box model). Ezt néha Byrd-doboz modellnek is hívják (Lawrence Byrd volt az egyik első Prolog nyomkövető rendszer létrehozója).

Az eljárás-doboz modellben egy eljáráshívást egy olyan dobozzal ábrázolunk amelynek négy ún. kapuja van: hívás (`Call`), kilépés (`Exit`), új-megoldás (`Redo`) és sikertelenség (`Fail`):



Amikor meghívunk egy eljárást, akkor a `Call` kapun megyünk be. Egy futó eljárásból kétféleképp léphetünk ki: sikeresen az `Exit` kapun, illetve sikertelenül a `Fail` kapun. Végül egy már sikeresen lefutott eljárásba visszaléphetünk egy új megoldás keresése végett a `Redo` kapun; ezután ismét az `Exit` vagy a `Fail` kapu következik, stb. A kapuk fenti elrendezése a dobozok olyan összekapcsolását teszi lehetővé, amely a Prolog végrehajtási sorrendnek felel meg. Ennek bemutatására tekintsük az előző alfejezet `jószám` eljárását szemléltető dobozt:



A jószám dobozán belül az általa meghívott eljárások dobozai találhatók. A külső (jószám) eljárás Call kapuja a törzsében levő első eljárás Call kapujához kapcsolódik. Az első hívás Exit kapujából a nyíl a második hívás Call kapujába vezet stb.; végül az eljárástörzs utolsó hívásának Exit kapuja után a külső eljárás Exit kapuja következik. Ezek a balról-jobbra menő nyilak az előrehaladó, sikeres végrehajtáshoz tartoznak.

Az ábra alsó részében található jobbról-balra menő nyilak a meghiúsulásnak és visszalépésnek felelnek meg. Egy törzsbeli eljárás Fail kapujából mindig a közvetlenül előtte levő hívás Redo kapujához vezet a nyíl (kivéve a legelső hívást, ahol a külső Fail kapuhoz). Ez felel meg a Prolog azon végrehajtási alapelvének, hogy meghiúsulás esetén a közvetlenül megelőző választási ponthoz megyünk vissza.

A legtöbb Prolog megvalósítás nyomkövető rendszere az eljárás-doboz modellt használja. Nézzünk most egy rövid párbeszédet a SICStus nyomkövető rendszerével:

```
| ?- trace, jószám(Szám).
{The debugger will first creep -- showing everything (trace)}
        1      1 Call: jószám(_199) ? s
?       1      1 Exit: jószám(27) ?

Szám = 27 ? ;
        1      1 Redo: jószám(27) ? s
        1      1 Fail: jószám(_199) ?

no
{trace}
| ?-
```

A trace beépített eljárással az ún. teljes nyomkövetési módot kapcsoljuk be. Így a rendszer az első hívásnál azonnal megáll. A megjelenített nyomkövetési sorban az első helyen egy hívásszámláló áll, ezzel a sorszámmal lehet az adott hívást azonosítani egyes összetettebb nyomkövetési parancsok esetén. A második szám a nyomkövetés mélységét mutatja, ezt követi a kapu megnevezése, majd maga a hívás (ebben a változókat a rendszer egy a belső alakjuknak megfelelő számkóddal jeleníti meg). Az ezt követő ? jelzi, hogy a rendszer válaszra vár. A fenti példában az első válasz az s parancs (skip), amely azt kéri, hogy a nyomkövető lépje át az adott eljáráshívást és csak akkor jelentkezzék újra, amikor az adott hívás egy újabb kapuhoz érkezik.

Az adott hívás sikeresen lefut, ezért a következő nyomkövetési sort az Exit kapunál kapjuk. Az eljáráshívást a pillanatnyi behelyettesítettségnek megfelelően írja ki a rendszer, ezért jelenik meg a feladvány megoldása (a 27) az argumentumban. A sor elején levő kérdőjel azt jelzi, hogy az eljárás sikeres lefutása során maradtak benne választási pontok, amelyek esetleg újabb sikeres eredményekhez vezethetnek.

A második nyomkövetési sorban a nyomkövetőnek egy újsor karakterrel válaszoltunk, ez a c paranccsal (creep) egyenértékű, amely teljes nyomkövetési módban futtat, azaz minden kapunál megáll. Esetünkben nincs újabb kapu, megkapjuk a feladvány megoldását, majd az arra adott ; választ követően megérkezünk a jószám hívás Redo kapujához. Itt egy újabb skip parancs hatására átlépünk a Fail kapuhoz, hiszen nincs több megoldás; és ezután a teljes futás is sikertelenül véget ér.

Most nézzünk egy kicsit részletesebb futást, ahol teljes nyomkövetési módban (azaz újsor paranccsal) lépkedünk a nyomkövetésben!

```
| ?- jószám(Szám).
        1      1 Call: jószám(_123) ?
        2      2 Call: első_jegy(_904) ?
?       2      2 Exit: első_jegy(1) ?
        3      2 Call: második_jegy(_899) ?
?       3      2 Exit: második_jegy(0) ?
        4      2 Call: _123 is 1*10+0 ?
```

```
        4      2 Exit: 10 is 1*10+0 ?
        5      2 Call: 10*10//10=:=0*10+1 ?
        5      2 Fail: 10*10//10=:=0*10+1 ?
        3      2 Redo: második_jegy(0) ?
?       3      2 Exit: második_jegy(1) ? +
{Plain spypoint for user:második_jegy/1 added, BID=1}
?+      3      2 Exit: második_jegy(1) ? l
 +      3      2 Redo: második_jegy(1) ?
```

A fenti nyomkövetési párbeszédben látjuk, hogy a 2-es mélységben sorra meghívódnak a `jószám` törzsében levő eljárások (2, 3, 4 és 5 sorszámmal) mígnem az utolsó meg nem hiúsul. A doboz modell szerint most a megelőző hívás (az `is` beépítetteljárás-hívás) `Redo` kapujára kellene mennünk. Mivel azonban ez determinisztikusan, azaz fennmaradó választási lehetőség nélkül futott le (nincs `?` a 4. `Exit` kapunál), a nyomkövető ezt a hívást mintegy rövidre zárja, és a legutolsó nem-determinisztikus hívás `Redo` kapujához lép vissza (azaz a 3. sorszámú `második_jegy` híváshoz).

A `második_jegy` második sikeres lefutásánál egy `+` nyomkövető parancsot adtunk ki. Ez egy ún. kémlelő-pontot helyez el az aktuális eljárásra, lehetővé téve, hogy nagyobb lépésekben haladjunk a nyomkövetésben. Az `l` parancs (leap) ugyanis mindaddig „csendben", azaz nyomkövetési kiírás nélkül futtatja a programot, amig egy kémlelő-pont-eljárás valamelyik kapujához nem ér. A nyomkövetési sor második pozícióján szereplő `+` jel mutatja, hogy az adott sor egy kémlelő-pont.

Alább mutatjuk a nyomkövetés folytatását (az ismétlődő sorok helyett néhány helyen a `(...)` jelzést alkalmaztuk). Az utolsó nyomkövetési sorban az `n` (notrace) parancsot adtuk ki, ez teljesen kikapcsolja a nyomkövetést, az adott futás végéig. A `notrace` beépített eljárással lehet tartósan kikapcsolni a nyomkövetést.

```
 +      3      2 Redo: második_jegy(1) ? l
?+      3      2 Exit: második_jegy(2) ? l
(...)
?+      3      2 Exit: második_jegy(8) ? l
 +      3      2 Redo: második_jegy(8) ? l
 +      3      2 Exit: második_jegy(9) ?
       22      2 Call: _123 is 1*10+9 ?
       22      2 Exit: 19 is 1*10+9 ?
       23      2 Call: 19*19//10=:=9*10+1 ?
       23      2 Fail: 19*19//10=:=9*10+1 ?
        2      2 Redo: első_jegy(1) ?
?       2      2 Exit: első_jegy(2) ?
 +     24      2 Call: második_jegy(_899) ?
?+     24      2 Exit: második_jegy(0) ? l
 +     24      2 Redo: második_jegy(0) ? l
(...)
?+     24      2 Exit: második_jegy(6) ? l
 +     24      2 Redo: második_jegy(6) ? l
?+     24      2 Exit: második_jegy(7) ?
       39      2 Call: _123 is 2*10+7 ?
       39      2 Exit: 27 is 2*10+7 ?
       40      2 Call: 27*27//10=:=7*10+2 ?
       40      2 Exit: 27*27//10=:=7*10+2 ?
?       1      1 Exit: jószám(27) ?

Szám = 27 ? ;
        1      1 Redo: jószám(27) ?
 +     24      2 Redo: második_jegy(7) ?
?+     24      2 Exit: második_jegy(8) ? l
```

```
 +     24      2 Redo: második_jegy(8) ? l
 +     24      2 Exit: második_jegy(9) ? l
?+     45      2 Exit: második_jegy(0) ? n

no
{trace}
| ?-
```

3.4. Összetett adatstruktúrák

A Prolog adatfogalma általánosabb mint a LISP nyelvé: a C vagy Pascal nyelvek struktúra (struct) fogalmához hasonló rekord-szerkezetet ad összetett adatok képzésére. A Prolog lista-fogalma ennek a rekord-szerkezetnek speciális esete. A listák, széleskörű alkalmazhatóságukra való tekintettel, mégis kitüntetett szerepet játszanak, olyannyira, hogy külön jelölésrendszer könnyíti meg a használatukat. Ezért most először a Prolog lista-fogalmát ismertetjük.

3.4.1. Listák

Útvonalkereső példa-eljárásunkat szeretnénk úgy továbbfejleszteni, hogy kört tartalmazó útvonalat ne adjon eredményül. Példa-adatbázisunkban csak a járatok megfordíthatósága miatt keletkeznek körök, és azok is csak két szakasz hosszúságúak. Szeretnénk azonban egy olyan megoldást kidolgozni, amely általánosan, tetszőleges kört tartalmazó járat-rendszer esetén is működik. Ehhez nyilván kell tartanunk a megtett utat, azaz az érintett városok sorozatát, hogy ezekbe ne térjünk vissza. Ilyen célra a Prologban, a LISP-hez hasonlóan **lista**-struktúrákat használunk.

Üres lista jelölésére, konvencószerűen a `'[]'` névkonstans szolgál, ennek írásakor nem kell az aposztróf-jeleket kitenni: `[]`. Nem üres lista építésére Prologban a `[Fej|Farok]` jelölés szolgál (az felel meg a LISP `cons` függvényének), ahol `Fej` és `Farok` tetszőleges Prolog objektumok, azaz szám- ill. névkonstansok, változók, listák vagy más összetett adatok lehetnek. Ahhoz, hogy ún. valódi listát kapjunk, `Farok`-nak (üres vagy nem-üres) listának kell lennie, ez esetben ugyanis egy ilyen objektum felírható

```
[Elem1|[Elem2|[... [ElemN|[]] ...]]]
```

alakban, ahol `Elem1`, ..., `ElemN` tetszőleges objektumok. Ezt a listát írhatjuk egyszerűbben így is:

```
[Elem1,Elem2, ...,ElemN]
```

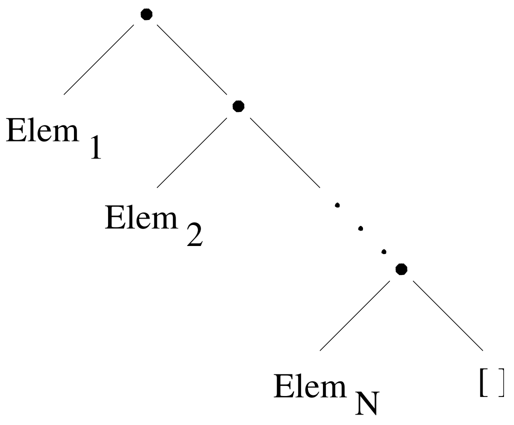
A Prolog rendszer belsejében mindenképpen a fenti sok-zárójeles alaknak megfelelő fastruktúra (3.3 ábra) tárolódik.

A lista-jelölés egyike a Prolog számos ún. „szintaktikus édesítőszereinek”, azaz olyan írásmódoknak, amelyek a programozó kényelmét szolgálják, de a program beolvasásakor átalakulnak valamilyen szabványos jelöléssé. Egy másik szintaktikus édesítőszer alkalmazható az `[Elem1|[Elem2|[... [ElemN|Farok] ...]]]` szerkezet írásmódjának egyszerűsítésére, ezt így írhatjuk:

```
[Elem1,Elem2, ...,ElemN|Farok].
```

Listák szétbontására a Prolog nyelv nem ad külön eszközt, mert a Prolog illesztési mechanizmusa lehetővé teszi, hogy a `[Fej|Farok]` jelölést ne csak listák építésére, hanem szétszedésére is használjuk. Például az alábbi tényállítással egy olyan eljárást definiálunk, amely az első argumentumaként átadott lista fejét adja ki a második argumentumban.

```
% feje(Lista, Fej): Lista feje Fej.
feje([Fej|Farok], Fej).

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



3.3. ábra: A LISTÁK FASTRUKTÚRA ALAKJA

A `feje` eljárás működésének megértéséhez vegyük figyelembe, hogy a fenti eljáráshívás szabványos alakja `feje([1|[2|[]]], X)`. A hívás elvégzéséhez most is az előző fejezetben leírt egyesítés fogalmát használjuk, azaz megkeressük a változók egy olyan behelyettesítését, amellyel a hívás és az eljárásfej azonos alakra hozható. Esetünkben ez a behelyettesítés: `Fej = 1, Farok = [2|[]], X = 1`, így áll elő a fenti eredmény. A `feje` eljárás arra is használható, hogy eldöntsük, hogy egy adott elem azonos-e a lista fejével:

```
| ?- feje([1,2], 1).
yes
| ?- feje([1,2,3], 2).
no
```

A `feje` eljáráshoz hasonlóan definiálható a `farka` eljárás is:

```
% farka(Lista, Farok): Lista farka Farok.
farka([Fej|Farok], Farok).
```

3.4.2. Listaelemek keresése

Írjunk most egy olyan `member` Prolog eljárást, amellyel eldönthetjük, hogy egy adott elem valahol előfordul-e egy adott listában (azért adtunk angol nevet az eljárásnak, mert többnyire ezen a néven szerepel a legtöbb Prolog rendszer listakezelő könyvtári eljárásai között, SICStus Prolog-ban a `lists` könyvtárban található):

```
% member(Elem, Lista):  Elem a Lista elemeként előfordul.
member(Elem, Lista) :-
    feje(Lista, Elem).
member(Elem, Lista) :-
    farka(Lista, Farok),
    member(Elem, Farok).
```

Először adjuk meg az eljárás deklaratív olvasatát. Az első klóz jelentése: egy elem előfordul egy listában, ha annak feje, a másodiké: egy elem előfordul egy listában, ha előfordul annak farkában. A két részt egybefoglalva: egy elem előfordul egy listában, ha vagy illeszthető a lista fejével, vagy előfordul a lista farkában. Ez a deklaratív olvasat megfelel elvárásainknak.

Tekintsük most a `member` eljárás egy meghívását, és kövessük a végrehajtást:

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

Nézzük, hogyan jut a Prolog rendszer erre az eredményre! Először az első klózzal redukálunk, a kapott célsorozat: `feje([1,2,3], 2)`. Ez meghiúsul, visszalépést okoz, ekkor a második klózzal próbálkozunk. Ez a redukciós lépés sikeres, eredménye a `farka([1,2,3], Farok), member(2, Farok)` célsorozat. Ennek első hívása sikeresen lefut, a `Farok=[2,3]` illesztéssel, marad a `member(2, [2,3])` hívás, majd az első klóz alkalmazásával ebből kapott `feje([2,3], 2)` hívás, amely szintén sikeresen fut le.

A `member` eljárás tehát, ha egy adott objektummal és egy listával hívjuk, úgy működik, hogy sorra megpróbálja illeszteni az adott objektumot a lista első, második stb. elemére, és sikeresen lefut, ha ez sikerül. A `member` hívás első argumentuma azonban lehet változó is, ekkor az eljárás először az első elemet adja értékül a változónak, majd ha további megoldásokat kérünk, sorban felsorolja a lista többi elemét:

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

Két egymás mellé helyezett `member` hívással megkereshetjük két lista közös elemeit:

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

Itt az első `member` hívás visszalépésesen *felsorolja* az első lista elemeit az `X` változóban, a második pedig *ellenőrzi*, hogy az adott elem szerepel-e a második listában.

A teljesség kedvéért nézzük meg, hogy működik a `member` eljárás, ha második argumentumában adunk meg változót:

```
| ?- member(1, L), member(2, L), member(1, L).
L = [1,2|_A] ?
```

Az első hívás a `member` első klózát használva sikeresen lefut és az `L = [1|Farok]` behelyettesítést eredményezi. A második hívás már ezt a behelyettesített `L` értéket kapja, ezért az első klózbeli `feje([1|Farok], 2)` meghiúsul. A második klóz segítségével viszont redukálódik a `member(2, Farok)` hívásra, amelyet a Prolog a `Farok = [2|Farok1]` behelyettesítéssel old meg. A harmadik hívás nem eredményez újabb behelyettesítést, mert az első `member` klózzal illesztve sikeresen lefut. Így alakul ki a kiírt behelyettesítés: `L = [1,2|Farok1]` (a Prolog rendszer a válaszban szereplő behelyettesítetlen változóknak az `_A`, `_B`, stb. nevet adja).

A `member` fenti használata tehát egy megadott elemet egy „nyílt végű lista” elemévé tesz. Nyílt végűnek nevezünk egy listát, ha valahány (esetleg 0) adott elem után a farok helyén egy értékkel nem bíró változó áll. Ez valójában egy olyan lista-minta, amely az összes adott kezdetű listát jelenti; például az imént eredményként kapott `[1,2|_A]` az összes olyan listát jelenti, amelynek első két eleme `1` és `2`.

Egy komoly gond van a `member` ilyen használatával. Nézzük mi történik ha kérjük a fenti célsorozat további megoldásait:

```
| ?- member(1, L), member(2, L), member(1, L).
L = [1,2|_A] ? ;
L = [1,2,1|_A] ? ;
L = [1,2,_A,1|_B] ? ;
L = [1,2,_A,_B,1|_C] ? ;
L = [1,2,_A,_B,_C,1|_D] ?
```

A `member` eljárásnak ez a használata egy végtelen választási lehetőséget hoz létre. Ennek az az oka, hogy egy adott elem, pl. az `1`, egy nyílt végű listában végtelen sok helyre helyezhető el. A fenti megoldás-sorozatban csak a harmadik `member` hívás ad újabb és újabb megoldásokat, az `1` számot az `L` listában először az első, majd a 3., 4., 5., 6. stb. helyre teszi. Így, miután az utolsó részcélnak végtelen sok megoldása van, a megelőző részcélok más megoldásaira soha sem kerülhet sor a visszalépés során. Sőt, ha egy ilyen `member` hívás a keresési fa egy olyan ágán van, amely zsákutcába vezet, akkor a Prolog rendszer végtelen ciklusba esik a visszalépés során.

A végtelen választási pontok tehát nagyon veszélyesek, ugyanúgy, mint a hagyományos programozási nyelvekben a végtelen ciklusok. A `member` nyílt végű listákra csak úgy használható a gyakorlatban, ha a végtelen választás lehetőségét korlátozzuk. Erre szolgál a később ismertetendő vágó beépített eljárás.

Mielőtt továbbmennénk, írjuk át a `member` eljárást úgy, hogy benne ne kelljen a `feje` és `farka` hívásokat használni. Ezt könnyen megtehetjük, mert pl. a `feje(Lista, Fej)` hívás helyettesíthető a `Lista=[Fej|Farok]` beépített eljáráshívással:

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

Ezután, mint azt korábban leírtuk, az egyenlőség is kiküszöbölhető, úgy hogy a `Lista` változó másik előfordulása helyébe írjuk az egyenlőség jobboldalán levő kifejezést:

```
member(Elem, [Elem|Farok]).
```

Így egy tényállításhoz jutottunk, amelynek deklaratív olvasata azonos az eredetivel: egy elem előfordul egy listában, ha annak fejeként szerepel. Procedurálisan viszont ez az alak hatékonyabb, hiszen kevesebb redukciós lépést kell elvégezni.

Még egy utolsó csiszolási lehetőség a fenti tényállításon: a benne előforduló `Farok` változó elnevezésére nincs szükség, hiszen csak egyszer fordul elő a klózban. Az ilyen változókat szokásos az egyetlen aláhúzásjelből álló, ún. névtelen változóval jelölni. Fontos megjegyezni, hogy ha egy klózon belül több `_` jel szerepel, akkor ezek páronként különböző változókat jelölnek.

Hasonló egyszerűsítést végezhetünk a `member` második klózán, és így jutunk az eljárás szokásos tömör definíciójához:

```
% member(Elem, Lista):  Elem a Lista elemeként előfordul.
member(Elem, [Elem|_]).
member(Elem, [_|Farok]) :-
    member(Elem, Farok).
```

A `member` eljárást diszjunkció formájában is felírhatjuk, így talán még könnyebben kiolvasható a jelentése: valami eleme egy listának ha annak feje, vagy a farkának egy eleme:

```
% member(Elem, Lista):  Elem a Lista elemeként előfordul.
member(Elem, [Fej|Farok]) :-
    (   Elem = Fej
    ;   member(Elem, Farok)
    ).
```

Záró példaként bemutatjuk a `member` egy hasznos kiterjesztését, a `select/3` eljárást, amelynek első két argumentuma megegyezik a `member`-ével, a harmadikban viszont visszaadja a kiválasztott elem elhagyásával keletkező listát (SICStus Prologban szintén a `library(lists)` könyvtár része).

```
% select(Elem, Lista, Marad):  Elem a Lista elemeként
% előfordul, kihagyása után marad a Marad lista.
select(Elem, [Elem|Marad], Marad).
select(Elem, [Egyéb|Farok], [Egyéb|Farok1]) :-
    select(Elem, Farok, Farok1).
```

A `select` eljárást legtöbbször a `member`-hez hasonlóan használjuk, tehát egy adott lista elemeit és a fennmaradó listákat soroltatjuk fel. A `select(E, L, M)` eljárás azonban használható visszafelé is: adott `E` és `M` esetén előállítja az összes lehetséges `L` listát, amelyből az `E` elhagyásával `M`-hez jutunk; tehát az adott `M` listába minden lehetséges módon beszúrja az `E` elemet:

```
| ?- select(E, [1,2,3], M).
E = 1, M = [2,3] ? ;
E = 2, M = [1,3] ? ;
E = 3, M = [1,2] ? ;
no
| ?- select(1, L, [2,3]).
L = [1,2,3] ? ;
L = [2,1,3] ? ;
L = [2,3,1] ? ;
no
| ?-
```

3.4.3. Listák összefűzése

Egy másik klasszikus listakezelő Prolog eljárás a két lista összefűzését végző `append/3` (ez szintén megtalálható a SICStus `lists` könyvtárban):

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

Az első klóz a rekurzió leállítását végzi: egy üres listát egy `L` lista elé fűzve az eredmény `L` maga. A második klóz arról az esetről szól amikor az első lista nem-üres: tekintsük az első lista farkát (ez lesz `L1`) és ezt fűzzük össze rekurzívan a második listával, ennek eredményét jelöljük `L3`-mal. Ez utóbbi elé helyezve az első lista fejét (`[X|L3]`) kapjuk az eredményt. Egy funkcionális nyelv esetén, mint pl. a LISP-ben, ezt a legutolsó műveletet az `append` rekurzív hívása után lehet csak elvégezni, ezért sokan az `append` második klózát így írnák:

```
append([X|L1], L2, L12) :-
    append(L1, L2, L3), L12 = [X|L3].
```

Mint már láttuk, az egyenlőség a Prologban kiküszöbölhető, és így kerül az eljárásfejbe, a kimenő argumentum-pozícióba a lista-kifejezés. Nézzük most meg, hogyan is hajtódik végre az eredeti tömör `append` eljárás!

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

Az első klózzal való illesztés nyilvánvalóan sikertelen, míg a rekurzív klóz fejével sikeres. Az illesztés eredménye egyrészt az, hogy szétszedjük az első listát: `X = 1, L1 = [2]`, változatlanul továbbadjuk a második listát: `L2 = [3,4]`, és végül a harmadik, kimenő argumentumba egy részlegesen kitöltött listát helyettesítünk: `L = [1|L3]`. Vegyük észre, hogy itt `L3` egy még behelyettesítetlen, ismeretlen mennyiség. Az illesztést követően redukáljuk az eredeti hívást az alkalmazott klóz törzsére:

```
append([2], [3,4], L3).
```

Az `L3` változó ebben a rekurzív hívásban kap értéket, `L3 = [2,3,4]`, és ennek következtében alakul ki az eredeti hívás végeredménye: `L = [1|L3] = [1,2,3,4]`. A Prolog változó- és egyesítés-fogalma így azt tette lehetővé, hogy az `1` listaelemet már akkor elhelyezzük az eredménylista fejében, amikor annak farka még nem állt rendelkezésre. Ez az `append` eljárás esetében azért is különösen fontos, mert így az eljárás jobbrekurzívvá vált, azaz saját magát csak a törzs utolsó hívásaként hívja vissza. A jobbrekurzív hívásokat ugyanis a Prolog rendszer nem rekurzióval, hanem a hagyományos nyelvek ciklusának megfelelő módon valósítja meg, amivel lényegesen csökken a futás memória- és időigénye (lásd 4.2).

A tömörebb `append` változatnak vannak más előnyei is. Nézzük például a következő ellenőrző jellegű hívást!

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

Ez azonnal meghiúsul, mert sem az első, sem a második klózzal nem illeszthető (az utóbbihoz az első és harmadik argumentumban levő listák fejének meg kellene egyeznie, és ez itt most nem teljesül). Tehát egyetlen redukciós lépésben kiderült, hogy az adott kérdésre nemleges a válasz. Ha az `append` másik változatával futtatjuk a fenti kérdést, akkor először kiszámolódik az első két lista összefűzöttje, és csak azután vizsgáljuk meg, hogy ez egyenlő-e a harmadik listával, ami tetemes időveszteség lehet.

Az `append` nemcsak két lista összefűzésére használható, hanem egy lista szétbontására is. Ehhez úgy kell meghívnunk, hogy a harmadik argumentum lesz a bemenő, itt adjuk meg a szétszedendő listát, és az első két argumentum lesz a kimenő: az a két lista, amelynek összefűzöttje a harmadik argumentumban megadott lista, pl.

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

Az első választ az `append` első klózával való illesztés adja. Visszalépéskor illesztünk a második klózzal, ami a `A = [1|L1]` behelyettesítéssel jár, és redukáljuk a problémánkat az `append(L1, B, [2,3,4])` hívásra. Amikor ez az első klózzal illesztődik, akkor kapjuk az eredeti probléma második megoldását, majd további visszalépések esetén a többi megoldást is.

A 3.4 ábra a fenti `append` hívás keresési fáját mutatja be.

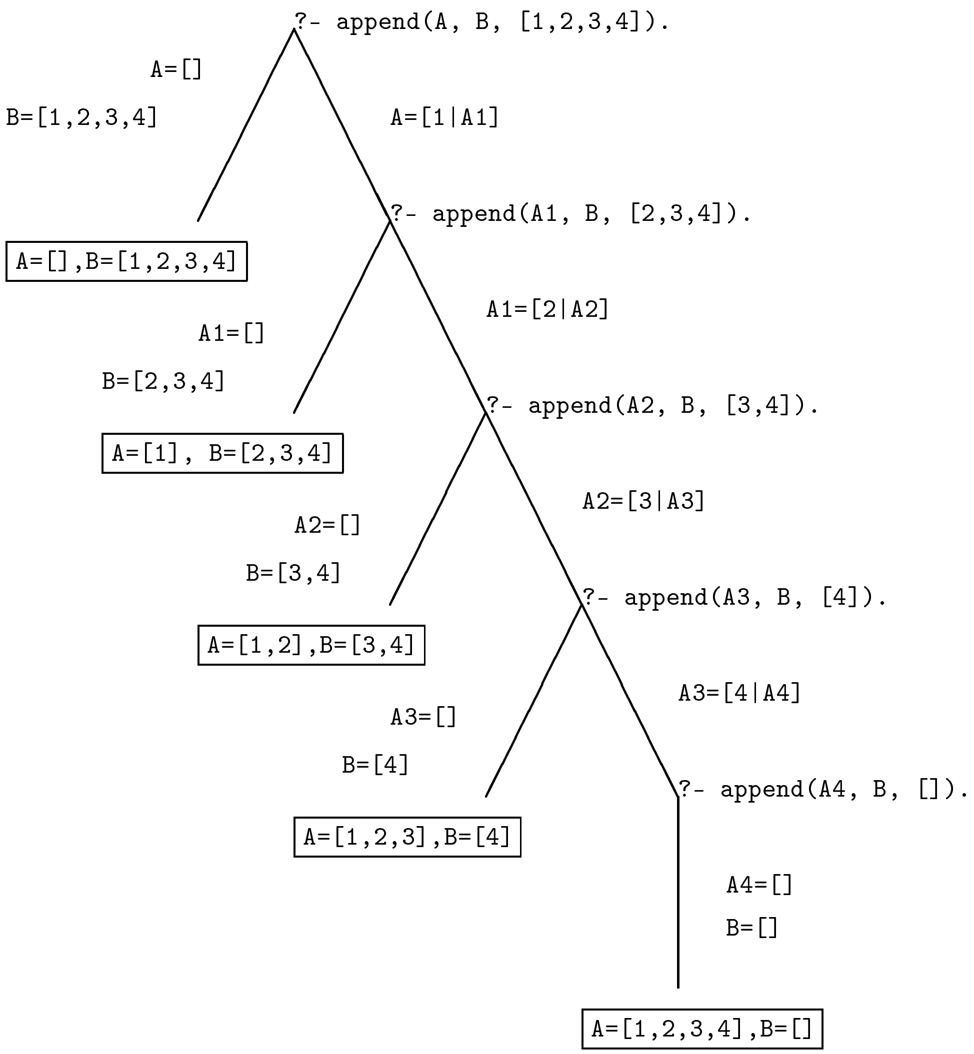
Az `append` szétbontó használata segítségével rendkívül egyszerűen tudunk összetettebb feladatokat is megoldani. Keressük meg például egy számlista azon elemeit, amelyek párban, ismétlődve fordulnak elő:

```
% párban(Lista, Elem): A Lista számlistának Elem olyan
% eleme, amely egy ugyanilyen értékű elemmel szomszédos.
párban(L, E) :-
    append(_, [E,E|_], L).
| ?- párban([1,8,8,3,4,4], E).
E = 8 ? ;
E = 4 ? ;
no
```

Itt az `append` eljárást szétbontó módban használtuk, úgy hogy a második részlistáról megköveteltük, hogy az két egyforma elemmel kezdődjék.

Végül nézzük meg hogyan tudunk egy listát Prologban megfordítani. Az első megoldásunk egyszerű, de nem hatékony (az ún. naive reverse megoldás):

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



3.4. ábra: Az `append` futás keresési (végrehajtási) fája

A rekurzió során itt egyre hosszabb listákhoz fűzünk hozzá egy egyelemű listát. Vegyük észre, hogy az `append` futási ideje az első argumentum hosszával arányos, ezért az `nrev` eljárás futási ideje a lista hosszával négyzetesen arányos.

A második, lineáris idejű megoldás egy segédeljárást használ — ez megtalálható a SICStus `lists` könyvtárban.

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

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

Ennek a megoldásnak az elemzését az olvasóra bízzuk.

3.4.4. Gyakorló feladatok

GY1.

Egy célsorozat keresési fája egy olyan irányított gráf, amelynek csúcsaiban célsorozatok vannak és két csúcs között akkor megy él, ha a kiinduló csúcs célsorozatából egyetlen (Prolog) redukciós lépéssel eljuthatunk a cél csúcs célsorozatába. Az élek a redukció során alkalmazott klóz sorszámával vannak címkézve. A fa gyökerében a teljes kiindulási célsorozat van, az üres célsorozatot egy négyzettel jelöljük.

Rajzolja fel az alábbi célsorozatok keresési fáját! A fa nem üres célsorozatot tartalmazó leveleinél jelezze a visszalépést! (A `select`, `write` és `append` szavakat rövidítheti (`s`, `w`, `a`).)

1. `select(A, [4,2,3,8], [4,2|B]), write(A-B).`
2. `select(U, [b,c], V]), write(U-V).`
3. `append([A|B], C, [1,2]), write([A|B]-C).`
4. `append(A, [B|C], [d,t]), write(A-[B|C]).`

3.4.5. Körmentes út keresése

A Prolog lista-fogalmának felhasználásával most egy olyan Prolog eljárást írunk meg, amely két város között egy körmentes útvonalat keres. A definiált eljárás magát az útvonalat is kiadja, egy olyan lista formájában, amelynek első eleme a kiindulási hely, utolsó eleme a célpont, és szomszédos elemei között van `járat`, egyik vagy másik irányban. Az egyszerűség kedvéért a távolságok számításától most eltekintünk:

```
% útvonal_2(N, A, B, L): Az A-ból B-be menő pontosan N
%   szakaszból álló körmentes út állomásainak listája L.
útvonal_2(N, Honnan, Hová, Út) :-
    útvonal_2(N, Honnan, Hová, [Honnan], Út).

% útvonal_2(N, A, B, K, L): L az A-ból B-be menő pontosan
% N szakaszból álló körmentes, K elemein át nem menő út.
útvonal_2(0, Hová, Hová, _, [Hová]) :- !.
útvonal_2(N, Honnan, Hová, Kizártak, [Honnan|Út]) :-
    N > 0, N1 is N-1,
    járatszakasz(Honnan, Közben, _),
    \+ member(Közben, Kizártak),
    útvonal_2(N1, Közben, Hová, [Közben|Kizártak], Út).
```

Az `útvonal_2/4` eljárás definiálásához egy `útvonal_2/5` segédeljárást használunk, amelynek új bemenő argumentuma a kizárt pontok listája, azaz azoknak a városoknak a felsorolása, amelyeken az útvonal már keresztülment. Ennek az argumentumnak a „kezdőértéke” a `[Honnan]` egyelemű lista.

Az `útvonal_2/5` eljárás szerkezete nagyon hasonlít a korábbi `útvonal/4` szerkezetéhez. Az első klóz most egy szabály lett, törzsét a `!` (vágó) beépített eljárás alkotja. A vágó eljárás végrehajtásakor mindig sikeresen lefut, de a Prolog keresési fának bizonyos ágait levágja; példánkban az `útvonal_2` második klózához vezető ágat. A két klóz eleve kizárja egymást (az elsőben `N = 0`, míg a másodikban `N > 0`), így a vágó nem módosítja a program keresési terét, csak olyan ágat vág le, amely biztosan meghiúsulna (ez az ún. zöld vágó). A vágó szerepe itt az, hogy a Prolog végrehajtási mechanizmust egyszerűsíti, a program futását gyorsítja, mivel a rendszer nem tart fenn egy felesleges választási pontot az `útvonal_2` eljárás meghívásakor.

A második klózban egy új feltétel van, a `\+ member(Közben, Kizártak)`, amelynek jelentése: `Közben` nem eleme a `Kizártak` listának. A `\+` művelet a Prologban a negáció egy gyenge formáját jelenti. A Prolog negáció-fogalmának gyengesége abban áll, hogy rossz eredményt adhat, ha a negálni kívánt eljárást nem tömör (nem teljesen behelyettesített) argumentumokkal hívjuk meg. A fenti programban ez a veszély nem áll fenn, hiszen a `member` mindkét argumentuma értékkel fog bírni mire a hívás sorra kerül.

A `!` és `\+` eljárások részletesebb megvitatására még visszatérünk.

Nézzük most az `útvonal_2/5` új argumentumainak szerepét! A 4. argumentum, a `Kizártak` lista, bemenő jellegű. Erre nincs szükség az első klózban (névtelen változó áll a helyén), a másodikban viszont a `\+ member` hívásban használjuk, majd a `Közben` várossal kiegészítve adjuk tovább az `útvonal_2/5` rekurzív hívásának. Ebben tehát szintén felépül az útvonal, csak fordított sorrendben, hiszen az új elemeket a régiek elé rakjuk.

Az ötödik argumentum, az `Út`, kimenő, amely az első klózban a megfelelő egyelemű listát kapja értékül. A második klózban a fejben behelyettesítjük az eredménylista fejét, majd a farok kitöltését a rekurzív hívásra bízzuk. Ez hasonló ahhoz ahogy az `append` eljárás építi a harmadik argumentumában levő listát.

Nézzük most az `útvonal_2/4` eljárás egy futását!

```
| ?- útvonal_2(2, 'Budapest', _, Út).
Út = ['Budapest','Bécs','Berlin'] ? ;
Út = ['Budapest','Bécs','Párizs'] ? ;
no
```

Vegyük észre, hogy a korábbi Budapest-Prága-Budapest kört is tartalmazó útvonal most nem szerepel a megoldások között.

3.4.6. Rekord-struktúrák

A lista, mint adatstruktúra, azt teszi lehetővé hogy két adatból alkossunk egy összetett objektumot. A Prolog nyelv egy olyan általánosabb adat-összetételi mechanizmust is tartalmaz, amellyel akárhány adatból egyetlen összetett és névvel ellátott objektumot képezhetünk. Az általános adatépítő jelölés formája *rekordnév*(A_1, A_2, ..., A_N)

Itt *rekordnév* egy tetszőleges névkonstans, míg A_1, ..., A_N tetszőleges, egyszerű vagy összetett objektumok. Ez tehát megfelel az algoritmikus nyelvek struktúra-fogalmának, azzal a különbséggel, hogy a Prolog a rekordban a mezők értékei előtt egy, a rekord fajtájára utaló rekordnevet is tárol, és az egyes mezőket nem kiválasztókkal, hanem felsorolási sorrendjükkel különbözteti meg.

Például, egy személy néhány adatának ábrázolására használható a következő adatszerkezet:

```
szemely( <Vezetéknév>, <Keresztnév>, <Szül. év>)
```

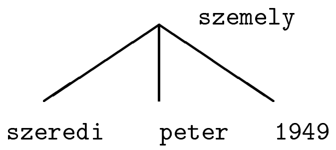
pl.

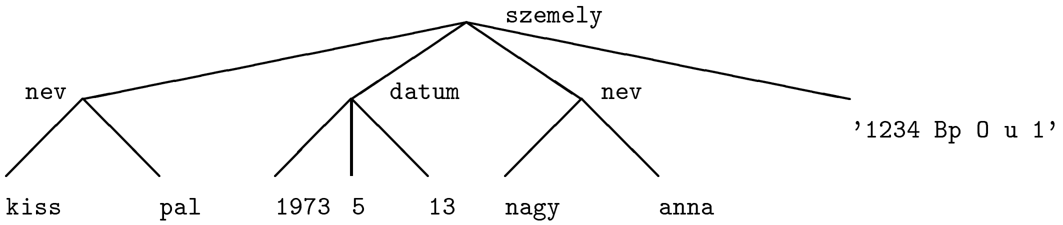
```
szemely(szeredi, peter, 1949).
```

Természetesen az összetett adatszerkezetek egymásba is ágyazhatók, pl. egy bonyolultabb személy-struktúra lehet a következő:

```
szemely(nev(kiss,pal), datum(1973, 5, 13), nev(nagy,anna), '1234 Bp O u 1').
```

A Prolog adatszerkezeteit egy olyan fastruktúrával tudjuk szemléltetni, amelynek csomópontjai a rekordnévvel vannak megcímkézve, levelei pedig nem-összetett adatstruktúrákkal (azaz számokkal, névkonstansokkal ill. változókkal). Két `szemely` példánk fa-alakja a következő:





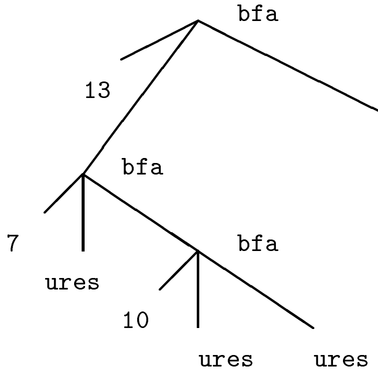
Gyakori, hogy az adatstruktúrák rekurzív módon vannak egymásba ágyazva. Például egy egész számokból képzett, rendezett bináris fa Prologban a következő módon ábrázolható:

bfa(*Szám*, *BalFa*, *JobbFa*)

Itt *BalFa* és *JobbFa* argumentumokban vagy hasonló `bfa` struktúra, vagy az üres fát jelző `ures` névkonstans áll. Példa:

```
bfa(13, bfa(7, ures, bfa(10, ures, ures)), bfa(20, ures, ures))
```

Grafikusan:



Az alábbi eljárás azt dönti el, hogy egy adott érték megtalálható-e egy fenti ábrázolású rendezett bináris fában.

```
% faban(Ertek, Fa):  Ertek megtalálható a Fában.
faban(Ertek, bfa(Ertek,_,_)).
```

```
faban(Ertek, bfa(Ertek0,Balfa,_)) :-
      Ertek < Ertek0,
      faban(Ertek, Balfa).
faban(Ertek, bfa(Ertek0,_,Jobbfa)) :-
      Ertek > Ertek0,
      faban(Ertek, Jobbfa).
```

3.4.7. Gráfok mint adatstruktúrák

Egy kicsit összetettebb példaként vizsgáljuk meg, hogyan ábrázolhatunk egy súlyozott gráfot Prolog adatstruktúraként. Matematikailag egy súlyozott gráf nem más mint élek halmaza, ahol egy élet három adat határoz meg: az él kezdő- és végpontja, valamint a hozzá rendelt súly. Természetesen adódik, hogy egy gráfot élei listájával ábrázoljunk, ahol egy él egy háromargumentumú rekord-struktúra.

A Prolog típustalan nyelv, ezért adatábrázolási döntésünk csak az adatokat felhasználó eljárások alakjában tükröződik. (Vannak típusos logikai programozási nyelvek, pl. a Mercury nyelv, amelyekben típusdeklarációkkal kell leírnunk a használt adatstruktúrákat.) Nézzünk most egy olyan Prolog eljárást, amely felsorolja egy adott irányítatlan gráf éleit, a fenti adatábrázolás mellett!

```
éle(Gráf, Kezdő, Vég, Súly) :-
    member(él(Kezdő, Vég, Súly), Gráf).
éle(Gráf, Kezdő, Vég, Súly) :-
    member(él(Vég, Kezdő, Súly), Gráf).

| ?- éle([él(a,b,5),él(c,a,2)], a, V, S).
V = b, S = 5 ? ;
V = c, S = 2 ? ;
no
```

Az `éle/4` eljárás meghívásában egy két élből álló gráfot adtunk meg egy kételemű listával. Az eljárást két szabállyal definiáltuk, az első felsorolja a gráf éleit az adott irányítás mellett, míg a második a fordított irányítással. Az összetett adatok építését, átadását és komponensekre szedését a Prolog nyelv már jól ismert mintaillesztése végzi el. Ennek bemutatására kövessük a fenti kérdés végrehajtásának egy rövid szakaszát! Kérdésünk először az `éle/4` eljárás első szabályára illesztődik, ennek eredményeként a előáll a `member` hívása:

```
member(él(a, V, S), [él(a,b,5)|(...)])   (*)
```

A második argumentumban a kételemű listát a szabványos `[(...)|(...)]` alakba írtuk át. Ezt a hívást próbáljuk a `member` eljárás első klózára (`member(Elem, [Elem|_])`) illeszteni, azaz vele azonos alakra hozni. Az illesztés sikeres, a `V = b, S = 5` behelyettesítéssel, ami az első megoldást adja. A fenti `member` hívás további futása nem ad újabb eredményt, így az `éle` eljárás második klózának alkalmazásával folytatjuk a keresést, ami egy másik `member` hívásra redukálva adja a második megoldást.

Az olvasó bizonyára észrevette, hogy korábbi mintapéldánk, a városok közötti útvonal-keresés is valójában egy irányítatlan gráfban való útkeresést jelent. Oldjuk meg most mintapéldánkat úgy, hogy a `járat`-okat, azaz a gráfot az imént ismertetett ábrázolásban adjuk meg.

```
% útvonal_3(N, G, A, B, L): A G gráfban van egy  A-ból
% B-be menő N szakaszból álló L út.
útvonal_3(0, _Gráf, Hová, Hová, [Hová]).
útvonal_3(N, Gráf, Honnan, Hová, [Honnan|Út]) :-
        N >0, N1 is N-1,
        select(Él, Gráf, Gráf1),
        él_végpontok(Él, Honnan, Közben),
        útvonal_3(N1, Gráf1, Közben, Hová, Út).
```

```
% él_végpontok(Él, A, B): Az Él irányítatlan él
% végpontjai A és B.
él_végpontok(él(A,B,_), A, B).
él_végpontok(él(A,B,_), B, A).

| ?- _Gráf = [él('Budapest','Bécs',245),
             él('Budapest','Prága',515),
             él('Bécs','Berlin',635),
             él('Bécs','Párizs',1265)],
     útvonal_3(2, _Gráf, 'Budapest', _, Út).
Út = ['Budapest','Bécs','Berlin'] ? ;
Út = ['Budapest','Bécs','Párizs'] ? ;
no
```

Itt a korábbi `select` eljárást használtuk a gráf egy élének kiválasztására, majd az `él_végpontok` eljárást az él esetleges forgatására. Ebben a megoldásban elmarad a bejárt útvonal gyűjtése, ehelyett az útvonal részévé választott éleket elhagyjuk a gráfból (`select`). Ez a megoldás nem garantálja a körmentességet, csak azt, hogy minden élet csak egyszer járunk be.

Befejezésül elmondjuk, hogy a Prolog lista-struktúrái is az általános rekord-struktúrákra vezetődnek vissza: egy `[X|L]` lista nem más, mint a `.(X,L)` rekord-struktúra. Ez egyben példa arra is, hogy rekordnévként nem csak alfanumerikus névkonstansok használhatók.

3.4.8. Az egyesítési algoritmus

Általánosan elmondhatjuk, hogy a Prolog olyan fastruktúrákat kezel adatként, amelyek csomópontjai nevekkel címkézettek (ezek a rekordnevek), míg a levelekben szám- és névkonstansok valamint változók szerepelhetnek. Ez a fastruktúra az adatkifejezések szabványos alakja, az egyesítési algoritmus is ténylegesen ezen a fastruktúrán dolgozik.

Az egyesítési algoritmus két azonos hosszúságú kifejezés-sorozatot kap, pl. a hívás és a fej argumentumainak sorozatát. Az egyesítés lehet sikeres, ekkor változó-behelyettesítések egy (esetleg üres) sorozatát adja eredményül, vagy lehet sikertelen.

Legyenek az egyesítendő kifejezés-sorozatok: $A_1, \ldots, A_N$ ill. $B_1, \ldots, B_N$. Az egyesítést a következőképpen végezzük:

1. Ha N>1, akkor megkíséreljük az A_1 és B_1 egyelemű sorozatok egyesítését. Ezután elvégezzük a kapott behelyettesítést az $A_2, \ldots, A_N$, ill. $B_2, \ldots, B_N$ sorozatokon, majd megkíséreljük ezeket a sorozatokat is egyesíteni. Az eredő egyesítés akkor és csak akkor sikeres, ha mindkét rész-egyesítés sikeres. A eredményezett behelyettesítést úgy kapjuk meg, hogy a kapott két behelyettesítés eredőjét vesszük.

A továbbiakban feltételezhetjük, hogy N=1.

2. Ha A_1 és B_1 azonos változók vagy konstansok, akkor az egyesítés sikeres, üres behelyettesítéssel.
3. Egyébként, ha A_1 változó, akkor az egyesítés sikeres az $A_1 = B_1$ behelyettesítéssel.
4. Egyébként, ha B_1 változó, akkor az egyesítés sikeres a $B_1 = A_1$ behelyettesítéssel.
5. Ha A_1 és B_1 azonos nevű és argumentumszámú összetett kifejezések, akkor rekurzívan meghívjuk az egyesítési algoritmust A_1 és B_1 argumentumainak sorozatára. Egyesítésük akkor és csak akkor sikeres, ha az argumentum-sorozatok egyesítése sikerül.
6. Minden más esetben az egyesítés meghiúsul.

Megjegyzés: az egyesítés fogalmának tételbizonyításbeli definíciója szerint, valahányszor egy $X = Y$ behelyettesítést hajtunk végre, ellenőrizni kell, hogy az X változó előfordul-e az Y kifejezés belsejében. Ez az ún.

előfordulás-ellenőrzés *(occurs check)*). Ha az előfordulás-ellenőrzés sikeres, akkor, az egyesítésnek elvben meg kell hiúsulnia. Hatékonysági okokból a legtöbb Prolog elhagyja ezt a vizsgálatot, hiszen alkalmazásakor a behelyettesítés konstans idejű művelet helyett a behelyettesített kifejezés méretével arányos idejű műveletté válik.

Példaként nézzük most végig, hogy működik az egyesítési algoritmus egy korábban már vizsgált esetre (*), a `member(él(a, V, S), [él(a,b,5),él(c,a,2)])` hívás és a `member(Elem, [Elem|_])` klózfej egyesítésére. Az argumentum-sorozatok tehát a következők (szabványosított lista-jelölés mellett):

A_1: `él(a, V, S),` A_2: `.(él(a,b,5),.(él(c,a,2),[]))`
B_1: `Elem,` B_2: `.(Elem,_)`

A_1 és B_1 egyesítése a 4. pont szerint sikeres, az `Elem = él(a, V, S)` behelyettesítéssel. Ezt a behelyettesítést elvégezzük B_2-n, ellenőrizzük, hogy A_2 és B_2 azonos nevű és argumentumszámú (5. pont), majd rekurzívan alkalmazzuk az egyesítési algoritmust ezek argumentum-listáira:

A_1': `él(a,b,5),` A_2': `.(él(c,a,2),[])`
B_1': `él(a,V,S),` B_2': `_`

A_1' és B_1' egyesítése most ismét az 5. pont alkalmazását igényli, majd egy újabb rekurziót az egyesítésben. Ennek eredménye sikeres, a `V = b, S = 5` behelyettesítéssel. A_2' és B_2' egyesítése is sikeres a 4. pont szerint, a behelyettesítést itt felesleges megjegyeznünk, mivel B_2' névtelen változó lévén biztosan nem fordul elő máshol sem a klózban sem a célsorozatban. Így az eredő egyesítés sikeres, az `Elem = él(a, b, 5), V = b, S = 5` helyettesítéssel.

3.4.9. Operátorok

Korábbi példáinkban szerepeltek aritmetikai beépített eljárások hívásai, pl. `T is T1+T2`. Itt az `is` és a `+` ún. **operátorok**, amelyek hívások ill. adatstruktúrák infix jelöléssel való írását teszik lehetővé. Az operátorok is „szintaktikus édesítőszerek”, mert a kifejezés beolvasását követően eltűnnek, a rendszeren belül a kifejezés szabványos alakú lesz. A fenti példa esetén ez az `is(T, +(T1,T2))` alak, tehát az `is/2` eljárás meghívásáról van szó, amelynek második argumentuma egy `+` nevű két-argumentumú rekord-struktúra. Hasonlóképpen a `T1+T2>1000` hívás szabványos alakja: `>(+(T1,T2), 1000)`.

Az `is/2`, `>/2`, és a többi aritmetikai beépített eljárás különlegesen kezeli az argumentumait: a bennük szereplő (akár többszörösen is) összetett adatstruktúrákat aritmetikai kifejezésnek tekintik, és ki is értékelik. Az aritmetikai eljárásokon kívül viszont ezek a kifejezések közönséges adatstruktúrák, kezelésük a szokásos egyesítési algoritmussal történik. Ezért pl. ha a 3.1.4-beli `útvonal/4` rekurzív hívásának első argumentumában a kiszámolt `N1` helyett az `N-1` kifejezés szerepelne, akkor ez sosem lenne egyesíthető a rekurziót leállító klóz első argumentumával, amely `0` (hiszen számkonstans és összetett adatstruktúra sosem egyesíthető).

Az aritmetikai operátorok beépítettek, de a Prolog programozó is definiálhat új operátort egy ún. **operátor-deklaráció** segítségével. Ennek alakja:

`:- op(`**prioritás**`,` **fajta**`,` **operátornév**`).`

Az **operátornév** tetszőleges névkonstans lehet, bár többnyire csak az aposztróf-jel nélkül írható alakok használatosak, azaz írásjelek és ezek sorozatai, ill. alfanumerikus névkonstansok.

A **prioritás** egy egész szám, amely a több operátort tartalmazó kifejezések zárójelezési sorrendjét határozza meg: a kisebb prioritású operátorok előbb zárójeleződnek mint a nagyobbak. A **fajta** jellemző azt határozza meg, hogy az azonos prioritású operátorok hogyan zárójeleződjenek, pontosabban azt, hogy az adott operátor operandusként zárójelezés nélkül fogadjon-e el saját magával azonos prioritású operandust. Infix operátorok esetén a **fajta** lehet `yfx`, amely balról jobbra való zárójelezést ír elő (ilyen a `+`, `-`, stb.), `xfy`, amely jobbról balra zárójeleződik, vagy `xfx` (pl. a `<`), amely nem engedi az azonos prioritású operátor (zárójelezetlen) használatát egyik oldalán sem.

Az infix operátorok mellett léteznek prefix és posztfix operátorok is, amelyek egyargumentumú adatstruktúrákká alakulnak. A prefix operátort az argumentuma elé, a posztfixet pedig mögé írjuk. A prefix operátor fajtája `fx` vagy `fy`, a posztfixé `xf` vagy `yf` lehet. Az `x` operandus-jelölés itt is azt jelzi, hogy önmagával azonos prioritású operátort nem fogad el, míg az `y` jelölés ezt lehetővé teszi.

Hangsúlyozzuk, hogy az operátorok csak a programozó munkáját könnyítő jelölések, az illesztést mindig a szabványos alakon végzi a rendszer. Tehát például az `a+b+c` kifejezés nem egyesíthető az `a+X`-szel, mert az előbbi zárójelezése `(a+b)+c`, tehát a külső `+` rekord első argumentumai nem egyesíthetőek. (A nehezen olvasható szabványos adatstruktúra-alak helyett többnyire elegendő, ha a teljesen zárójelezett operátoros alakon gondoljuk végig az egyesíthetőséget.)

Megjegyezzük, hogy a Prolog klózban használt összekötő jelek, így a `:-`, a vessző, a `\+` (negáció) mind szabványos operátorok, és így maga a klóz is egy Prolog kifejezésként olvasódik be. Ezért van az, hogy az operátor-jelölés használható nemcsak a struktúra-kifejezésekben hanem az eljáráshívásokban is (pl. `is/2`). Azt a tényt, hogy a klózok kifejezésként is felírhatók, az ún. **adatbázis-kezelő** beépített eljárások is kihasználják.

1200	`xfx`	`:-, -->`
1200	`fx`	`:-, ?-`
1100	`xfy`	`;`
1050	`xfy`	`->`
1000	`xfy`	`','`
900	`fy`	`\+`
700	`xfx`	`<, =, \=, =.., =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \==, is`
500	`yfx`	`+, -, /\, \/`
400	`yfx`	`*, /, //, rem, mod`[1]`, <<, >>`
200	`xfx`	`**`
200	`xfy`	`^`
200	`fy`	`-`[2]`, \`

3.3. táblázat: A BEÉPÍTETT SZABVÁNYOS OPERÁTOROK

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

3.4. táblázat: EGYÉB BEÉPÍTETT OPERÁTOROK

A 3.3 és a 3.4 táblázat a SICStus Prolog beépített operátordefinícióit adja meg. Itt jegyezzük meg, hogy a vessző jel (`,`) több értelemben is szerepel a Prolog szintaxisában: egyrészt mint 1000-s prioritású operátor, másrészt pedig mint az argumentum-lista szeparátor jele. Ezért az argumentum-listában zárójelezés nélkül legfeljebb 999-es prioritású operátorok fordulhatnak elő, az ennél nagyobb prioritású operátort tartalmazó argumentum-kifejezést zárójelezni kell.

3.4.10. Az operátorok alkalmazása

A Prolog általános operátorfogalma többféle módon is megkönnyíti a programfejlesztési munkát.

Először is az operátorfogalom teszi lehetővé, hogy az aritmetikai beépített eljárásokban a megszokotthoz hasonló módon írjuk le számításainkat, pl.

```
X is N*8 + A mod 8
```

Másodszor, az operátorfogalom teszi azt is lehetővé, hogy a Prolog szabályokat, vezérlési szerkezeteket le tudjuk írni mint Prolog kifejezésekként. Ez a homogén szintaxis nemcsak a rendszer megvalósítóinak munkáját könnyíti, de számos ún. meta-programozási lehetőségre ad módot. Például lehetőség van ún. dinamikus predikátumok létrehozására, amelyekhez futási időben adhatunk hozzá új klózokat, pl.

[1] `sicstus` módban 300 `xfx` operátor
[2] `sicstus` módban 500 `fx` operátor
[3] `iso` módban 200 `fy` operátor

```
... , asserta( (p(X):-q(X),r(X)) ), ...
```

Harmadszor, operátorok segítségével a Prolog programok természetesebbé, olvashatóbbá tehetők. Például a bevezető fejezetben ismertetett nagyszülője predikátum a következő alakba írható át:

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

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

Negyedszer, operátorokat használhatunk az adatok természetesebb formában való felírására is. Például, ha a `'.'` jelet operátornak deklaráljuk, akkor kémiai vegyületek leírására a szakmai nyelvhez közel álló jelölést használhatunk:

```
:- op(100, xfx, [.]).

...  h.2-s-o.4 ....
```

Végezetül az operátorok teszik lehetővé a „klasszikus” szimbolikus kifejezésfeldolgozást, például a szimbolikus deriválást (lásd a P13 példaprogramot, 69. oldal).

3.4.11. Gyakorló feladatok

GY2.

Írja fel az alábbi kifejezéspárok alapstruktúra-alakját (azaz szintaktikus édesítőszerek nélküli formáját) vagy rajzolja fel a hozzájuk tartozó fastruktúrákat! Állapítsa meg, hogy a két kifejezés egyesíthető-e, és ha igen, milyen behelyettesítéssel!

1.	`.(A+C*A,[C])`	`[w+B,3]`
2.	`f(2*A+B, [C], B*C)`	`f(U+V, .(V,_), U)`
3.	`f(1*Y-U, [V], V*U)`	`f(A-3, B, A)`
4.	`[f(B*2, A, [B|A]),_]`	`[f(Y*Y, [], X)|X]`

3.5. A Prolog szintaxis összefoglalása

A korábbiakban megadtuk a Prolog programelemek (klózok, célok) szintaxisát, leírva hogy ezek hogyan épülnek fel Prolog kifejezésekből a ‘`:-`’, ’`,`’, ’`;`’ stb. összekötő jelek segítségével.

A Prolog rendszerek valójában minden programelemet Prolog kifejezésként olvasnak be. Ezt azért lehet megtenni, mert a használt összekötő jelek mind szabványos operátorok. A programszöveg beolvasott kifejezéseit ezután a (legkülső) funktoruk alapján osztályozzák mint szabályt, tényállítást, stb. Ez pl. azt jelenti, hogy egy `p :- q.` szabályt programjainkban írhatjuk `(p :- q).`, vagy akár `:-(p, q).` alakban is.

A Prolog kifejezések szintaxisát egy ún. kétszintű nyelvtannal adjuk meg. Az alábbi <kifejezés *N*> nyelvtani szabályt *N* 1 és 1200 közötti minden értékére le kell írni, úgy hogy a szabály törzsében is minden *N* ill *N-1* hivatkozást a megfelelő számra cserélünk.

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

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

Megjegyzések:

1. A <záró-pont> egy olyan pont jel, amit legalább egy nem látható karakter (szóköz, újsor, tabulátor) követ.

2. A <kifejezés N> szabályában szereplő <köz> legalább egy nem látható karaktert jelöl, de csak abban az esetben ha az őt követő kifejezés `(`-lel kezdődik.

3. A `{` <kifejezés> `}` szerkezet azonos a `{}(`<kifejezés>`)` struktúrával, ennek felhasználását lásd pl. a DCG nyelvtanoknál, a 7.5.2 fejezetben.

4. Egy <füzér> `"` jelek közé zárt karaktersorozat, általában a karakterek kódjainak listájával azonos (lásd még 6.3).

5. <név>-ként a következő jelsorozatok megengedettek

- kisbetűvel kezdődő alfanumerikus jelsorozat (ebben megengedve kis- és nagybetűt, számjegyeket és aláhúzásjelet);
- egy vagy több ún. speciális jelből (`+-*/\^<>=‘~:.?@#$&`) álló jelsorozat;
- az önmagában álló `!` vagy `;` jel;
- a `[]` `{}` jelpárok;
- idézőjelek (`'`) közé zárt tetszőleges jelsorozat, amelyben `\` jellel kezdődő escape-szekvenciákat is elhelyezhetünk.

6. Egy <változó> nagybetűvel vagy aláhúzással kezdődő alfanumerikus jelsorozat lehet.
7. <természetes szám> lehet
   - (decimális) számjegysorozat;
   - 2, 8 és 16 alapú számrendszerben felírt szám, ilyenkor a számjegyeket rendre a `0b`, `0o`, `0x` karakterekkel kell prefixálni;
   - karakterkód-konstans `0'c` alakban, ahol `c` egyetlen karakter, esetleg escape szekvencia formájában megadva (az escape szekvenciák formája megegyezik a C nyelvben találhatókéval).
8. A <lebegőpontos szám> mindenképpen tartalmaz tizedespontot, ennek mindkét oldalán legalább egy számjeggyel. Az `e` vagy `E` betűvel jelzett exponens nem kötelező.
9. Az egyes lexikai elemek között szabadon előfordulhatnak nem látható karakterek és megjegyzések. Kivétel: a struktúrakifejezés neve és az azt követő nyitózárójel között nem lehet más jel.
10. A megjegyzések két alakja megengedett:
    - A `%` százalékjeltől a sor végéig
    - A `/*` jelpártol a legközelebbi `*/` jelpárig.

A programelemként beolvasott kifejezések a következők lehetnek:

kérdés `?-` *`Cél`*`.`
(A *`Cél`*t lefuttatja, és a változó-behelyettesítéseket kiírja.)

parancs `:-` *`Cél`*`.`
(A *`Cél`*t csendben lefuttatja. Különféle deklarációkat parancsként helyezhetünk el a programban.)

szabály *`Fej`* `:-` *`Törzs`*`.`

nyelvtani szabály *`Fej`* `-->` *`Törzs`*`.` (Lásd a 7.5.2 fejezetet).

tényállítás (*Minden egyéb kifejezés tényállításnak tekintődik.*)

3.6. Típusok Prologban

A (szabványos) Prolognak nem része a predikátumok és argumentumaik típusának specifikációja. Bizonyos Prolog kiterjesztések (pl. Visual Prolog, Mercury) azonban bevezetik a típus fogalmát.

A típusok feltüntetése és a típushelyesség biztosítása nagymértékben elősegíti a Prolog programok olvashatóságát, karbantarthatóságát és az alapvető programhibák kiszűrését. Ezért célszerű összetettebb alkalmazásoknál a predikátumok argumentumainak típusát is feltüntetni egy ún. predikátumtípus-deklaracioban. Ehhez az is szükséges, hogy az argumentumok jellemzésére használt típusneveket is leírjuk egy ún. típusdeklarációban.

Ebben a jegyzetben a típusdeklarációk alakjára a Mercury nyelvben használt szintaxist használjuk, csekély módosítással. Mivel a használt Prolog rendszerek (SICStus, SWI) nem értelmezik a típusdeklarációkat, ezert a deklarációkat kommentbe kell ágyazzuk. A predikátumtípusokat célszerű a predikátum fejkommentjében megadni.

3.6.1. Típusdeklarációk

Alaptípusok: `int`, `float`, `number`, `atom`.

Univerzális típus: `any`.

Ezt akkor használhatjuk, ha nem kívánunk típusinformációt közölni.

Típus-konstrukció

```
:- type típusnév ---> típustörzs .
```

Itt ***típusnév*** vagy egy atom, vagy egy struktúrakifejezés csupa változó-argumentummal. A ***típustörzs*** pedig struktúrák és atomok pontosvesszőkkel elválasztott sorozata (lehet egyelemű). A struktúra argumentumaiban típusnevek állnak.

Példák:

```
:- type gyumolcs ---> alma ; korte; szilva.
    % egy gyümölcs típusú adat a fenti három atom valamelyike.

:- type szemely ---> szemely(atom, atom, int).
    % egy szemely típusú adat egy olyan háromargumentumú struktúra,
    % amelynek első két argumentuma atom, a harmadik egész.

:- type binarisfa ---> ures
        ;             bfa(int, binarisfa, binarisfa).

:- type list(T) --->   []
        ;              [T|list(T)].

:- type pair(T1, T2) ---> T1 - T2.
    % egy olyan '-' nevű kétargumentumú struktúra, amelynek első
    % argumentuma T1, második argumentuma pedig T2 típusú.
```

Tipusátnevezések:

```
:- type típusnév == típusnév .
```

Példák:

```
:- type hossz == int.

:- type assoc_list(KeyType, ValueType)
          == list(pair(KeyType, ValueType)).

:- type szotar == assoc_list(magyar, angol).
:- type magyar == atom.
:- type angol == atom.
```

3.6.2. Predikátum-deklarációk

```
:- pred predikátumnév(típusnév, ...)  .
```

Például:

```
:- pred member(T, list(T)).

:- pred append(list(T), list(T), list(T)).
```

Sokszor hasznos a típusinformációt kiegészíteni az használt adatáramlásra utaló ún. módinformációval:

:- mode *predikátumnév*(*módnév*, ...) .

Ebben a jegyzetben módnévként csak az `in` és `out` azonosítókat használjuk. Az `in` jelentése: híváskor teljesen kitöltött (nem tartalmaz változót). Az `out` jelentése: híváskor kitöltetlen változó, sikeres lefutás után kitöltött érték.

Egy predikátumra több móddeklaráció is vonatkozhat, pl.

```
:- mode append(in, in, in).          % ellenőrzésre
:- mode append(in, in, out).         % két lista összefűzésére
:- mode append(out, out, in).        % egy lista szétszedésére
```

Ha csak egy móddeklaráció van, összevonható a típusdeklarációval:

:- pred *predikátumnév*(*típusnév*::*módnév*, ...) .

Példa:

Ha az `append` eljárást csak listák összefűzésére használjuk, akkor a következő összevont deklarálációt írhatjuk fel:

```
:- predicate append(list(T)::in, list(T)::in, list(T)::out).
```

3.6.3. Módinformáció megadása a beépített eljárásoknál

A Prolog irodalomban szokásos még a módinformációt a fejkommentben megadni: az eljárás argumentumait a `'+'`, `'-'` vagy `'?'` jelekkel előzzük meg. A `+` jel bemenő argumentumot jelöl, a `-` jel kimenőt, míg a `?` jel azt jelzi, hogy az adott argumentumban változó és nem-változó értéke egyaránt elfogadható. Használatos még a `'@'` jel is, olyan összetett argumentum jelzésére, amely tisztán bemenő, azaz a benne szereplő változók közvetlenül nem helyettesítődnek be. Mi is ezeket a jeleket fogjuk használni a beépített eljárások ismertetésénél, és néhány más esetben is, amikor típus-információt nem akarunk közölni.

A beépített eljárások leírási formáját részletesebben lásd az 5.1. pontban.

4. fejezet

Programozási módszerek

Ez a fejezet néhány a Prolog nyelvre jellemző programozási módszert mutat be.

4.1. A keresési tér szűkítése

A keresési tér szűkítését, azaz a keresési fa egyes ágainak lemetszését alapvetően a vágó beépített eljárás segítségével végezhetjük el.

A vágót két célból használhatjuk:

- a megoldások halmazát ténylegesen módosítani akarjuk (ezt *vörös* vágónak nevezzük);
- a Prolog fordítóprogram tudomására akarjuk hozni, hogy bizonyos ágakon biztosan nincs a feladatnak megoldása (ez az ún. *zöld* vágó).

A zöld vágók azért fontosak, mert a választási pontot nem tartalmazó eljárások végrehajtása sokkal hatékonyabb mint azoké, amelyek tartalmaznak választási pontot, lásd alább (4.2).

4.1.1. A vágó beépített eljárás

A vágó beépített eljárás neve a `!`. Mindig sikeresen fut le. Mellékhatásként a Prolog keresési fa egyes ágait levágja: a végrehajtás adott állapotától visszafelé egészen a szülő célig, azt is beleértve, minden választási pontot megszüntet.

Egy cél szülője az a cél, amelyik az őt tartalmazó klóz fejével illesztődött. Például a

```
p:-q, r.  q:- s, t, u.
```

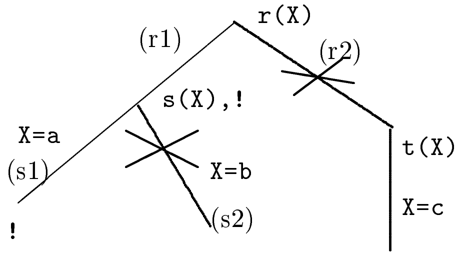
programban az `u` cél szülője a `q` cél a `p` klózban.

Tekintsük egy egyszerű példát! Ebben a `q` és az `r` eljárások között csak az a különbség, hogy az `r` első klózának végén szerepel egy vágó:

```
q(X):- s(X).     % (q1)
q(X):- t(X).     % (q2)

r(X):- s(X), !.  % (r1)
r(X):- t(X).     % (r2)

s(a).            % (s1)
s(b).            % (s2)
```

```
t(c).
```

A `| ?- q(X).` cél meghívásakor három megoldás áll elő az `X` értékeként, `a`, `b` és `c`. Ezzel szemben az `| ?- r(X).` célnak csak egy megoldása van: `X = a`. Az alábbi ábra mutatja az `r(X)` hívás végrehajtását:



Mint az ábra mutatja, `r(X)` meghívásakor létrejön egy kétágú választási pont, az (r1), (r2) klózoknak megfelelően. Az első redukció az (r1) klózzal történik, eredménye az `s(X), !` célsorozat. Ennek első tagja ismét egy kétágú választási pontot hoz létre, az (s1), (s2) klózoknak megfelelően. Megint az első, (s1) ágon megy tovább a Prolog program végrehajtása, és az `X = a` behelyettesítéssel járó redukció után az egytagú `!` célsorozat keletkezik. A vágó most következő végrehajtásakor először meg kell keresnünk a szülő célt, azaz azt a célt, amely a vágót tartalmazó klóz fejével illesztődött. Esetünkben ez az `r(X)` cél, amely a keresési fa gyökerében található. A vágó sikeresen lefut és mellékhatásként megszünteti a választási pontokat a szülő célig visszamenőleg, azt is beleértve. Példánkban ez két választási pontot jelent: megszüntetjük a vágót megelőző `s(X)` hívás (s2) választását és az eredeti `r(X)` hívásnak az (r2) választását. A fenti ábrán a keresési fának a vágó által levágott részeit vastag vonalakkal, magát a vágást pedig áthúzással jelöltük.

4.1.2. A vágások fajtái

Mint a fenti példából is látszik, a vágó kétféle értelemben módosíthatja a keresési teret:

Első megoldásra való megszorítás A vágó egyrészt kijelenti, hogy a vágó klózában az őt megelőző célsorozatnak csak az első megoldása érdekel minket, hiszen ebben a célsorozatban fennmaradt választási pontokat megszünteti; a példában ilyen az (s2) választás.

Elkötelezettség az adott klóz mellett A vágó másrészt pedig „elkötelezi” magát egy adott klóz-választás mellett, hiszen letiltja az őt követő klózok választását; a példában az (r2) választást.

Az első fajtájú vágót használjuk pl. a következő eljárás-definícióban: (az `útvonal(N, A, B, Hossz)` eljárást a 3.1.4 fejezetben definiáltuk.)

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

A fenti eljárásban tehát minden argumentum bemenő, azaz annak eldöntésére kívánjuk használni, hogy *adott* városok között van-e megfelelő út, pl.

```
| ?- van_elég_hosszú_út(2, 'Budapest', 'Párizs', 1000).
yes
```

Ilyen esetben fontos a vágó jelenléte, mert biztosítja azt, hogy erre az eldöntő kérdésre ne kapjunk többszörös igen választ. Ha ugyanis elhagyjuk a vágót, akkor visszalépéskor az `van_elég_hosszú_út` eljárás annyiszor sikerül, ahány megfelelő út van az adott két város között.

Nézzünk most egy listakezelő eljárást, amelyben szintén első fajtájú vágót használunk!

P1 Példa: Első elem ismétlődési száma

```
% Az L nem-üres lista első eleme H-szor ismétlődik a lista kezdőszeleteként.
kezdethossz(L, H) :-
        L = [E|_],
        append(Ek, Farok, L),
        \+ Farok = [E|_], !,
        /* egyformák(Ek, E), */
        length(Ek, H).

/*
% egyformák(Ek, E): Az Ek lista minden eleme E.
egyformák([], _).
egyformák([E|Ek], E) :-
        egyformák(Ek, E).
*/

| ?- kezdethossz([1,1,1,2,3,5], H).
H = 3 ? ;
no
| ?- kezdethossz([1,1,2,1,3,5], H).
H = 2 ? ;
no
| ?- kezdethossz([1,1,1,1], H).
H = 4 ? ;
no
```

Először értelmezzük deklaratív módon a `kezdethossz` eljárást úgy, hogy a vágót és az `egyformák` hívás körüli megjegyzés-jeleket elhagyjuk! Az első részcélban kikötjük, hogy az `L` lista ne legyen üres, és első elemét elnevezzük `E`-nek. Ezután a listát két részre — `Ek` és `Farok` — bontjuk, majd kikötjük, hogy `Farok` ne legyen egy `E`-vel kezdődő lista, az `Ek` lista viszont csupa `E`-ből álljon (vegyük észre, hogy ez első kikötés a `Farok = []` esetet is megengedi!). Így tehát `Ek` az a leghosszabb kezdőszelet lesz, amely csupa `E`-ből áll. A `kezdethossz` utolsó hívásában a `length` beépített eljárással megállapítjuk ennek a kezdőszeletnek a hosszát, ami a keresett ismétlődési szám.

Nézzük most, hogy hogyan hajtódik végre a `kezdethossz` eljárás, a vágót is figyelembe véve! Az `append` rendre 0, 1, 2, ... hosszú kezdőszeleteket ad vissza az `Ek` változóban. Amikor először teljesül az `append` hívás utáni feltétel, tehát az, hogy a `Farok` nem egy `E`-vel kezdődő lista, akkor biztos hogy az `Ek` egy csupa `E` elemből álló lista, tehát az `egyformák` feltétel ekkor biztosan teljesül. Az `append` ezutáni `Ek` megoldásaira (ha vannak) viszont nem teljesülhet ez a feltétel, hiszen ezekben már benne van első `E`-től különböző elem. Tehát érdemes a `\+ Farok = [E|_]` feltétel sikeressége esetén letiltani az `append` többi megoldását a vágóval. Az `egyformák` feltétel most már feleslegessé vált, tehát elhagyható. Az olvashatóság érdekében azonban érdemes legalább megjegyzésben felhívni a figyelmet erre a feltételre.

A fenti `kezdethossz` tipikusan egy *gyorsprogramozással* előállított eljárás. A kód tömör és gyorsan előállítható, viszont nem a leghatékonyabb, hiszen például kétszer kell végigmennie a kezdőszeleten (mind az `append` mind a `length` eljárások a kezdőszelet hosszával arányos ideig futnak). A fejezetben később (4.5 végén) mutatunk egy hatékonyabb rekurzív megoldást.

Az első megoldásra való megszorítás fontos szerepet kap a végtelen választási pontok kiküszöbölésében is, lásd alább a `memberchk` eljárást (4.1.3).

A vágó másik, egy adott klóz melletti elkötelezettséget jelző fajtájára példaként írjuk meg az abszolút-érték kiszámítását végző Prolog eljárást!

P2 Példa: Abszolút érték

```
% abs(X, A): A az X szám abszolút értéke.
abs(X, A) :- X < 0, !, A is -X.
abs(X, X).
```

Itt az első klóz az `X < 0` vizsgálattal kezdődik. Ha ez nem teljesül, akkor a második klózra térünk át. Ha viszont ez a feltétel igaz, akkor a vágó lefut, és kizárja a második klózt. Tehát a második klózra akkor és csak akkor kerülhet a vezérlés, ha `X < 0` nem teljesül, azaz `X >= 0`. Ezért a fenti eljárás logikailag azonos az alábbi változattal:

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

A második megoldás logikailag tiszta, és könnyebben is megérthető, mint az első, de kevésbé hatékony, mivel negatív számok esetén egy felesleges választási pontot hagy maga után. Kompromisszumként javasoljuk, hogy az ilyen vörös vágók esetén legalább megjegyzésként írjuk be a levágandó klózokba a megfelelő feltételt:

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

A példa általánosításaként elmondhatjuk, hogy egy

```
p :- felt, !, akkor.
p :- /* nem_felt, */egyébként.
```

alakú eljárás, ahol `felt` determinisztikus (tehát legfeljebb egy megoldása van), egy feltételes szerkezet. A `p` egy hívásának végrehajtásakor az `akkor` célsorozattal folytatjuk, ha `felt` teljesül, és `egyébként`-tel, ha nem. A feltételes szerkezet logikai jelentését úgy kapjuk, hogy a vágót és a második klózbeli megjegyzés-határoló jeleket elhagyjuk:

```
p :- felt, akkor.
p :- nem_felt, egyébként.
```

Itt `nem_felt` a `felt` negáltja, pl. ha az egyik `X >= 0`, akkor a másik `X < 0`.

A vágót használó feltételes szerkezetet diszjunkcióhoz hasonló formában is felírhatjuk:

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

Például az abszolút érték kiszámítására szolgáló eljárás a következő alakban is felírható:

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

Előfordulhat, hogy egyszerre kettőnél több esetet kell szétválasztanunk. Ilyenkor is használható a fenti séma. Egy szám előjelének kiszámításához háromfelé kell ágazni:

```
sign(X, Y) :- X < 0, !, Y = -1.
sign(X, Y) :- X > 0, !, Y =  1.
sign(_, 0).
```

Az ilyen alakú eljárások is felírhatók vágó nélküli feltételes szerkezetként:

```
sign(X, Y) :-
        (   X < 0 -> Y = -1
        ;   X > 0 -> Y =  1
        ;   Y = 0
        ).
```

Vegül néhány általános tanács a vágó használatához:

- Ha egy predikátumról tudjuk, hogy csak egyféleképpen sikerülhet, akkor abban helyezzük el a vágót, ne bízzunk abban, hogy majd valaki „felettünk” levágja a fölösleges ágakat.
- Egy klózban mindig pontosan arra a helyre tegyük a vágót, ahol eldőlt, hogy ez a helyes ág, tehát az utolsó olyan feltétel után, amelynek meghiúsulásakor még figyelembe akarjuk venni a többi klózt.

  Például ha az `abs/2` legutóbbi vágót tartalmazó definíciójában a vágót a klóz végére tesszük, akkor `abs(-5, -5)` sikerül.
- Kezdetben minden klózt önmagában értelmes szabályként írjunk fel, ami a fejével illeszthető célok igazságát definiálja. Csak ezután használjuk a vágót a fölösleges ágak eliminálására.
- A vágó leggyakrabban közvetlenül a fej vagy egy egyszerű vizsgálat után következik a törzsben.
- Elkötelezettséget jelző vágó használatára soha nincs szükség egy eljárás utolsó klózában.

4.1.3. Példák a vágó használatára

P3 Példa: Két szám maximumának számítása

Írjunk egy olyan Prolog eljárást, amely két szám maximumát számítja ki! Az első változat tiszta Prolog-ban íródott, nincs benne vágó:

```
% max(+X, +Y, ?Z): X és Y maximuma Z.
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- Y > X.
```

Ennek a változatnak hátránya, hogy az `X >= Y` esetben egy választási pontot hagy maga után. Ezért célszerű egy vágót elhelyezni az első klóz végén:

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

Ez egy zöld vágó, hiszen csak akkor jutunk el a vágóig, ha `X >= Y` fennáll, és ilyenkor a második klóz feltétele biztosan nem teljesül. Ebben a változatban viszont felmerülhet, hogy felesleges a második klózbeli feltétel, hiszen az az első klóz törzsében levő feltétel negáltja. Így jutunk a következő változathoz:

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

Ez a változat működőképes, de csak azzal a feltétellel, hogy a maximum *kiszámítására* használjuk csak, azaz a harmadik argumentumában változóval hívjuk meg. Hibás eredményt kapunk viszont, ha a maximum *ellenőrzésére* használjuk, tehát például az „igaz-e hogy 10 és 1 maximuma 1?” kérdést tesszük fel:

```
| ?- max(10, 1, 1).
yes
```

Ez a hívás nem tud az első klózzal illesztődni, mert a fejillesztés nem sikerül, viszont a másodikkal illeszthető, és ezért sikeresen fut le. Az első klózt átírva jobban látszik a probléma oka:

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

Tehát nemcsak a `X >= Y` feltétel hamis volta esetén, hanem a `Z = X` meghiúsulásakor is a második klózzal folytatjuk a futást. Ezt a hibát úgy tudjuk kijavítani, hogy a `Z = X` feltételt a vágó után helyezzük el:

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

Általános elvként elmondhatjuk, hogy a **kimenő argumentumok értékadását mindig a vágó után végezzük el!**. Ezzel nem csak azt érjük el, hogy az eljárás általánosabban alkalmazhatóvá válik, de hatékonyabb lesz a végrehajtása is. A fenti `(*)`-gal jelzett változatban ugyanis a fordítóprogram a `Z = X` hívásból egy általános egyesítést kell generáljon, ami miatt ténylegesen létre kell hozzon egy választási pontot a `max` hívás elején. A végső megoldásként javasolt `(**)` változatban a két klóz közötti választás csak az `X >= Y` aritmetikai összehasonlításon múlik, amit az igényesebb Prolog fordítóprogramok egy hagyományos `if-then-else` szerkezetként, választási pont létrehozása nélkül fordítanak.

P4 Példa: Listaelemek ellenőrzése — a `memberchk` eljárás

Tekintsük a korábban ismertetett `member` eljárásnak egy olyan változatát, amelyben egy vágó segítségével csak az első megoldásra szorítkozunk.

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

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

Ez a `memberchk/2` eljárás SICStus Prologban a `lists` könyvtárban megtalálható.

A `memberchk` eljárást célszerű használni a `member` helyett akkor, ha eldöntő kérdésként egy adott elem adott listában való jelenlétét vizsgáljuk. Például az alábbi hívás sikeresen fut le, de egy választási pontot hagy maga után:

```
member(1, [1,2,3,4,5,6,7,8,9])
```

Ezután visszalépéskor a lista maradék nyolc elemét is végignézi, hogy nem talál-e közöttük egy `1` értéket, hiszen a `member` logikája szerint egy ilyen kérdésnek annyiszor kell sikerülnie, ahányszor megtalálható a listában az adott elem. Általában viszont nem erre a logikára van szükség. Ha egy determinisztikus igen-nem válaszra van szükségünk, használjuk a `memberchk` eljárást:

```
memberchk(1, [1,2,3,4,5,6,7,8,9])
```

A `memberchk` egy érdekesebb alkalmazása lehet az, hogy nyílt végű listák elemévé tud tenni megadott Prolog kifejezéseket:

```
| ?- memberchk(1, L), memberchk(2, L), memberchk(1, L).
L = [1,2|_A] ? ;
no
```

Emlékezzünk vissza, hogy korábban ezt a célsorozatot a `member` eljárással próbáltuk ki, akkor is ezt az (első) eredményt kaptuk, viszont több végtelen választási pont jött létre. Ezt kerüli el a `memberchk`-be helyezett vágó, amely így tehát egy adott elemet egy nyílt végű lista (lehető legbaloldalibb) elemévé tesz, úgy hogy az ismétlődő elemek csak egyszer kerülnek a listába.

A `memberchk` eljárásnak ezt a tulajdonságát használja ki az alábbi rövid program, amely Prolog kifejezéseket olvas be és egy nyílt végű listával megvalósított `Szótár` változó elemévé teszi ezeket.

```
szótáraz(Szótár):-
      read(M-A), !,
      memberchk(M-A,Szótár),
      write(M-A), nl,
      szótáraz(Szótár).
szótáraz(_).
```

Itt `read` és `write` általános Prolog kifejezések beolvasására ill. kiírására szolgáló beépített eljárások, `nl` sort emel. Íme a fenti program egy futása:

```
| ?- szótáraz(Szótár).
|: alma-apple.
alma-apple
|: körte-pear.
körte-pear
|: alma-_.
alma-apple
|: _-pear.
körte-pear
|: vege.

Szótár = [alma-apple,körte-pear|_A] ?
```

Tehát szótárunkba például `Magyar-Angol` szópárokat írhatunk be, ezek a `memberchk` eljárás segítségével a `Szótár` nyílt végű lista elemeivé lesznek. Ha egy `Magyar-_` vagy egy `_-Angol` kifejezést írunk be, akkor ugyanaz a `memberchk` hívás kikeresi az első olyan elemet (szópárt), amely a megadott mintával illeszthető, és azt kiírja; ezzel megvalósítva a szótárból való kikeresést. Ha egy olyan Prolog kifejezést írunk be, amely nem `X-Y` szerkezetű, akkor a `read` eljárás meghiúsul és a `szótáraz` hívás a második klóz segítségével sikeresen véget ér. A futás végén a hívás paraméterében megkapjuk szótárunk végső álapotát.

4.2. Determinizmus és indexelés

A Prolog nyelv egyik alapvető vezérlési szerkezete a visszalépéses keresés. Azonban sok olyan Prolog eljárás van, amelynek bizonyos hívásai csak egyféleképpen sikerülhetnek. Az ilyen hívásokat **determinisztikus**, a többféleképpen sikerülőket pedig **nem-determinisztikus** hívásoknak nevezzük. A determinisztikus eljáráshívások Prolog megvalósítása hasonló lehet a hagyományos eljárás-szervezéshez, amely sokkal hatékonyabb mint a Prolog visszalépést is lehetővé tevő eljárás-szervezése. Ezért különösen fontos, hogy a rendszer meg tudja különböztetni determinisztikus és a nem-determinisztikus eljáráshívásokat.

4.2.1. Indexelés

A determinizmus felismerése érdekében a legtöbb Prolog fordítóprogram alkalmazza az ún. *indexelés* módszerét. Ez azt jelenti, hogy amikor egy hívást elkezdünk végrehajtani, először az őt definiáló eljárás klózai közül valamilyen egyszerű ismérv szerint kiszűrjük azokat, amelyek biztosan nem lesznek vele illeszthetők. A legtöbb Prolog rendszer, így a SICStus Prolog is, az első argumentum szerint indexel: ha a hívásban az első argumentum változó, akkor az eljárás mindegyik klózával próbál illeszteni, ha viszont nem változó, akkor a klózoknak csak a megfelelő rész-sorozatával illeszt. A rész-sorozatok kialakításakor az első argumentum legkülső funktorát vesszük figyelembe, azaz az első argumentumpozíción azonos konstans-értékeket, illetve az azonos nevű és argumentumszámú struktúrákat tartalmazó klózokat rakjuk egy csoportba (az első helyen változót tartalmazó klózok mindegyik csoportba belekerülnek). Példaként tekintsük a következő eljárásdefiníciót:

```
p(0, a).        % (1)
```

```
p(s(0), b).     % (2)
p(s(1), c).     % (3)
p(2, d).        % (4)
p(3, e).        % (5)
```

Itt a `p(0, X)` hívás esetén az indexelés csak egyetlen klózt, az (1)-t választja ki. A `p(s(0), X)` hívás esetén viszont a (2) és (3) klózok választódnak ki, mivel az indexelés a struktúrák argumentumaiban levő értékeket már nem veszi figyelembe. Ez azt jelenti, hogy a `p(0,X)` hívás nem hoz létre választási pontot, a `p(s(1),X)` létrehoz ugyan egyet, de még az adott híváson belül meg is szünteti, míg a `p(s(0),X)` hívás után meg is marad a létrehozott választási pont, annak ellenére, hogy mindhárom hívás determinisztikus. Ugyanúgy determinisztikus a `p(Y, a)` hívás is, de mivel a második argumentumra nem terjed ki az indexelés, ez a hívás a legtöbb Prolog rendszerben választási pont létrehozásával jár.

Az struktúra-argumentumokat is bevonhatjuk az indexelésbe segédeljárások segítségével. A fenti példában ehhez a (2) és (3) klózt cseréljük le a következő klózra:

```
p(s(A), B) :- pp(A, B).
```

Továbbá a `pp` eljárást definiáljuk a következőképpen:

```
pp(0, b).
pp(1, c).
```

Vágó alkalmazásával is megszüntethetjük a felesleges választási pontokat:

```
p1(A, B) :- p(A, B), !.
```

Ha a `p1` eljárást mindig úgy használjuk, hogy vagy az első, vagy a második argumentuma behelyettesített, akkor ez a vágó zöld lesz, hiszen a `p` egy-egyértelmű hozzárendelést valósít meg.

4.2.2. Listakezelő eljárások indexelése

Számos listafeldolgozó eljárás két klózból áll: egy üres listára és egy nem-üres listára vonatkozó klózból. Példaként idézzük fel az `append` eljárást:

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

Az első argumentum szerinti indexelés megkülönbözteti a két klózt, tehát a listák összefűzését végző hívások (pl. az `| ?- append([1,2], [3,4], L).`) választási pont létrehozása nélkül futnak le. Ezzel szemben az `| ?- append(L1, L2, []).` hívás választási pontot hagy maga után, pedig a második klózzal nem illeszthető (ezt nem tekintjük túl nagy problémának, mivel az `append` szétszedő üzemmódja az általános esetben úgyis nem-determinisztikus).

Nézzünk egy másik listakezelő eljárást:

P5 Példa: Lista utolsó eleme

```
% last(L, E): Az L lista utolsó eleme E.
% SICStus Prologban a lists könyvtárban megtalálható
last([E], E).
last([_|L], E) :-
        last(L, E).
```

Ez a természetes megfogalmazása a „lista utolsó eleme” relációnak, de nem a leghatékonyabb. Az szokásos indexelés nem különbözteti meg ugyanis a két klózt, hiszen mindkettőnek az első argumentuma nem-üres lista (azaz a fő funktor a `'.'/2`). A felesleges választási pontot egy zöld vágóval szüntethetjük meg:

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

Ennek a definíciónak az első klózában a második, „kimenő” argumentumnak a vágó meghívása előtt adunk értéket. Ebben az esetben ez nem jár a `max/3` eljárás kapcsán megismert problémákkal, mert a második klózból nem „szedtük ki” az ellenőrzést: a rekurzív hívás nem sikerülhet, ha `L = []`. Mivel itt az indexelés az első argumentumok azonos funktora miatt amúgy sem működik, a „korai” értékadás még hatékonyságveszteséggel sem jár. Ezt a kérdést részletesebben tárgyalja a 4.2.4 szakasz.

Igazán hatékony megoldást egy segédeljárás bevezetésével nyerhetünk, ebben az esetben ugyanis az indexelés miatt egyáltalán nem is jön létre választási pont.

```
% last(L, E): Az L lista utolsó eleme E.
last([X|L], E) :-
        last1(L, X, E).

% last1(L, X, E): Az [X|L] lista utolsó eleme E.
last1([], E, E).
last1([X|L], _, E) :-
        last1(L, X, E).
```

Fontos a `last1` eljárás argumentumainak sorrendje, ez talán egy kicsit furcsa, de a lista farkát az indexelés miatt az első argumentum-pozícióba kell tenni, míg a lista feje a második argumentum lesz.

4.2.3. Aritmetikai eljárások indexelése

A legtöbb Prolog rendszer nem képes felismerni azt, hogy két klóz kizárja egymást, ha ez aritmetikai összehasonlítások miatt van így. Például az alábbi faktoriális eljárás:

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

a vágó nélkül felesleges választási pontot hozna létre. Aritmetikai elágaztatások esetén tehát mindig használjunk vágót.

4.2.4. A vágó és az indexelés kölcsönhatása

Figyeljük meg azt is, hogy az első `fakt` klózban a második argumentum értékadását a vágó után végeztük el. Miután itt a vágó zöld, az egyszerűbb,

```
fakt(0, 1) :- !.                                        % (**)
```

alak nem vezetne olyan hamis eredményhez mint a korábbi `max` esetén. Mégis érdemes a kimenő argumentum értékadását a vágó utánra tenni, mert így a fordítóprogram indexelő algoritmusa felismeri hogy így egy egyszerű `if N = 0 then ... else ...` szerkezetről van szó, és ennek megfelelő elágaztató kódot generál.

Az indexelés csak akkor tudja figyelembe venni a vágó jelenlétét, ha garantált, hogy az adott klóz kiválasztása csak az első argumentum legkülső funktorán múlik. Ehhez az kell, hogy a vágó a törzs elején álljon, és az első kívételével minden fejbeli argumentum különböző változó legyen (ha az első argumentum struktúra, akkor ennek argumentumait is beleértve).

Az előbbi faktoriális program `(*)` klóza megfelel ennek a feltételnek. Ha a rövidebb `(**)` alakot tekintjük, ebben a fejillesztés meghiúsulhat attól is, hogy a hívás 2. argumentuma nem változó, de nem is az 1 érték.

Nézzünk erre még egy példát: legyen feladatunk egy olyan (nem igazán értelmes) számfüggvény megírása Prologban, amely a 0 és 1 számokhoz az 1 értéket, és minden más számhoz a 2 értéket rendeli. Ennek leghatékonyabb módja:

```
p(0, Y) :- !, Y = 1.
p(1, Y) :- !, Y = 1.
p(_, 2).
```

Miután a kimenő argumentum csak a vágó után kapja meg értékét, az indexelés mechanizmus ki tudja következtetni, hogy a harmadik klóz a 0 és 1 esetben mindenképpen kizáratik, és ennek megfelelően egy imperatív nyelvű feltételes kifejezéshez hasonló kódot tud generálni.

4.2.5. A vágó és az indexelés hatékonysága

Vizsgáljuk a vágó és az indexelés hatékonyságát egy Fibonacci-szerű sorozat kiszámítását végző programon. A program a következő képlettel definiált sorozat n-edik tagját számítja ki.

$$f_1 = 1; \quad f_2 = 2; \quad f_n = f_{\lfloor 3n/4 \rfloor} + f_{\lfloor 2n/3 \rfloor}, \quad n > 2$$

A fenti rekurzív képletet használó három programváltozatot mutatunk be alább egymás mellett. A `fib` változat nem használ vágót, míg a `fibc` és a `fibci` igen. Az utóbbi két változat között annyi a különbség, hogy a `fibc` esetében létrejönnek választási pontok, míg a `fibci` eljárásnál nem.

```
fib(1, 1).
fib(2, 2).
fib(N, F) :-
        N > 2,
        N2 is N*3//4,
        N3 is N*2//3,
        fib(N2, F2),
        fib(N3, F3),
        F is F2+F3.
```

```
fibc(1, 1) :- !.
fibc(2, 2) :- !.
fibc(N, F) :-
        N > 2,
        N2 is N*3//4,
        N3 is N*2//3,
        fibc(N2, F2),
        fibc(N3, F3),
        F is F2+F3.
```

```
fibci(1, F) :- !, F = 1.
fibci(2, F) :- !, F = 2.
fibci(N, F) :-
        N > 2,
        N2 is N*3//4,
        N3 is N*2//3,
        fibci(N2, F2),
        fibci(N3, F3),
        F is F2+F3.
```

Hasonlítsuk össze a három program futását az `N = 1600` bemenő érték mellett:

	`fib`	`fibc`	`fibci`
futási idő	4410 ms	4060 ms	3820 ms
meghiúsulási idő	730 ms	0 ms	0 ms
összesen	5140 ms	4060 ms	3820 ms

A „futási idő” sorban a megoldás előállításához szükséges idő, a „meghiúsulási idő” sorban pedig a sikeres híváson való visszalépés ideje szerepel. Mint látjuk, a vágót használó változatokban az utóbbi idő 0-ra csökkent, hiszen nem kell a `fib` harmadik klózának alkalmazhatatlanságáról meggyőződni. Emellett a `fibc` változat esetében közel 10%-nyi nyereséget jelent a tényleges futási időben az, hogy a választási pontokat a vágók szinte azonnal megszüntetik. A `fibci` változatban ezen felül további 5%-ot jelent az, hogy az indexelés miatt a választási pontok létre sem jönnek.

4.3. Jobbrekurzió és akkumulátorok

Deklaratív nyelvekben a ciklus, mint vezérlési szerkezet helyét a rekurzió veszi át. Az általános rekurzió sokkal költségesebb mint a ciklus. Van azonban a rekurzív szerkezeteknek egy olyan speciális esete, amelynek bonyolultsága lényegében megegyezik a ciklusével, ez az ún. *jobbrekurzió*, vagy farok-rekurzió (tail recursion).

4.3.1. Jobbrekurzió

Prologban jobbrekurzióról akkor beszélünk, ha egy eljárás törzsében utolsóként szerepel egy rekurzív hívás. Ilyenkor a Prolog implementáció megkísérli kiküszöbölni a rekurziót: a paraméterátadás elvégzése után felszabadítja az adott eljárás által által lefoglalt helyet és „visszaugrik” az eljárás elejére. Ezt azonban csak akkor tudja megtenni, ha az eljárásnak a jobbrekurzív hívás előtti része nem tartalmaz választási pontot. Ha ugyanis van választási pont az eljárásban, akkor nem szabadítható fel az eljárás által lefoglalt hely, hiszen arra visszalépéskor szükség lehet. Ezért rendkívül fontos, hogy a determinisztikus eljáráshívások a Prolog rendszer számára is felismerhetők legyenek (az indexelés, ill. a felhasználó által elhelyezett vágók segítségével).

A Prolog megvalósítások tulajdonképpen egy a jobbrekurziónál általánosabb módszer alkalmaznak, az utolsó hívás optimalizálást (last call optimisation). Ez akkor is működik, ha az utolsó hívás nem magának az eljárásnak a rekurzív visszahívása. Így (esetleg több lépésben) kölcsönösen rekurzív eljáráspárok esetén is működik ez a fajta optimalizálás.

Most néhány példán mutatjuk be, hogyan érhetjük el, hogy eljárásaink jobbrekurzívak legyenek.

P6 Példa: Számlisták összegzése

Tekintsük az alábbi egyszerű feladatot: adott számlista összegét kell előállítani. Első megoldásunk:

```
% sum(+L, ?S): Az L számlista elemeinek összege S.
sum([], 0).
sum([X|L], S):-
      sum(L,S0), S is S0+X.
```

Ahhoz, hogy ezt jobbrekurzív alakra hozzuk, egy háromargumentumú segédeljárást kell definiálni `sum(L, S0, S)` feladata az, hogy az `S0` adott számértékhez hozzádja `L` összes elemét, és az eredményt adja ki `S`-ben.

```
% sum_list(+L, ?S): Az L számlista elemeinek összege S.
% (sum jobbrekurzív változata)
% SICStus Prologban a lists könyvtárban megtalálható
sum_list(L, S):-
      sum(L, 0, S).

% sum(+L, +S0, ?S): Az L számlista elemeinek összege S-S0.
sum([], S, S).
sum([X|L], S0, S):-
      S1 is S0+X, sum(L, S1, S).
```

Vegyük észre, hogy a `sum/3` eljárás fejkommentjében az `L`, `S0` és `S` közötti összefüggést írtuk le, és nem a korábbi meghatározást (`S0`-hoz hozzádva `L` elemeit kapjuk `S`-t), amely egy kicsit imperatívabb volt a szükségesnél.

4.3.2. Akkumulátorok

A `sum(L, S0, S)` eljárás második és harmadik argumentuma ugyanahhoz a mennyiséghez kapcsolódik, mindkettő valahány listaelem összegét tartalmazza: `S0` egy részösszeget, míg `S` a végső összeget. Egy ilyen argumentum-párt **akkumulátornak** vagy **gyűjtőargumentum-párnak** nevezünk. Egy gyűjtőargumentum természetesen nemcsak számokat tárolhat, hanem tetszőleges Prolog kifejezéseket. A lényeg az, hogy a pár első tagja egy ténylegesen változó mennyiség belépéskori állapotát jelenti, míg a második az adott eljárás szempontjából végső állapotot tartalmazza.

Az akkumulátor-változók jelölésére azt a konvenciót alkalmazzuk, hogy a fejben az akkumulátor-párt mindig *Vált*`0`, *Vált* formában írjuk, ahol *Vált* egy tetszőleges változónév, pl. `S`. A törzsben az első olyan hívásban, amely változtatja az adott állapotot, a *Vált*`0`, *Vált*`1` pár szerepel, a másodikban a *Vált*`1`, *Vált*`2` pár stb., míg az utolsóban a *Vált*`N`, *Vált* pár. Például tekintsük a következő eljárást:

```
% sum_3_lists(+L, +LL, +LLL, +S0, ?S): Az L, LL, LLL számlisták
% összegeinek összege S-S0
```

```
sum_3_lists(L, LL, LLL, S0, S) :-
      sum(L, S0, S1),
      sum(LL, S1, S2),
      sum(LLL, S2, S).
```

Természetesen egyszerre több akkumulátor-párt is használhatunk egyetlen eljárásban. Példaként tekintsünk egy olyan eljárást, amely egy adott számlista összegét és négyzetösszegét is előállítja!

```
% sum12(L, S0, S, Q0, Q): S = S0+ΣL, Q = Q0+ΣL^2
sum12([], S, S, Q, Q).
sum12([X|L], S0, S, Q0, Q):-
      S1 is S0+X, Q1 is Q0+X*X,
      sum12(L, S1, S, Q1, Q).
```

4.3.3. Listák gyűjtése

A számértékek mellett a lista-értékek is gyakran szerepelnek akkumulátorként. A listafordításban használt `revapp` is lista-akkumulálást végez. Idézzük fel ezt az eljárást!

```
% revapp(Xs, L0, L): Xs megfordítását L0 elé fűzve kapjuk L-t;
% másképpen:  Xs megfordítása L-L0.
revapp([], L, L).
revapp([X|Xs], L0, L) :-
      L1 = [X|L0],
      revapp(Xs, L1, L).
```

Figyeljük meg a fejkomment második változatát: az ebben levő `L-L0` kifejezés alatt egy olyan listát értünk, amelyet `L0` elé fűzve `L`-et kapjuk; a kifejezés csak akkor értelmes, ha van ilyen lista (tehát `L0` az `L`-nek egy záró szelete). Azért vezetjük be ezt a néha „különbség-listának” is nevezett formulát, mert ez gyakran megkönnyíti a listák akkumulálását végző eljárások jelentésének megfogalmazását.

Nézzük most meg, hogy ez a `revapp` eljárás miben különbözik a 35. oldalon ismertetett változattól! Egyrészt átírtuk a változóneveket, hogy az akkumulátor jelleg nyilvánvalóbbá váljék, másrészt bevezettünk egy `L1` segédváltozót, hogy az akkumulálási lépést jobban meg tudjuk mutatni.

A már sok szempontból vizsgált `append` eljárás is tulajdonképpen akkumulál:

```
% append(Xs, L, L0):  Xs = L0-L
append([], L, L).
append([X|Xs], L, L0) :-
      L0 = [X|L1],
      append(Xs, L, L1).
```

Az `append`, mint akkumuláló eljárás két szempontból is szokatlan. Egyrészt egy formai különbséget látunk: az állapotváltozó régi és új értéke fel van cserélve, tehát a régi érték van az utolsó, harmadik argumentumban, míg az újabb érték a második pozíción szerepel. Másrészt, az `append`-et összefűző módban tekintve, maga az akkumulálandó mennyiség nem egy konkrét lista, hanem egy változó, amelybe a két lista összefűzöttje kerül. Egy akkumulálási lépésben, amit itt is a törzs elején levő egyenlőség jelent, ezt a változót töltjük fel egy lista-struktúrával, amelynek a farka lesz az új változó, ahová a rekurzív hívásnak az eredménye kerül. A leálló klózban helyettesítődik be végleg a farok-változónk a második argumentumban megadott listára.

Bár az `append` eljárásnak az akkumulálási sémába való „kényszerítése” esetleg egy kicsit erőltettnek tünhet, fontos megjegyezni, hogy a Prolog nyelvben két irányból is építhetjük, gyűjthetjük a listákat. Nézzünk most erre egy érdekes példát!

P7 Példa: $a^n b^n$ alakú sorozatok

Írjunk olyan Prolog eljárást, amely adott `N>=0`-ra felépíti azt a `2N` hosszúságú listát, amelynek első `N` eleme az `'a'`, hátsó `N` eleme pedig a `'b'` atom! Például az `| ?- anbn(2, L).` hívás eredménye `L = [a,a,b,b]` lesz.

Első megoldásunk működőképes, de nem hatékony:

```
% anbn(N, L): L = [a, ..., a, b, ..., b]
%                      N db       N db
anbn(N, L) :-
        an(N, a, AN),
        an(N, b, BN),
        append(AN, BN, L).

% an(N, A, L): L az A elemet N-szer tartalmazó lista
an(0, _A, L) :- !, L = [].
an(N, A, [A|L]) :-
        N > 0,
        N1 is N-1,
        an(N1, A, L).
```

Azért nem hatékony, mert miután külön felépítette az `a`-k és a `b`-k listáját még egyszer végig kell mennie az `a`-k listáján, hogy azokat az `append` segítségével a `b`-k listája elé fűzze. Az append hívást úgy kerülhetjük el, hogy az `an` eljárásban egy akkumulátor-párt használunk:

```
anbn(N, L) :-
        an(N, b, [], BN),
        an(N, a, BN, L).

% an(N, A, L0, L): L-L0 az A elemet N-szer tartalmazó lista
an(0, _A, L0, L) :- !, L = L0.
an(N, A, L0, L) :-
        N > 0,
        N1 is N-1,
        an(N1, A, [A|L0], L).
```

Vegyük észre, hogy míg az első megoldásban elölről építjük a listát (mint az `append`-ben), addig a másodikban hátulról (mint a `revapp`-ban). Miután csupa egyforma elemből kell listát építeni, az akkumulátoros megoldásban mindkét irány alkalmazható. (Az olvasóra bízzuk az utóbbi `an/4` eljárás egy olyan változatának megírását, amely az `append`-hez hasonlóan elölről építi a listát.)

A két irányból való listaépítés ötlete alapján készült az alábbi harmadik megoldás, az eddigiek közül a legtömörebb és a leghatékonyabb. Ez egyetlen ciklusban építi fel a keresett listát:

```
anbn(N, L) :-
        anbn(N, [], L).

% anbn(N, L0, L): L = [a, ..., a, b, ..., b |L0]
%                           N db       N db
anbn(0, L0, L) :- !, L = L0.
anbn(N, L0, [a|L]) :-
        N > 0,
        N1 is N-1,
        anbn(N1, [b|L0], L).
```

4.4. Algoritmusok Prologban

Gyakran előfordul, különösen olyan programozók esetén, akiknek nagy gyakorlata van imperatív programozási nyelvekben, hogy egy például C nyelven megfogalmazott algoritmust szeretnének átültetni Prologba. Erre mutatunk most két példát.

P8 Példa: Hatványozás

Első példánk egy hatékony hatványozási algoritmus, amely az alap 2^i kitevőjű hatványainak szorzataként állít elő egy megadott hatványt. Íme az algoritmust megvalósító C nyelvű függvény:

```
/* hatv(a, h) = a**h */
int hatv(int a, unsigned h)
{
  int e = 1;
  while (h > 0)
  {
    if (h & 1) e *= a;
    h >>= 1;
    a *= a;
  }
  return e;
}
```

A kétargumentumú C függvénynek nyilvánvalóan egy háromargumentumú Prolog eljárás felel majd meg, ahol a harmadik, kimenő argumentumban jelenik meg a függvény eredménye. A C függvényben levő ciklusból egy Prolog segédeljárást kell készíteni. Minden egyes C változónak a segédeljárás egy vagy két argumentuma felel majd meg. Azok a változók, amelyek csak „bemenő” értékei a ciklusnak, tehát a ciklus lefutása után nincs rájuk szükség — ilyenek az `a` és a `h` — egy-egy bemenő paraméterré válnak a segédeljárásban. Az `e` változó viszont, amelyre a ciklus után is szükség van — ilyen az `e` — egy gyűjtőargumentum-párrá változik. A gyűjtőargumentum kezdőértéke 1, végértéke pedig azonos az eredeti eljárás értékével:

```
% hatv(A, H, E): A**H = E.
hatv(A, H, E) :-
        hatv(H, A, 1, E).
```

Nézzük most a ciklusnak megfelelő segédeljárást! Megjegyzésként a Prolog kód mellé írtuk a megfelelő C kódot, némileg módosítva a könnyebb megfeleltethetőség érdekében.

```
% hatv(H, A, E0, E):
%        E0 * (A**H) = E.
                              % ism:
hatv(0, _, E0, E) :- !, E=E0. %   if (h == 0) return e;
hatv(H, A, E0, E) :-          %
        H > 0,                %
        (   H /\ 1 =:= 1      %   if (h & 1)
        ->  E1 is E0*A        %     e *= a;
        ;   E1 = E0           %
        ),                    %
        H1 is H >> 1,         %   h >>= 1;
        A1 is A*A,            %   a *= a;
        hatv(H1, A1, E1, E).  %   goto ism;
```

Amikor a C kód egy változó értékét megváltoztatja, akkor a Prolog kódban egy új változóba kell az új értéket beírni. Vigyázni kell arra, hogy a végrehajtás minden ágán ez az új Prolog változó megkapja a C változó pillanatnyi értékét. Erre példa a C `if` utasítása: ebből Prologban egy diszjunktív feltételes szerkezet lett, amely az `e` változó új értékét az `E1` Prolog változóban tárolja. Vigyázni kell tehát arra, hogy ha a feltétel nem teljesül, `E1` akkor is megkapja a megfelelő értéket a feltételes szerkezet „egyébként” ágán (`E1 = E0`).

A ciklus végén rekurzív módon vissza kell hívni az adott eljárást, minden argumentumban az adott C változó pillanatnyi értékét tároló Prolog változót írva. Példánkban a ciklusmag minden változó értékét egyszer változtatta meg, ezért minden változó az 1-es indexet viseli a visszahívásban, de ez természetesen nem mindig van így.

Végül nézzük a fenti Prolog kód legelejét, azaz a fejkommentet. A C kód logikája az, hogy az eredeti hatványozási feladat egy részét már elvégeztük, ezt az `e` változó értéke már tükrözi, de ugyanakkor az `a` és `h` változók értékét is módosítottuk, úgy hogy $\mathtt{a}^{\mathtt{h}}$-nal kell már csak az `e` értéket megszorozni, hogy a kívánt végeredményt megkapjuk. Pontosan ezt fejezi ki a Prolog kód fejkommentje: `E0 * (A**H) = E`.

A Prolog fejkomment azért is érdekes, mert ez nagyon közel áll ahhoz a *ciklus-invariáns* feltételhez, amivel egy C ciklus helyességét bizonyítani lehet. A ciklus-invariáns egy olyan logikai feltétel, amelyre bebizonyítható, hogy

1. a ciklusba való belépéskor következik az előfeltételekből, pl. a változók kezdőértékeiből;
2. ha (induktív módon) feltételezzük a fennállását egy ciklus elején, akkor egyszer lefuttatva a ciklusmagot az új változóértékekre is fennáll;
3. amikor a ciklusból kilépünk, akkkor belőle következik a ciklus utófeltétele.

Alább megismételjük a `hatv` függvény C kódját egy `assert` hívásban megadva a ciklus-invariánst, és kommentekben megadva a változók új értékét a ciklus lefutása után.

```
/* hatv(a, h) = a**h */
int hatv(int a0, unsigned h0)
{
  int e = 1, a = a0, h = h0;
  while (h > 0)
  {
    /* assert( a0**h0 == e * a**h)); */
    assert( abs(pow(a0,h0)-e*pow(a,h)) < 0.00001 );
    if (h & 1) e *= a;        /* e1 = e * (a ** (h&1)) */
    h >>= 1;                  /* h1 = (h-(h&1))/2 */
    a *= a;                   /* a1 = a*a */
  }
  return e;
```

GY3.

Bizonyítsuk be ciklus-invariáns segítségével a `hatv` függvény helyességét.

P9 Példa: Fibonacci sorozatok hatékony kiszámítása

Második példaként álljon itt a Fibonacci sorozat adott elemét kiszámoló hatékony C függvény:

```
/* fib(0) = 0; fib(1) = 1; fib(n) = fib(n-1)+fib(n-2), n > 1 */
unsigned fib(unsigned n)
{
  unsigned f0 = 0, f1 = 1, t;
  while (n > 0) t = f1, f1 += f0, f0 = t, --n;
  return f0;
}
```

Ennek a C függvénynek megfelelő Prolog eljárás pedig a következő:

```
fib(N, F) :-                % unsigned fib(unsigned N)
        fib(N, 0, 1, F).    % {
                            %   unsigned F0 = 0, F1 = 1, F2;
% fib(N, F0, F1, FN):       %
%    Az F0 és F1 kezdőértékű %
%    fib sorozat N. eleme FN. %
                            % ism:
fib(0, F0, _, F0).          %   if (N == 0) return F0;
fib(N, F0, F1, F) :-        %
        N > 0,              %
        N1 is N-1,          %   --N;
        F2 is F0+F1,        %   F2 = F0+F1;
        fib(N1, F1, F2, F). %   F0 = F1; F1 = F2; goto ism;
                            % }
```

4.5. Megoldások gyűjtése és felsorolása

Egy keresési feladatra alapvetően kétféle Prolog eljárást készíthetünk:

Gyűjtés Az eljárás a megoldásokat összegyűjti pl. egy listába.

Felsorolás Az eljárás a megoldásokat felsorolja, azaz először kiadja az első megtalált megoldást, majd visszalépés esetén adja a következőt, stb.

Ebben a fejezetben néhány példán keresztül bemutatjuk, hogyan hozhatók a kétféle fajtájú programok hasonló alakra, és hogyan származtathatók a felsoroló fajtájúak a gyűjtő fajtájúakból. Ezt azért fontos, mert míg a gyűjtő megoldási módot más programozási nyelvekből (pl. SML-ből) sokan ismerik, addig a felsoroló eljárások csak a logikai nyelvekben találhatók meg.

P10 Példa: Kettő hatványai

Legyen a feladat egy adott számnál nem nagyobb (természetes kitevős) kettőhatványok egy listába való összegyűjtése. Például ha a maximum 10, akkor a várt eredmény `[1,2,4,8]`.

```
% L azon H = 2**i alakú egészek listája, amelyekre 1 =< H =< Max.
khatvanyok(Max, L) :-
        khatvanyok(1, Max, L).

% L azon H = 2**i alakú egészek listája, amelyekre H0 =< H =< Max
% (ahol H0 maga is 2**j alakú).
khatvanyok(H0, Max, L) :-
        H0 =< Max, !,
        L = [H0|L1],
        H1 is 2*H0,
        khatvanyok(H1, Max, L1).
khatvanyok(_H0, _Max, []) /* :-
        _H0 > _Max */.
```

A megoldáshoz tehát egy segédeljárást használunk, amelynek első argumentumában a soron következő kettőhatvány szerepel. Ha ez nem nagyobb mint a `Max`, akkor az eredménylista első elemévé tesszük (`L = [H0|L1]`), előállítjuk a következő kettőhatványt és ezzel rekurzívan hívjuk a segédeljárást. Ha a soron következő kettőhatvány nagyobb mint `Max`, üres listát adunk eredményül.

Most vizsgáljuk meg ugyanennek a feladatnak a felsoroló megoldását!

```
% H = 2**i alakú egész, amelyre 1 =< H =< Max.
khatvany(Max, H) :-
        khatvany(1, Max, H).

% H = 2**i alakú egész, amelyre H0 =< H =< Max.
% (ahol H0 maga is 2**j alakú).
khatvany(H0, Max, H) :-
        H0 =< Max,
        (   H = H0
        ;   H1 is 2*H0, khatvany(H1, Max, H)
        ).
```

Itt is egy hasonló paraméterezésű segédeljárást használunk, de ez csak egy klózból áll. Ha a soron következő kettőhatvány (`H0`) már nagyobb `Max`-nál, akkor az eljárás meghiúsul, hiszen nincs több megoldás. Egyébként meg vagy a soron következő kettőhatványt adjuk eredményül, vagy az őt követő kettőhatványokat — ezt a két esetet fedi le a diszjunkció két ága.

Vegyük észre, hogy a felsoroló eljárásban, a gyűjtővel ellentétben, nincs „leálló” klóz, hiszen a felsorolás „végen” az eljárásnak meg kell hiúsulnia.

Bonyolítsuk egy kicsit a feladatot azzal, hogy csak 8-ra végződő kettőhatványokat keressük, és próbáljunk egy általános sémát adni a megoldásokra!

```
% L azon H = 2**i alakú, 8-as jegyre végződő egészek listája,
% amelyekre 1 =< H =< Max.
khatvanyok8(Max, L) :-
        khatvanyok8(1, Max, L).

% L azon H = 2**i alakú, 8-as jegyre végződő egészek listája,
% amelyekre H0 =< H =< Max (ahol H0 maga is 2**j alakú).
khatvanyok8(H0, Max, L) :-
        következő(H0, Max, E, H1), !,
        L = [E|L1],
        khatvanyok8(H1, Max, L1).
khatvanyok8(_, _, []).

% E a legkisebb olyan kettőhatvány, amelyre H0 =< E =< Max
% és amely 8-as jegyre végződik. H az E-t követő kettőhatvány.
következő(H0, Max, E, H) :-
        H0 =< Max, H1 is H0*2,
        (   H0 mod 10 =:= 8 -> E = H0, H = H1
        ;   következő(H1, Max, E, H)
        ).
```

Itt tehát a `következő` eljárás az, amely a `H0` ciklusváltozó egy adott értékéhez megkeresi a kiegészítő feltételt kielégítő (azaz 8-ra végződő) soron következő `E` értéket, és a ciklusváltozó ezután következő értékét (`H`). Ha a `Max`-nál nem nagyobb értékek között ilyet nem talál akkor meghiúsul.

Ugyanezt az eljárást használhatjuk a felsoroló megoldásban is:

```
% H = 2**i alakú 8-as jegyre végződő egész, amelyre 1 =< H =< Max.
khatvany8(Max, H) :-
        khatvany8(1, Max, H).

% H = 2**i alakú 8-as jegyre végződő egész, amelyre H0 =< H =< Max
% (ahol H0 maga is 2**j alakú).
khatvany8(H0, Max, H) :-
```

```
        következő(H0, Max, E, H1),
        (   H = E
        ;   khatvany8(H1, Max, H)
        ).
```

Alább egymás mellett mutatjuk gyűjtő és felsoroló eljárások általános sémáját:

```
megoldások(V0, Param, L) :-
    következő(V0, Param, E, V1), !,
    L = [E|L1],
    megoldások(V1, Param, L1).
megoldások(_, _, []).
```

```
megoldás(V0, Param, E) :-
    következő(V0, Param, E0, V1),
    (   E = E0
    ;   megoldás(V1, Param, E)
    ).
```

Itt már nyilvánvaló a hasonlóság a kétfajta eljárás között: amikor eljutunk arra a pontra, ahol a gyűjtő eljárásban listát építünk, a felsoroló eljárásban egy diszjunkciót kell elhelyeznünk. Fontos észrevennünk, hogy amikor a felsoroló változatban a `következő` eljárást hívjuk, akkor a megoldást kiadó argumentumba nem az eredmény-változót (E-t) írjuk, hanem egy másik változót, `E0`-t. Ezt azért van így, mert a `következő` eljárás csak a *most következő* eredményt adja ki, ha ezt a végső eredménnyel egyesítenénk, akkor a később előállítandó további eredményeket már nem tudnánk az eredmény-változóba tenni.

Jegyezzük még meg, hogy a fenti sémában a `V0` ciklusváltozó, az `E` eredmény és a `Param` paraméter akár több változót is jelenthet (ill. az utóbbi esetleg el is maradhat). Ez lesz a helyzet a következő példában.

P11 Példa: Fennsíkok

Egy számlistában fennsíknak nevezünk egy csupa azonos elemből álló, maximális, legalább kételemű folytonos részlistát. A maximalitási feltétel itt azt jelenti, hogy fennsík egyik irányba sem terjeszthető ki. Írjunk egy eljárást amely egy adott számlistában felsorolja a benne levő összes fennsík kezdőpozícióját és hosszát. A pozíciókat 1-től számozzuk.

Két megoldást adunk. Az első, gyors-programozásos szemléletű, az `append` szétszedő módjára épít. Ebben felhasználjuk a 4.1.2 alfejezetben definiált `kezdethossz` eljárást és a 4.2.2 szakaszban ismertetett `last` (lista utolsó eleme) könyvtári eljárást is.

```
% Az L listában az F pozíción egy H hosszú fennsík van.
fennsík0(L, F, H) :-
        Teste = [E,E|_],
        append(Eleje, Teste, L),
        \+ last(Eleje, E),
        kezdethossz(Teste, H),
        length(Eleje, F0), F is F0+1.
```

A `fennsík0` eljárás törzsében először létrehozzuk a `Teste` lista-mintát, ez az összes olyan legalább kételemű listával illeszthető, amelynek az első két eleme azonos. Ezt a mintát használjuk az `append` eljáráshívás szétszedő módjában a második részlistaként. Ezután ellenőrizzük, hogy az első listaszelet nem végződik a fennsíkot alkotó elemre (az ellenőrzés, helyesen, akkor is sikerül, ha `Eleje = []`, hiszen a `last` meghiúsul az üres listára). Ha idáig eljutottunk, akkor már biztos van egy fennsíkunk, ennek hosszát a `kezdethossz` eljárással határozzuk meg, majd a `length` beépített eljárás segítségével kiszámoljuk a fennsíkot megelőző kezdőszelet hosszát, és ennek alapján a fennsík kezdőpozícióját.

Ez a megoldás nagyon tömör, de nem a leghatékonyabb. Az `append` ugyanis az összes felbontást felsorolja, köztük azokat is amelyek egy fennsíkot kettévágnak, és csak a kezdőszelet hosszával arányos futási idejű `last` eljárás fogja visszautasítani ezeket.

A korábban ismertetett felsoroló sémát használva most egy hatékony megoldást adunk meg:

```
% Az L listában az F pozíción egy H hosszú fennsík van.
fennsík(L, F, H) :-
        fennsík(L, 1, F, H).
```

```
% Az L0 listában (P0-tól számozva) az F pozíción egy H hosszú fennsík van.
fennsík(L0, P0, F, H) :-
        első_fennsík(L0, P0, F0, H0, L1),
        (   F = F0, H = H0
        ;   P1 is F0+H0, fennsík(L1, P1, F, H)
        ).
```

A sémabeli `következő` eljárásnak itt az `első_fennsík` felel meg, az ottani `V` ciklusváltozónak pedig két változó: a feldolgozandó lista (`L0`), és első elemének indexpozíciója (`P0`). A felsoroláshoz paraméterre itt nincs szükség, viszont két eredmény-argumentumunk van, a fennsík kezdőpozíciója (`F`) és hossza (`H`). Az `első_fennsík` egy determinisztikus eljárás, amely meghatározza az első fennsík jellemzőit (`F0, H0`) és visszadja az azt követő listát (`L1`). Ezután következik a sémából ismert diszjunkció, amelynek második ágán a fennsík adataiból először kiszámoljuk a folytatás kezdő indexpozícióját (`P1`), majd visszahívjuk az eljárást.

Az `első_fennsík` eljárást a következőképpen írhatjuk meg:

```
% első_fennsík(L0, P0, F, H, L): Az L0-ban levő legelső fennsík hossza H,
% az F pozíción van  (P0-tól számozva), és a fennsík utáni maradék lista L.
első_fennsík([E,E|L1], P0, F, H, L) :-
        !, F = P0, azonosak(L1, E, 2, H, L).
első_fennsík([_|L1], P0, F, H, L) :-
        P1 is P0+1,
        első_fennsík(L1, P1, F, H, L).

% azonosak(L0, E, H0, H, L): Az L0-L lista H-H0 darab E elemből áll,
% és L nem kezdődik E-vel.
azonosak([X|L0], E, H0, H, L) :-
        E = X, !,
        H1 is H0+1,
        azonosak(L0, E, H1, H, L).
azonosak(L, _, H, H, L).
```

A fenti `fennsík` eljárás egy futása:

```
| ?- fennsík([1,1,1,1,2,0,0,0,3,3], F, H).
F = 1, H = 4 ? ;
F = 6, H = 3 ? ;
F = 9, H = 2 ? ;
no
```

Most ígéretünk szerint megmutatjuk, hogyan vezethető vissza a korábban (4.1.2) gyorsprogramozási szemléletben definiált `kezdethossz` eljárás a fenti `azonosak` eljárásra:

```
% Az L nem-üres lista első eleme H-szor ismétlődik a lista kezdőszeleteként.
kezdethossz(L, H) :-
        L = [E|L1],
        azonosak(L1, E, 1, H, _).
```

Végezetül egy példát mutatunk arra, hogy ha az összes megoldást összegyűjtöttük, akkor azt könnyedén felsorolhatjuk a `member` eljárás segítségével:

```
khatvany(Max, H) :-
        khatvanyok(Max, Hk), member(H, Hk).
```

Sajnos a fordított eset (felsorolásból gyűjtés) az eddig megismert eszközökkel ilyen röviden nem oldható meg, de szerencsére léteznek erre a célra beépített Prolog eljárások, amelyeket a következő alfejezetben tárgyalunk részletesebben.

4.6. Megoldásgyűjtő beépített eljárások

Az egyik gyakran használt megoldásgyűjtő eljárás a `findall(?Gyűjtő, +Cél, ?Lista)`. A `findall` a `Cél` kifejezést eljáráshívásként értelmezi, meghívja és minden egyes megoldásához előállítja `Gyűjtő` egy másolatát (vagyis a megoldásban levő változókat új változókra cseréli). Végül ezeket a `Gyűjtő` másolatokat egy listába összegyűjti és egyesíti `Lista`-val:

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

A `findall/3` használatával egy a megoldásokat felsoroló eljárásból könnyen előállítható az összes megoldást összegyűjtő program. Az előző alfejezetbeli példa esetében:

```
khatvanyok(Max, Hk) :-
        findall(H, khatvany(Max, H), Hk).
```

A `bagof(?Gyűjtő, +Cél, ?Lista)` eljárás hasonlít a `findall`-hoz, a `Cél` kifejezést eljáráshívásként értelmezi, és összegyűjti a megoldásait. Azonban ha a `Cél`-ban vannak olyan üres változók, amelyek a `Gyűjtő`-ben nem szerepelnek, akkor ezek minden egyes behelyettesítését felsorolja és külön-külön mindegyikhez összegyűjti a `Gyűjtő` összes megoldását `Lista`-ba. Például:

```
gráf([a-b,a-c,b-c,c-d,b-d]).

| ?- gráf(_G), findall(B, member(A-B, _G), VegP).
VegP = [b,c,c,d,d] ? ;
no
| ?- gráf(_G), bagof(B, member(A-B, _G), VegP).
A = a, VegP = [b,c] ? ;
A = b, VegP = [c,d] ? ;
A = c, VegP = [d] ? ;
no
```

Ha a `bagof` eljárás második argumentuma $\mathtt{V}_1$`^ ...`$\mathtt{V}_n$`^Cél` alakú (egzisztenciális kvantifikálás: léteznek olyan $\mathtt{V}_1,\ldots,\mathtt{V}_n$ értékek, hogy `Cél` igaz), akkor a $\mathtt{V}_1,\ldots,\mathtt{V}_n$ változók behelyettesítéseit nem sorolja fel. Ha a `Cél`-beli összes szabad változót így felsoroljuk, akkor a `findall` eljáráshoz hasonló viselkedést kapunk:

```
| ?- gráf(_G), bagof(B, A^member(A-B, _G), VegP).
VegP = [b,c,c,d,d] ? ;
no
```

Jó példa egzisztenciális kvantor használatára a következő `fokszámai` eljárás, ahol egy irányított gráf minden pontjának ki-fokát határozzuk meg, majd gyűjtjük egy listába.

```
% G gráf fokszámlistája FL. A fokszámlista olyan A-F
% párokból áll, ahol A a gráf egy pontja,
% és F>0 az A pont fokszáma.
fokszámai(G, FL) :-
        bagof(A-F, Vk^(bagof(V, member(A-V, G), Vk), length(Vk, F)), FL).

| ?- gráf(_G), fokszámai(_G, FL).
FL = [a-2,b-2,c-1] ? ;
no
```

A kvantort kiküszöbölhetjük egy segédeljárás bevezetésével:

```
% Az A pont foka a G gráfban F>0.
pont_foka(A, G, F) :-
        bagof(V, member(A-V, G), Vk),
        length(Vk, F).

fokszámai(G, FL) :-
        bagof(A-F, pont_foka(A, G, F), FL).
```

Bár a `bagof` és `findall` hasonló eljárások, fontos megemlíteni néhány további különbséget közöttük:

- Ha `Cél`nak nincs megoldása, `findall` üres listát ad, `bagof` meghiúsul.
- Ha `Gyűjtő` nem tömör (van benne üres változó), akkor
  - `findall` ezeket megoldásonként szisztematikusan új változókra cseréli,
  - `bagof` megőrzi a változókat.
- A `bagof` végrehajtása időigényesebb.

Ezeket a különbségeket mutatják be az alábbi példák:

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

A megoldásgyűjtő eljárások csoportjába tartozik még a `setof(?Gyűjtő, :+Cél, ?Lista)`, amely tulajdonképpen ugyanaz mint `bagof`, de az eredménylistát rendezi (az ismétlődések kiszűrésével). A rendezéshez a minden Prolog kifejezésre alkalmazható `@<` összehasonlító beépített eljárást használja (amiről részletesebben a 4.7.4. pontban lesz szó).

Az előző példát tekintve, a gráf pontjainak rendezett listáját nyerhetjük a `setof` eljárás segítségével.

```
gráf([a-b,a-c,b-c,c-d,b-d]).

% Gráf egy pontja P.
pontja(Gráf, P) :-
        member(P-_, Gráf).
pontja(Gráf, P) :-
        member(_-P, Gráf).

% G gráf pontjainak listája Pk.
gráf_pontjai(G, Pk) :-
        setof(P, pontja(G, P), Pk).

| ?- gráf(_G), gráf_pontjai(_G, Pk).
Pk = [a,b,c,d] ? ;
no
```

4.7. Beépített meta-logikai eljárások

Meta-logikai eljárásoknak az olyan, a „tisztán logikai” módszereken túlmutató beépített eljárásokat hívjuk, amelyek

a. a Prolog kifejezések pillanatnyi behelyettesítettségi állapotát tekintik:

- általános kifejezések osztályozása
- általános kifejezések rendezése

b. kifejezéseket szétszednek vagy összeraknak

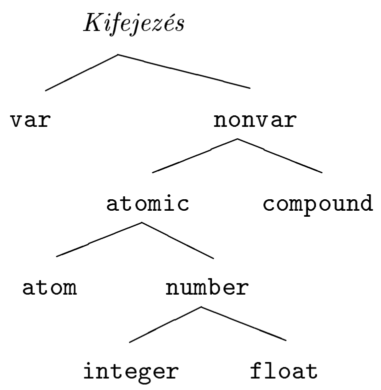
- (struktúra) kifejezés $\Longleftrightarrow$ név és argumentumok
- atomok és számok $\Longleftrightarrow$ karaktereik

Azt, hogy az **a.** típusú eljárások nem tisztán logikaiak, az is jól mutatja, hogy ezek eredménye általában sorrend-függő. Erre mutatunk két példát (a `var` osztályozó és a `@<` összehasonlító eljárások esetére):

```
| ?- var(X) /* X változó? */, X = 1.
X = 1 ?
yes
| ?- X = 1, var(X).
no
| ?- X @< 3 /* X megelőzi 3-t? */, X = 4.
   % a változók megelőzik a nem változó kifejezéseket
X = 4 ?
yes
| ?- X = 4, X @< 3.
no
```

4.7.1. Kifejezések osztályozása

Az osztályozásra szolgáló eljárások megértéséhez először vizsgáljuk meg a kifejezés-osztályok fastruktúráját.



Az osztályok feltüntetett nevei egyben egyargumentumú ellenőrző eljárások is. Ezen eljárásokkal dönthetjük el egy adott kifejezésről, hogy az beletartozik-e az adott osztályba vagy sem. Hogy mire használhatók az osztályozó eljárások?

A teljesség igénye nélkül csak pár gyakori példát említünk. A `var`, `nonvar` — többirányú eljárások esetén — a különböző irányok elágaztatásánál használható. A `compound`, a `number`, és az `atom` eljárás pedig olyan esetekben alkalmazható, amikor nem-megkülönböztetett únió típusú adatokkal dolgozunk.

P12 Példa: A `length/2` beépített eljárás megvalósítása

Az alábbi példa a `length/2` beépített eljárás megvalósítását mutatja be. A `var` osztályozó eljárás segítségével ellenőrizzük, hogy melyek a be- ill. kimenő argumentumok, és ennek megfelelően más-más kódrészt futtatunk.

```
% length(?L, ?N): Az L lista N hosszú.
length(L, N) :-
        var(N), !, length(L, 0, N).
length(L, N) :-
        dlength(L, 0, N).

% length(?L, +I0, -I): Az L lista I-I0 hosszú.
length([], I, I).
length([_|L], I0, I) :-
        I1 is I0+1, length(L, I1, I).

% dlength(?L, +I0, +I): Az L lista I-I0 hosszú.
dlength([], I, I) :- !.
dlength([_|L], I0, I) :-
        I0<I, I1 is I0+1, dlength(L, I1, I).

| ?- length([1,2], Len).
Len = 2 ? ;
no
| ?- length([1,2], 3).
no
| ?- length(L, 3).
L = [_A,_B,_C] ? ;
no
| ?- length(L, Len).
L = [], Len = 0 ? ;
L = [_A], Len = 1 ? ;
L = [_A,_B], Len = 2 ? ;
L = [_A,_B,_C], Len = 3 ?
L = [_A,_B,_C,_D], Len = 4 ?
```

P13 Példa: Szimbolikus kifejezés-feldolgozásra

Az alábbi `deriv` eljárás egy egyszerű deriváló program, mellyel a `+, -, *, /` műveletekkel atomokból és számokból felépített kifejezések deriválását lehet elvégezni, de apró módosítás segítségével ki lehet terjeszteni az eljárást például a szinusz, koszinusz, exponenciális függvényekre is. Itt az `atomic` eljárást használjuk annak eldöntésére, hogy a deriválandó konstans-e.

```
% deriv(Kif, X, D):  Kif-nek az X atom szerinti
%   deriváltja D.  Kif a +, -, *, /  műveletekkel
%   atomokból és számokból felépített kifejezés.
deriv(X, X, D) :- !, D = 1.
deriv(C, _X, D) :-
      atomic(C), !, D = 0.
deriv(U+V, X, DU+DV) :-
      deriv(U, X, DU), deriv(V, X, DV).
deriv(U-V, X, DU-DV) :-
      deriv(U, X, DU), deriv(V, X, DV).
deriv(U*V, X, DU*V + U*DV) :-
      deriv(U, X, DU), deriv(V, X, DV).
deriv(U/V, X, (DU*V - U*DV)/(V*V)) :-
      deriv(U, X, DU), deriv(V, X, DV).
```

```
| ?- deriv(x*y+1, x, DX), deriv(x*y+1, y, DY).
DX = 1*y+x*0+0,
DY = 0*y+x*1+0 ? ;
no

| ?- deriv((x+y)*(2+x), x, D).
D = (1+0)*(2+x)+(x+y)*(0+1) ? ;
no
| ?-
```

4.7.2. Struktúrák szétszedése és összerakása

Az univ eljárás

Az univ eljárást két különböző feladatra használhatjuk, amelyek lényegében egymás ellentettjei:

- struktúrák szétszedése
- struktúrák összerakása, létrehozása egyszerűbb struktúrákból

Tehát az univ tulajdonképpen egy olyan kétargumentumú reláció A és B között, ahol A és B pontosan akkor állnak relációban, ha A „szétbontottja” B. Az univ eljárás szintaxisa:

```
+Kif =.. ?Lista
-Kif =.. +Lista
```

Az univ eljárás a `Kif` kifejezést egy olyan `List`ára bontja (ill. egy olyan `List`ából építi fel), amelyiknek az első eleme a kifejezés neve, a többi eleme pedig a kifejezés argumentumai, a megfelelő sorrendben.

```
| ?- el(a,b,10) =.. L.
L = [el,a,b,10] ?
| ?- el(a,b,10) =.. [F|As].
F = el, As = [a,b,10] ?
| ?- Kif =.. [/,1,2+3].
Kif = 1/(2+3) ?
| ?- [a,b,c] =.. L.
L = ['.',a,[b,c]] ?
```

Emlékeztetjük az olvasót arra, hogy a Prolog nyelv nem engedi meg struktúranév-pozícióban a változót, tehát `Kif = S(X,Y)` nem megengedett! Ehelyett használható viszont a `Kif =..  [S,X,Y]` hívás. Az univ persze ennél is „többet tud”, hiszen megengedi a kifejezés szétbontását akkor is, ha nem ismert az argumentumok száma (ez emlékeztet a `C` nyelv *vararg* nyelvi szerkezetére).

Az univ eljárás építőelemei

Az univ eljárás két egyszerűbb, beépített eljárásra épül, melyek külön-külön is használhatók. Az egyik eljárás a `functor`. Ez az eljárás a `Kif` kifejezés, és funktora, `Név/Argszám` közötti kapcsolatot írja le. Így tehát használható egy adott `Kif` kifejezés nevének (`Név`) és argumentumszámának (`Argszám`) meghatározására. A `functor` az univhoz hasonlóan egy kétirányú eljárás, fordítva is működik, tehát segítségével létre is tudunk hozni egy adott nevű és argumentumszámú kifejezést. A kifejezés az adott funktorúak közül a *legáltalánosabb*, tehát argumentumai különböző változók lesznek. A `funktor` eljárás használata:

```
functor(-Kif, +Nev, +ArgSzam)
functor(+Kif, ?Nev, ?ArgSzam)
```

Megjegyzés: A számok és atomok 0-argumentumúnak számítanak.

```
| ?- functor(szemely(kiss, pal, 1990), Nev, Aszam).
Nev = szemely, Aszam = 3 ?
yes
| ?- functor([a,b,c], Nev, Aszam).
Nev = '.', Aszam = 2 ?
yes
| ?- functor(Str, szemely, 3).
Str = szemely(_A,_B,_C) ?
yes
| ?- functor(25, Nev, Aszam).
Nev = 25, Aszam = 0 ?
yes
```

Az argumentumok vizsgálatára a `arg(+Sorszám, +Kif, ?Arg)` eljárás áll rendelkezésünkre. Ez akkor sikerül, ha a `Kif` kifejezés `Sorszám`-adik argumentuma `Arg`-gal egyesíthető:

```
| ?- arg(2, szemely(kiss, pal, 1990), Arg).
Arg = pal ?
yes
| ?- arg(1, [a,b,c], A_1), arg(2, [a,b,c], A_2).
A_1 = a, A_2 = [b,c] ?
yes
```

Az univ visszavezethető a `functor` és `arg` eljárásokra. Például a

```
Kif =.. [F,A1,A2]
```

hívás helyettesíthető a következő hívássorozattal:

```
functor(Kif, F, 2), arg(1, Kif, A1), arg(2, Kif, A2)
```

Példa:

```
| ?- functor(Str, szemely, 3), arg(1, Str, kiss), arg(2, Str, pal).
Str = szemely(kiss,pal,_A) ?
yes
| ?- Str =.. [szemely,kiss,pal,_].
Str = szemely(kiss,pal,_A) ?
yes
```

A P19 példa bemutatja, hogyan lehet ezt a visszavezetést általánosan elvégezni.

Az univ eljárás alkalmazásai

Legyen feladatunk az, hogy egy `+` és `*` operátorokat tartalmazó kifejezésben a csak számokból álló részkifejezéseket helyettesítsük a részkifejezés számértékével. Az első példa az univ nélküli, a második pedig az univ segítségével történő megvalósítást mutatja be:

P14 Példa: Kifejezések egyszerűsítése, univ nélkül

```
% Az X szimbolikus kifejezés egyszerűsítése EX.
egysz0(X, EX) :- atomic(X), !, EX = X.
egysz0(U+V, EKif) :-
```

```
        egysz0(U, EU), egysz0(V, EV),
        kiszamol(EU+EV, EU, EV, EKif).
egysz0(U*V, EKif) :-
        egysz0(U, EU), egysz0(V, EV),
        kiszamol(EU*EV, EU, EV, EKif).

% EU és EV részekből képzett EUV egyszerűsítése EKif.
kiszamol(EUV, EU, EV, EKif) :-
        number(EU), number(EV), !, EKif is EUV.
kiszamol(EUV, _, _, EUV).

| ?- eriv((x+y)*(2+x), x, D), egysz0(D, ED).
        D = (1+0)*(2+x)+(x+y)*(0+1),
        ED = 1*(2+x)+(x+y)*1 ?
```

P15 Példa: Kifejezések egyszerűsítése, univ segítségével

```
egysz(X, EX) :- atomic(X), !, EX = X.
egysz(Kif, EKif) :-
        Kif =.. [Muv,U,V],    % Kif = Muv(U,V)
        egysz(U, EU), egysz(V, EV),
        EUV =.. [Muv,EU,EV],  % EUV = Muv(EU,EV)
        kiszamol(EUV, EU, EV, EKif).
```

Vegyük észre, hogy a második megoldás sokkal általánosabb: nem csak a `+` és `*` operátorok esetén, hanem az `is/2` által elfogadott összes kétargumentumú műveletre működik.

Az alábbi példák az univ általános kifejezés-bejáró képességét demonstrálják. Az első egy kifejezés kiirató, a másik egy változómentesítő program.

P16 Példa: Speciális kifejezés kiiratás

A feladat az, hogy egy tetszőleges kifejezést kiírjunk úgy, hogy

- az összetett kifejezések alap-struktúra alakban jelennek meg, de
- a kétargumentumú operátorok infix (zárójeles) formában íródnak ki.

Annak eldöntésére, hogy egy `A` atom `F` fajtájú `P` prioritású operátor-e a `current_op(P, F, A)` beépített eljárást használjuk.

```
% Kif-et kiírja a fenti speciális alakban
alapki(Kif) :-
        compound(Kif), !, Kif =.. [Func,A1|ArgL],
        (   current_op(_, Kind, Func),     % Func operátor?
            (Kind = xfy ; Kind = yfx ; Kind = xfx),
            ArgL = [A2] ->      % kétargumentumú használat?
            write('('), alapki(A1), write(' '),
            write(Func), write(' '), alapki(A2), write(')')
        ;   write(Func), write('('), alapki(A1),
            arglistaki(ArgL), write(')')
        ).
alapki(Kif) :- write(Kif).

% Az [A1,...,An] listát ",A1,...,An" alakban kiírja.
arglistaki([]).
```

```
arglistaki([A|AL]) :-
        write(','), alapki(A), arglistaki(AL).

| ?- alapki(1+2+3).
((1 + 2) + 3)

| ?- alapki([1,2]).
.(1,.(2,[]))

| ?- alapki(f(X,2,g(X))).
f(_117,2,g(_117))

| ?- alapki(f(+a, b*c*d, e)).
f(+(a),((b * c) * d),e)
```

P17 Példa: Kifejezés változómentesítése

Tekintsük a következő beépített eljárást:

`numbervars(?Kif, +N0, ?N)`: A `Kif` kifejezést tömörré (`ground`) teszi úgy, hogy a benne szereplő különböző változókat sorra a `'$VAR'`(n_0), `'$VAR'`(n_0+1) … `'$VAR'`(n) struktúrákkal helyettesíti, ahol $n0 =$ `N0`, és `N`-t az n+1 értékkel egyesíti.

Ezt az eljárást például akkor használhatjuk, ha kiiratáskor a változók belső nevei helyett (_⟨szám⟩) egy rövidebb alfanumerikus változónevet szeretnénk megjeleníteni. A `write/1` beépített eljárás ugyanis a `'$VAR'(0)`, `'$VAR(1)`, … struktúrákat rendre az `A`, `B`, … változónevekként jeleníti meg. Ha a `numbervars` eredményeként előálló kifejezés tényleges formáját szeretnénk látni, akkor a `write_term/2` beépített eljárást kell használnunk, a `numbervars(false)` opcióval (a `quoted(true)` opciót pedig azért használjuk az alábbi példában, hogy a `'$VAR'` név aposztrofjelek között jelenjen meg).

```
| ?- Kif = [f(_X),g(_),_X], numbervars(Kif, 0, N),
     write_term(Kif, [quoted(true),numbervars(false)]).
     [f('$VAR'(0)),g('$VAR'(1)),'$VAR'(0)]
N = 2, Kif = [f(A),g(B),A] ?
```

Az univ eljárás segítségével most megírjuk a `numbervars` beépített eljárás egy változatát:

```
% Term változóit sorra a '$myvar'(N0), '$myvar'(N0+1),  ..., '$myvar'(n)
% struktúrákra cseréli, és N = n+1.
numbervars1(Term, N0, N) :- var(Term), !,
        Term = '$myvar'(N0), N is N0+1.
numbervars1(Term, N0, N) :-
        Term =.. [_|Args],
        number_list(Args, N0, N).

% number_list(List, N0, N): a List listában szereplő változókat sorra
% a '$myvar'(N0) ... '$myvar'(n) struktúrákra cseréli, és N = n+1.
number_list([], N, N).
number_list([A|As], N0, N) :-
        numbervars1(A, N0, N1),
        number_list(As, N1, N).

| ?- Kif = [f(_X),g(_),_X], numbervars1(Kif, 0, N).
N = 2,
Kif = [f('$myvar'(0)),g('$myvar'(1)),'$myvar'(0)] ?
```

Az imént bemutatott `numbervars1` alkalmazása a következő példa.

P18 Példa: Két kifejezés azonosságának vizsgálata

Két kifejezést akkor mondunk azonosnak, ha változó-behelyettesítés *nélkül* egyesíthetőek, azaz, ha az egyik változót tartalmaz, akkor a másik ugyanott ugyanazt a változót tartalmazza. Az `azonos/2` `==` néven, `nem_azonos/2` `\==` néven szabványos beépített eljárás (és operátor).

```
nem_azonos(X, Y) :-
        (   numbervars1(X, 0, N), numbervars1(Y, N, _),
            X = Y -> fail
        ;   true
        ).

azonos(X, Y) :- \+ nem_azonos(X, Y).

| ?- azonos(X, 1).
no
| ?- azonos(X, Y).
no
| ?- azonos(X, X).
true ?
| ?- append([], L1, L2), azonos(L1, L2).
L2 = L1 ?
```

Végül bemutatjuk az univ eljárás egy megvalósítását a `functor` és `arg` segítségével.

P19 Példa: Az univ eljárás Prolog megvalósítása

```
Kif =..  [Nev|ArgL] :-
      var(Kif), !,
      atomic(Nev),
      length(ArgL, N),
      functor(Kif, Nev, N),
      arglista(0, Kif, N, ArgL).
Kif =..  [Nev|ArgL] :-
      functor(Kif, Nev, N),
      arglista(0, Kif, N, ArgL).

% arglista(N0, Str, N, L): Str N0 utáni argumentumainak listája
% L, ahol N Str argumentumszáma (Str lehet konstans, amikoris N=0)

arglista(N, _Str, N, []) :- !.
arglista(N0, Str, N, [A|L]) :-
      N0 < N, N1 is N0+1,
      arg(N1, Str, A),
      arglista(N1, Str, N, L).
```

Ez a definíció a hibás hívások kezelésével nem foglalkozik, hiba esetén meghiúsul, vagy valamelyik részeljárás fog hibát jelezni.

4.7.3. Konstansok szétszedése és összerakása

Az univ eljárással kifejezéseket bontottunk fel kifejezésekre, de az univ számára a konstans kifejezések már lényegében felbonthatatlanok. A konstansok karakterekre bontását, ill. ezekből való felépítését teszik lehetővé az alábbi beépített eljárások.

Az `atom_codes(Atom, KódLista)` eljárás a következőképpen működik. Ha híváskor `Atom` ismert, és a $c_1c_2...c_n$ karakterekből áll, akkor a rendszer `KódListát` egyesíti egy $[k_1, k_2, ..., k_n]$ számlistával, ahol k_i a c_i karakterkódja. Ha `Atom` változó, akkor a `KódLista` karakterkód-listából összerak egy nevet, és azt írja be `Atom`-ba.

```
| ?- atom_codes(ab, Cs).
Cs = [97,98] ?

| ?- Cs = [0'b,0'c], atom_codes(Atom, Cs).
Cs = [98,99], Atom = bc ?
```

A `number_codes(Szám, KódLista)` eljárás hasonló kétirányú relációt valósít meg számok esetében. Tehát, ha a `Szám` adott, és tizes számrendszerbeli alakja a $c_1c_2...c_n$ karakterekből áll, akkor `KódLista =` $[k_1, k_2, ..., k_n]$ lesz, ahol k_i a c_i karakterkódja. Ha `Szám` változó, akkor a `KódLista` karakterkód-listából összerak egy számot, és azt írja be `Szám`-ba.

```
| ?- number_codes(1234, Cs).
Cs = [49,50,51,52] ?

| ?- Cs = [0'1,0'2], number_codes(Num, Cs).
Cs = [49,50], Num = 12 ?
```

A konstansok szétszedésére és összerakására szolgáló eljárásokkal szövegmanipulációs feladatokat oldhatunk meg. Ezekre mutatunk most egy-két példát:

```
| ?- use_module(library(lists)).

| ?- atom_codes(ab, _L), reverse(_L, _R),
     append(_L, _R, LR), atom_codes(X, LR).
X = abba, LR = [97,98,98,97] ?

% Rész olyan részatomja Atom-nak, amelyet egy
% vele közvetlenül megegyező részatom követ.
dadogó_rész(Atom, Rész) :-
        atom_codes(Atom, Cs),
        dadogó(Cs, Ds),
        atom_codes(Rész, Ds).

% dadogó(L, D): D olyan nem üres részlistája L-nek,
% amelyet egy vele megegyező részlista követ.
dadogó(L, D) :-
        D = [_|_],
        append(_, Farok, L),
        append(D, Vég, Farok),
        append(D, _, Vég).

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

4.7.4. Kifejezések rendezése: szabványos sorrend

A Prolog kifejezéseknek létezik egy szabványos sorrendje. Ez gyakran hasznos, pl. halmazokat rendezett listaként tudunk ábrázolni. A 4.6. pontban bemutatott `setof` eljárás is ezt az elvet használja: a megoldások halmazából kiszűri az egyformákat, a többit pedig rendezi a szabványos sorrend szerint.

A sorbarendezéshez szükség van egy összehasonlító relációra, amely két *tetszőleges* Prolog kifejezés sorrendjét eldönti.

Vezessük be a szabványos rendezési relációt a következőképpen. Jelentse az $X \prec Y$ formula azt, hogy X megelőzi Y-t a szabványos rendezés szerint. A $\prec$ relációt a következőképpen definiáljuk:

1. Ha X és Y azonos, akkor $X \prec Y$ és $Y \prec X$ egyike sem igaz.
2. Ha X és Y típusa különböző, akkor a típus dönt: *változó* $\prec$ *lebegőpontos szám* $\prec$ *egész szám* $\prec$ *név* $\prec$ *struktúra*.
3. Ha X és Y különböző változók, akkor rendszerfüggő módon vagy $X \prec Y$, vagy $Y \prec X$ igaz.
4. Ha X és Y lebegőpontos vagy egész szám, akkor $X \prec Y \Leftrightarrow X < Y$.
5. Ha X és Y név, akkor sorrendjük megegyezik az abc sorrenddel.
6. Ha X és Y struktúrák:
   (a) Ha X és Y aritása különböző, $X \prec Y \Leftrightarrow X$ aritása kisebb mint Y aritása.
   (b) Egyébként, ha a rekordok neve különböző, $X \prec Y \Leftrightarrow X$ neve $\prec Y$ neve.
   (c) Egyébként balról az első nem azonos argumentum dönt (lexikografikus rendezés).

Végtelen (ciklikus) kifejezésekre a fenti rendezés nem érvényes.

Kifejezések összehasonlítására az alábbi hat beépített eljárás szolgál:

`(==)/2, (\==)/2, (@<)/2, (@=<)/2, (@>)/2, (@>=)/2`

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

Ezen eljárások minden argumentuma tisztán bemenő tetszőleges kifejezés.

Fontos megjegyezni, hogy ezek az eljárások logikailag nem tiszták, hiszen a rendezés a pillanatnyi behelyettesítettségtől függ:

```
| ?- X @< 3, X = 4.
X = 4 ?
yes
| ?- X = 4, X @< 3.
no
```

Azt is fontos szem előtt tartani, hogy mindig a szabványos belső struktúra-alak szerint rendezzük a kifejezéseket, pl.

```
| ?- [1, 2, 3, 4] @< struktúra(1, 2, 3).
```

sikerül (ld. 6(a) szabály), hiszen a bal oldalon `.(1,...)` struktúra áll.

Az alábbi példa az eddig megismert beépített eljárások segítségével valósítja meg a $\prec$ rendezési relációt.

P20 Példa: $\prec$ egy megvalósítása

```
% T1 megelőzi T2-t a szabványos sorrendben (lényegében T1 @< T2), de
% a változókat az első előfordulás szerint rendezi.
precedes(T1, T2) :-
        \+ \+ (numbervars1(T1-T2, 0, _), prec(T1, T2)).

% T1 megelőzi T2-t, ahol mindkettőben a változók '$myvar'(n)
% konstansokra vannak már cserélve.
prec(T1, T2) :-
        class(T1, C1), class(T2, C2),
        (   C1 =:= C2 ->
            (   C1 =:= 1 -> T1 < T2   % 4. szabály (lebegőpontos szám)
            ;   C1 =:= 2 -> T1 < T2   % 4. szabály (egész szám)
            ;   struct_prec(T1, T2)   % 3., 5. és 6. szabály
            )                         % (változó, név, struktúra)
        ;   C1 < C2
        ).

% class(T, C): T a C osztályba tartozik
% (0 -> változó, 1 -> lebegőpontos, 2 -> egész, 3 -> atom, 4 -> struktúra).
class('$myvar'(_), C) :-   !, C = 0.
class(T, C) :- float(T),   !, C = 1.
class(T, C) :- integer(T), !, C = 2.
class(T, C) :- atom(T),    !, C = 3.
class(_T, 4).

% S1 megelőzi S2-t (struktúra-kifejezésekre és atomokra).
struct_prec(S1, S2) :-
        functor(S1, F1, N1), functor(S2, F2, N2),
        (   N1 =:= N2 ->
            (   F1 = F2 ->
                args_prec(1, N1, S1, S2)
            ;   atom_prec(F1, F2)
            )
        ;   N1 < N2
        ).

% Az S1 struktúra-kifejezés N0, ..., N sorszámú argumentumai
% lexikografikusan megelőzik S2 azonos sorszámú argumentumait.
args_prec(N0, N, S1, S2) :-
        N0 =< N, arg(N0, S1, A1), arg(N0, S2, A2),
        (   A1 = A2 -> N1 is N0+1,
            args_prec(N1, N, S1, S2)
        ;   prec(A1, A2)
        ).

% A1 atom megelőzi A2 atomot (előfeltétel: A1 \= A2).
atom_prec(A1, A2) :-
        atom_codes(A1, C1), atom_codes(A2, C2),
        struct_prec(C1, C2).
```

4.8. Egyenlőségfajták

Prologban sokfajta beépített eljárás van, amelyben az egyenlőség fogalma szerepel. Ennek az alfejezetnek a célja az ilyen eljárások egybegyűjtése és a köztük lévő különbségek áttekintése. Vessük össze az egyenlőség-szerű, majd a nemegyenlő-szerű eljárásokat, s végül nézzünk pár példát az elvek illusztrálására. (Az alábbi példákban `X` egy behelyettesítetlen változó.)

A Prolog egyenlőség-szerű beépített eljárásai:

- *U* `=` *V*: *U* egyesítendő *V*-vel. Soha sem jelez hibát. Pl. `X = 1+2` eredménye az `X = 1+2` behelyettesítés.
- *U* `==` *V*: *U* azonos *V*-vel. Soha sem jelez hibát és soha sem helyettesít be. Pl. `X == 1+2` meghiúsul.
- *U* `is` *V*: *U* egyesítendő a *V* aritmetikai kifejezés értékével. Hiba, ha *V* nem (tömör) aritmetikai kifejezés. Pl. `X is 1+2` eredménye az `X = 3` behelyettesítés.
- *U* `=:=` *V*: Az *U* és *V* aritmetikai kifejezések értéke megegyezik. Hibát jelez, ha *U* vagy *V* nem (tömör) aritmetikai kifejezés. Pl. `X =:= 1+2` hibát jelez.

A Prolog nemegyenlő-szerű beépített eljárásai (az alábbi eljárások egyike sem helyettesít be változót):

- *U* `\=` *V*: *U* nem egyesíthető *V*-vel. Soha sem jelez hibát. Pl. `X \= 1+2` meghiúsul.
- *U* `\==` *V*: *U* nem azonos *V*-vel. Soha sem jelez hibát. Pl. `X \== 1+2` sikerül.
- *U* `=\=` *V*: A *U* és *V* aritmetikai kifejezések értéke különbözik. Hibát jelez, ha *U* vagy *V* nem (tömör) aritmetikai kifejezés. Pl. `X =\= 1+2` hibát jelez.

Példák az egyenlőség-szerű eljárások használatára (`X` és `Y` behelyettesítetlen változók):

U	*V*	*U* = *V*	*U* \= *V*	*U* == *V*	*U* \== *V*	*U* is *V*	*U* =:= *V*	*U* =\= *V*
1	2	nem	igen	nem	igen	nem	nem	igen
a	b	nem	igen	nem	igen	hiba	hiba	hiba
1+2	+(1,2)	igen	nem	igen	nem	nem	igen	nem
1+2	2+1	nem	igen	nem	igen	nem	igen	nem
1+2	3	nem	igen	nem	igen	nem	igen	nem
3	1+2	nem	igen	nem	igen	igen	igen	nem
X	1+2	X=1+2	nem	nem	igen	X=3	hiba	hiba
X	Y	X=Y	nem	nem	igen	hiba	hiba	hiba
X	X	igen	nem	igen	nem	hiba	hiba	hiba

Jelmagyarázat

- igen ⇒ siker.
- nem ⇒ meghiúsulás.

Az alábbi párbeszéd bemutatja a fenti táblázatban szereplő példák egy részét.

```
| ?- X = 1+2.               % X egyesíthető a +(1, 2) struktúrával
X = 1+2 ?
yes
| ?- X == 1+2.              % egy változó nem azonos +(1, 2)-vel
no
| ?- X is 1+2.              % X egyesíthető 1 + 2 értékével
X = 3
yes
| ?- X =:= 1+2.             % X-et nem lehet kiértékelni
{INSTANTIATION ERROR: _32=:=1+2 - arg 1}
| ?- +(1,2) = 1+2.          % +(1,2) egyesíthető 1+2-vel...
```

```
yes
| ?- +(1,2) == 1+2.        %   és ráadásul azonos is
yes
| ?- +(1,2) is 1+2.        % +(1,2) nem egyesíthető 1+2 értékével
no
| ?- +(1,2) =:= 1+2.       % a két kifejezés értéke egyenlő
yes
| ?- .(1,'[]') == [1].     % a két struktúra azonos
yes
| ?-
```

4.9. Modularitás

Ebben az alfejezetben röviden ismertetjük a SICStus Prolog modulkezelésének alapelveit. A Prolog nyelvhez kidolgozott különféle modul-fogalmakat a 6.1 alfejezetben tekintjük majd át.

A SICStus Prolog minden eljárást valamilyen modulban helyez el. Amíg nem intézkedünk másképp, az eljárások alaphelyzetben a `user` modulba kerülnek.

Ha Prolog programunkat strukturálni kívánjuk, akkor ún. modul-állományokat kell létrehoznunk. Egy állományban egy modult tudunk elhelyezni, az állomány első programeleme egy modul-parancs kell, hogy legyen:

```
:- module(Modulnév, [Funktor1, Funktor2, ...]).
```

Itt *Funktor1*, ... a modulból exportálni kívánt eljárások funktorai (azaz Név/Aritás alakú kifejezések, ahol Név egy atom, Aritás egy egész).

Például, ha a korábban definiált `fennsík/3` eljárást egy modulba kívánjuk foglalni, akkor a szükséges eljárásokat be kell írnunk egy állományba, mondjuk `plato.pl`-be, és ennek az állománynak az elejére el kell helyeznünk a következő parancsot:

```
:- module(plató_keresés, [fennsík/3]).
```

Ha ezután be akarjuk tölteni ezt a modult, akkor a SICStus rendszer promptjánál ki kell adnunk egy `use_module` parancsot, argumentumában az állománynévvel:

```
:- use_module(plato).
```

Ez a parancs az adott állományban levő modult betölti, és az általa exportált összes eljárást importálja a kurrens modulba (példánkban a `user` modulba). Ezáltal az importált eljárások ebből a modulból hívhatókká válnak. Ugyanezt a beépített eljárást használhatjuk a SICStus könyvtárak betöltésére is, pl. a `lists` könyvtárat a következőképpen tölthetjük be:

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

A `use_module` beépített eljárásnak van egy kétargumentumú változata, ez betölti a modult, de csak azokat az eljárásokat importálja, amelyek funktorai szerepelnek a második argumentumbeli import-listában. Például:

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

csak a `last/2` eljárást fogja láthatóvá tenni, a többi könyvtári eljárást nem. Ekkor például lehet egy saját `append` eljárásunk, anélkül, hogy ez a könyvtári példánnyal összeütközésbe kerülne.

A `use_module` parancs szerepelhet egy állományban, akár egy modul-állományban is. Ez utóbbi esetben csak ebbe a modulba fogja importálni a betöltött eljárásokat. Ugyanazt a modul-állományt több modulba is betölthetjük, ez nem jár felesleges memória-foglalással, mivel a SICStus Prolog rendszer az eljárásokat csak egy példányban tárolja.

Megjegyezzük, hogy a SICStus Prolog modulfogalma nem szigorú. Bármely betöltött eljárás meghívható, ha az ún. modul-kvalifikált hívási formát használjuk, azaz az eljáráshívás elé írjuk a modulnevet, attól a kettőspont operátorral elválasztva. Ha például a fennsík-kereső programot a fent példaként idézett módon foglaltuk modulba, és azt betöltjük, akkor a `fennsík/3` eljárást modul-kvalifikálás nélkül tudjuk hívni, de a többi eljárást is meghívhatjuk, például:

```
| ?- plató_keresés:első_fennsík([1,2,2,3], 4, F, H, L).
```

A SICStus rendszernek ez a tulajdonsága különösen hasznos modularizált programok nyomkövetésénél.

Végezetül következzék néhány, a magasabbrendű eljárások használata során felmerülő, a modularitással kapcsolatos fontos tudnivaló. A magasabbrendű (meta-) eljárás egy olyan eljárás, amelynek egy másik eljárás az argumentuma. Tipikus példák a meta-eljárásokra a 4.6. pontban bemutatott `findall, bagof, setof` eljárások. Világos, hogy ha modulközi meta-eljárásokat írunk, azaz a meta-eljárás által meghívandó eljárás más modulban van, akkor szükség van arra, hogy az eljárás meta-argumentumát modul-kvalifikált módon adjuk át. Ezért a meta-eljárásokra egy `meta_predicate` deklarációt kell megadnunk, amelyben jelezzük, hogy melyek az eljárás-argumentumok.

A meta-deklaráció formája:

```
:- meta_predicate ⟨eljárásnév⟩(⟨argleíró1⟩, ...).
```

Itt az ⟨argleíró$_i$⟩ lehet a `:` jel, annak jelzésére, hogy az adott argumentum egy eljárás, vagy bármilyen más atom a többi argumentum jelzésére. Ez utóbbi helyeken szokás a be- ill. kimenő jellegre utaló jeleket elhelyezni (`+`,`-`,`?`).

Például a `bagof` beépített eljárásra a következő deklaráció vonatkozik (ezt természetesen nem kell megadni, a rendszer beépítve tartalmazza):

```
:- meta_predicate bagof(?, :, ?).
```

4.10. Magasabbrendű eljárások

A magasabrendű eljárások tehát olyan eljárások, melyek argumentuma lehet egy másik eljárás. A megoldásgyűjtő eljárások is ide tartoznak, hiszen az az eljárás, melynek a megoldását gyűjtjük szükségképpen argumentuma a gyűjtő eljárásnak.

Először nézzünk két példát arra, hogyan lehet a `findall` megoldásgyűjtő eljárás segítségével lista-műveleteket definiálni, rekurzió nélkül.

```
% Az L egész-lista páros elemeinek listája Pk.
páros_elemei(L, Pk) :-
        findall(X, (member(X, L), X mod 2 =:= 0), Pk).

% Az L számlista  elemei négyzeteinek listája Nk.
négyzetei(L, Nk) :-
        findall(Y, (member(X, L), Y is X*X), Nk).

| ?- páros_elemei([1,2,3,4], Pk).
Pk = [2,4] ?

| ?- négyzetei([1,2,3,4], Nk).
Nk = [1,4,9,16] ?
```

4.10.1. Meta-eljárások megoldásgyűjtő eszközökkel

Az előző példa általánosításaként megmutatjuk, hogy nem csak egy lista páros elemeit választhatjuk ki, hanem tetszőleges Prolog predikátum szerint is megszűrhetjük a listát. A felhasznált magasabbrendű eljárás itt ismét a `findall`:

```
% Az L lista X elemeinek Pred szerinti szűrése FL.
:- meta_predicate filter(+, ?, :, -).
filter(L, X, Pred, FL) :-
        findall(X, (member(X, L), call(Pred)), FL).

| ?- filter([1,2,3,4], X, X mod 2 =:= 0, Pk).
Pk = [2,4] ? ;
```

A meta-argumentum meghívása a `call(Cél)` beépített eljárással történik, ez a `Cél` (atom vagy struktúra) kifejezést hívássá alakítja és végrehajtja. Megengedett, hogy hívásként egy `X` változó szerepeljen, ez ekvivalens `call(X)` hívással, tehát a fenti `findall` hívást így is írhatjuk:

```
        findall(X, (member(X, L), Pred), FL).
```

Megint a `findall` felhasználásával mutatunk be egy példát, amely egy lista minden elemére alkalmaz egy a lista elemein értelmezett leképezést. Ez a funkcionális nyelvekből jól ismert `map` fügvénynek megfelelő Prolog eljárás:

```
% Az L lista X elemeit Pred Y-ba képezi le.
% A kapott Y értékek listája ML.
:- meta_predicate map(+, ?, :, ?, -).
map(L, X, Pred, Y, ML) :-
        findall(Y, (member(X, L), Pred), ML).

| ?- map([1,2,3,4], X, Y is X*X, Y, Nk).
Nk = [1,4,9,16] ?
```

4.10.2. Részlegesen paraméterezett eljárások

A fenti `map` eljárás második, harmadik és negyedik argumentuma felfogható egyetlen argumentumként — egy részlegesen paraméterezett eljárásként. Ez a három kifejezés együtt egy olyan predikátumot ír le, amelynek argumentumai `X` és `Y`, és törzse `Pred`. Ha explicitté tesszük ezt az eljárást:

```
negyzet(X, Y) :- Y is X*X.
```

akkor a hívási példa így alakul:

```
| ?- map([1,2,3,4], X, negyzete(X,Y), Y, Nk).
```

Elképzelhető, hogy az elvégzendő műveletekhez további paraméterekre is szükség van, mint az alábbi példában:

```
masodfoku(L0, P, Q, L) :-
        map(L0, X, masodfoku_ertek(P,Q,X,Y), Y, L).

masodfoku_ertek(P, Q, X, Y) :-
        Y is X*X + P*X + Q.
```

Ha bevezetjük azt a konvenciót, hogy a magasabbrendű műveletekben érdekelt argumentumok mindig a predikátum utolsó argumentum-pozícióin vannak, akkor egyetlen kifejezéssel ábrázolhatjuk az elvégzendő magasabbrendű műveletet.

Célunk tehát egy `map/3` eljárás megírása, amelynek jelentése a következő:

```
% map(L0, PartialPred, L): Az L0 lista minden elemére alkalmazva a
% PartialPred által előírt leképezést kapjuk az L listát.
```

A `PartialPred` struktúrakifejezés által előírt leképezést úgy végezhetjük el, hogy a struktúrához két argumentumot hozzáveszünk. Ezek közül az első a leképezendő érték, a második pedig a leképezés eredménye. Az így kiegészített struktúrakifejezés meghívásával végezzük el az eredmény kiszámítását. A fenti két példa esetén a `map/3` hívása tehát a következőképpen alakul:

```
masodfoku(L0, P, Q, L) :-
        map(L0, masodfoku_ertek(P,Q), L).

negyzete(L0, L) :-
        map(L0, negyzet, L).
```

Látjuk tehát, hogy a `map` eljárásnak olyan kifejezéseket adunk át, amelyek önmagukban nem hívhatók meg, hanem csak úgy, hogy ezeket a kifejezéseket ellátjuk két további paraméterrel. Ezeket a kifejezéseket tehát joggal nevezhetjük részlegesen paraméterezett eljáráshívásoknak.

Mielőtt a `map/3` eljárást a következő alfejezetben bemutatnánk, néhány segédeljárást vezetünk be. Ezeknek az lesz a feladata, hogy egy részlegesen paraméterezett eljáráshívás és a hiányzó paraméterek alapján állítsa össze a „teljesen" paraméterezett eljáráshívást, és azt hajtsa is végre.

A `call1/2` egy paraméterrel egészit ki egy részlegesen paraméterezett eljáráshívást, a `call2/3` kettővel stb. Ezek az eljárások számos Prolog megvalósításban beépítettek, de a SICStusban nem. Ezért ezeket az alábbi módon kell definiálnunk:

```
% Pred utolsó argumentumként A-val kiegészítve igaz (sikeresen fut le).
call1(Pred, A) :-
        Pred =.. FArgs, append(FArgs, [A], FArgs1),
        Pred1 =.. FArgs1, call(Pred1).

% Pred utolsó két argumentumként az A és B kifejezésekkel kiegészítve igaz.
call2(Pred, A, B) :-
        Pred =.. FArgs, append(FArgs, [A,B], FArgs2),
        Pred2 =.. FArgs2, call(Pred2).

% Pred utolsó három argumentumként az A, B és C kifejezésekkel kiegészítve igaz.
call3(Pred, A, B, C) :-
        Pred =.. FArgs,
        append(FArgs, [A,B,C], FArgs3),
        Pred3 =.. FArgs3, call(Pred3).
```

4.10.3. Részleges paraméterezést használó magasabbrendű eljárások

A `call2/3` eljárás felhasználásával megírhatjuk a `map` rekurzív definícióját. Az előző megvalósításban a `map` a lista elemein a `member` eljárás alkalmazásával ment végig (minden egyes elem után visszalépéssel). Az alábbi változatban pedig az általánosabban alkalmazható rekurzív listabejárást alkalmazzuk.

```
% map(Xs, Pred, Ys): Az Xs lista elemeire a Pred transzformációt
% alkalmazva kapjuk az Ys listát.
map([X|Xs], Pred, [Y|Ys]) :-
        call2(Pred, X, Y),
        map(Xs, Pred, Ys).
map([], _, []).

negyzet(X, Y) :- Y is X*X.
```

```
| ?- map([1,2,3,4], negyzet, L).
L = [1,4,9,16] ? ;
no
```

A `map` eljárás megvalósítása során választhattunk, hogy melyik módszert használjuk: a `findall`-t használjuk vagy rekurzívan dolgozzuk fel a listát. A funkcionális nyelvekből ismert `fold` jellegű eljárások esetén, amelyek egy művelet ismételt elvégzését biztosítják, szükség van az előző müvelet eredményére, ezért nem élhetünk az elemeket egymástól függetlenül feldolgozó `findall`-os módszerrel. Így a `foldl` és `foldr` alábbi definíciójában részlegesen paraméterezett eljárásokat használunk, és a `call3` eljárás segítségével végezzük el a meta-predikátumok meghívását.

```
% foldl(Xs, Pred, Y0, Y): Az Xs elemeire balról
% jobbra alkalmazott, a Pred által leírt
% kétargumentumú függvény Y0 kezdőértékre
% alkalmazott eredménye Y.
foldl([X|Xs], Pred, Y0, Y) :-
        call3(Pred, X, Y0, Y1),
        foldl(Xs, Pred, Y1, Y).
foldl([], _, Y, Y).

% foldr(Xs, Pred, Y0, Y): Az Xs elemeire jobbról
% balra alkalmazott, a Pred által leírt függvény
% Y0 kezdőértékre alkalmazott eredménye Y.
foldr([X|Xs], Pred, Y0, Y) :-
        foldr(Xs, Pred, Y0, Y1),
        call3(Pred, X, Y1, Y).
foldr([], _, Y, Y).

jegyhozzá(A, J, E0, E) :- E is E0*A+J.
```

Példák a fenti meta-eljárások használatára:

```
| ?- foldr([1,2,3], jegyhozzá(10), 0, E).
E = 321 ?

| ?- foldl([1,2,3], jegyhozzá(10), 0, E).
E = 123 ?
```

4.11. Dinamikus adatbáziskezelés

A Prolog nyelv lehetővé teszi, hogy az ún. dinamikus eljárásokat futási időben módosítsuk, hozzáadjunk ill. elvegyünk klózokat. Ezek az eljárások ugyan általában lassabbak, mint a statikusak, de jelentősen megnövelik a programozó lehetőségeit, szabadságát. A dinamikus eljárásokat a

```
:- dynamic(Eljárásnév/Argumentumszám).
```

deklaráció vezeti be, ennek az adott eljárás első (a program szövegében megadott) klóza előtt kell szerepelnie.

4.11.1. Beépített eljárások

A következő eljárások segítségével klózokat vehetünk fel, kérdezhetünk le illetve törölhetünk a programból.

Az `asserta(:@Klóz)`[1] és az `assertz(:@Klóz)` eljárások a `Klóz` kifejezést klózként értelmezik és felveszik a programba. A különbség közöttük, hogy az `asserta` a predikátum első klózaként veszi fel, az `assertz` pedig az utolsóként.

```
| ?- assertz((p(1,X):-q(X))), asserta(p(2,0)),
     assertz((p(2,Z):-r(Z))), listing(p).
p(2, 0).
p(1, A) :-
        q(A).
p(2, A) :-
        r(A).
```

A klózokat természetesen nem csak felvenni tudjuk a dinamikus adatbázisba, hanem szükség esetén ki is törölhetjük őket. Erre a célra szolgál a `retract(:@Klóz)` eljárás, amely keres egy olyan dinamikus eljárást, melynek van a `Klóz`-zal egyesíthető klóza. Az első megfelelő klózzal illeszti majd kitörli a klózt.

```
| ?- retract((p(2,_):-_)), listing(p),
     write(----------), nl, fail.
p(1, A) :-
        q(A).
p(2, A) :-
        r(A).
----------
p(1, A) :-
        q(A).
----------
no
```

Fontos megjegyezni, hogy az eljárás *többszörösen sikerülhet*, ugyanis visszalépéskor az eljárás újabb klózt keres, illeszti, majd kitörli azt is. A fenti példában is látszik, hogy a visszalépések során az első és harmadik klózt is kitörölte a `retract` hívás. A `retract` többszörös sikerességét felhasználva készítettük el a következő eljárást:

```
% retractall(Fej): kitörli az összes klózt,
% amelynek feje illeszthető Fej-jel.
retractall(Head) :- retract((Head :- _)), fail.
retractall(_).
```

A `retractall(:@Fej)` hívás kitörli az összes klózt amelynek feje illeszthető `Fej`-jel. Mindig sikerül (akkor is ha nincs kitörlendő klóz). Ez az eljárás beépítve megtalálható a SICStus Prologban.

Az adatbázisban klózok felvétele és törlése mellett le is kérdezhetjük azt, hogy milyen klózok szerepelnek. A `clause(:+Fej, ?Törzs)` eljárás segítségével megvizsgálhatjuk, hogy létezik-e egy interpretált (F:-T) klóz, amely egyesíthető a (`Fej`:-`Törzs`) struktúrával. Ha létezik, akkor a `(Fej :- Törzs)` klózzal illeszthető első klózt megkeresi és illeszti. Ez az eljárás is *többszörösen sikerülhet*, hasonlóan a `retract`-hoz.

4.11.2. Alkalmazási példák dinamikus predikátumokra

P21 Példa: Legnagyobb megoldás keresése

Egy `p(N)` eljárás legnagyobb megoldásának keresése (feltesszük, hogy az `N` megoldás mindig pozitív szám):

[1] Emlékeztetjük az olvasót, hogy a `@` jelölés arra utal, hogy az adott argumentum tisztán bemenő, azaz a benne szereplő változók nem kaphatnak értéket; részletesebben lásd a 3.6.3. és 5.1. pontokban . A `:` jel pedig azt jelzi, hogy az argumentum modulkvalifikált, azaz pl. egy távoli modulba is felvehetünk egy klózt, ha az `assert` paraméterében egy modulnevet szerepeltetünk, a klóztól egy `:` operátorral elválasztva.

```
max_p(_):-
      retractall(legnagyobb(_)),          % egy előző, megszakadt futásból maradhatott
      asserta(legnagyobb(0)),
      p(N),
      legnagyobb(Max),
      (  Max >= N -> fail
      ;  retract(legnagyobb(_)), asserta(legnagyobb(N)), fail
      ).
max_p(N):-
      retract(legnagyobb(N)).
```

P22 Példa: Egyszerű `findall` megvalósítása

Az alábbi `findall1` egyszerűbb mint a beépített, mert skatulyázva nem működik.

```
:- dynamic(solution/1).

:- meta_predicate findall1(?,:,?).

findall1(Templ, Goal, _Sols) :-
        call(Goal),
        asserta(solution(Templ)),  fail.
findall1(_Templ, _Goal, Sols) :-
       collect_sols([], Sols).

% A solution tényállítások argumentumában tárolt
% kifejezések megfordított listája L-L0.
collect_sols(L0, L) :-
        retract(solution(S)), !,
        collect_sols([S|L0], L).
collect_sols(L, L).

| ?- findall1(Y, (member(X, [1,2,3]), Y is X*X), S).
S = [1,4,9] ? ;
no
```

P23 Példa: Egy egyszerű nyomkövető interpreter

```
interp(true, _) :- !.
interp((G1, G2), D) :- !,
        interp(G1, D), interp(G2, D).
interp(G, D) :-
        (   trace(G, D, call)
        ;   trace(G, D, fail), fail
        ),
        D2 is D+2,
        clause(G, B), interp(B, D2),
        (   trace(G, D, exit)
        ;   trace(G, D, redo), fail
        ).

trace(G, D, Port) :-
        tab(D), % D szóközt ír ki
        write(Port), write(': '), write(G), nl.
```

Az interpreter működése - példafutás

```
:- dynamic app/3, app/4.

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

app(L1, L2, L3, L123) :-
        app(L1, L23, L123), app(L2, L3, L23).

| ?- interp(app(_, [b,c], L, [c,b,c,b]), 0).
call: app(_117,[b,c],_167,[c,b,c,b])
  call: app(_117,_572,[c,b,c,b])
  exit: app([],[c,b,c,b],[c,b,c,b])
  call: app([b,c],_167,[c,b,c,b])
  fail: app([b,c],_167,[c,b,c,b])
  redo: app([],[c,b,c,b],[c,b,c,b])
    call: app(_779,_572,[b,c,b])
    exit: app([],[b,c,b],[b,c,b])
  exit: app([c],[b,c,b],[c,b,c,b])
  call: app([b,c],_167,[b,c,b])
    call: app([c],_167,[c,b])
      call: app([],_167,[b])
      exit: app([],[b],[b])
    exit: app([c],[b],[c,b])
  exit: app([b,c],[b],[b,c,b])
exit: app([c],[b,c],[b],[c,b,c,b])

L = [b] ?
```

4.12. Nyelvtani elemzés Prologban — Definite Clause Grammars

A Prolog visszalépéses keresési módszere nagyszerűen alkalmazható nyelvtanok elemzésére is.

4.12.1. Példasor: számok elemzése

Tekintsük a bináris számokat leíró alábbi egyszerű nyelvtant!

⟨szám⟩ ::=	⟨jegy⟩ ⟨számmaradék⟩
⟨számmaradék⟩ ::=	⟨jegy⟩ ⟨számmaradék⟩ | ϵ
⟨jegy⟩ ::=	0 | 1

Tételezzük fel, hogy beolvastunk egy karakter-sorozatot és a kapott karakter-kód listáról szeretnénk eldönteni, hogy az megfelel-e pl. a fenti szabályok által leírt szintaxisnak. Ehhez célszerű a nyelvtan minden egyes nem-terminális jelének (pl. ⟨szám⟩, ⟨számmaradék⟩, ...) egy kétargumentumú Prolog eljárást megfeleltetnünk (az egyszerűség kedvéért ugyanazt a nevet adva neki mint a nem-terminálisnak). Az eljárás első argumentuma egy karakter-kód lista lesz, és azt várjuk tőle, hogy sikerüljön, ha ennek a listának egy kezdőszelete kielemezhető az adott nem-terminálisnak megfelelő szabályok szerint; egyébként hiúsuljon meg. Siker esetén elvárjuk, hogy a második argumentumban adja vissza a bemenő listának a kielemzett kódsorozat elhagyásával keletkező zárószeletét. A két lista kapcsolatát röviden úgy fogjuk leírni, hogy a bemenő listáról *leelemezhető* az adott nem-terminális és marad a kimenő lista.

Tekintsük a ⟨szám⟩ nem-terminálist, és vizsgáljuk meg, milyen Prolog eljárás feleltethető meg neki a fenti elvek szerint:

```
% L0 kódlistáról leelemezhető egy <szám>, marad L.
szám(L0, L) :-
        jegy(L0, L1), számmaradék(L1, L).
```

A Prolog elemző követi a nyelvtani szabály szerkezetét: az eljárástörzsben sorra hívjuk a szabály jobboldalán szereplő nem-terminálisoknak megfeleltetett eljárásokat, a lista-argumentumokat akkumulátor-szerűen összekapcsolva. Ez nem meglepő, hiszen akkumulátorokat állapotváltozások leírására tudunk használni, és itt is van egy állapot-fogalmunk: a bemenő karakterfolyamból még le nem elemzett karakterek listája.

A `számmaradék` nem-terminálisnak a jobboldalán két alternatíva áll, ebből két Prolog szabályt készítünk. A második, üres alternatívából egy olyan ága keletkezik az elemzőnek, amely változtatás nélkül visszaadja a bemenő listát:

```
% Leelemezhető jegyek egy esetleg üres listája.
számmaradék(L0, L) :-
        jegy(L0, L1), számmaradék(L1, L).
számmaradék(L, L).
```

Miután az akkumulátor-párok láncolása egy mechanikus feladat, ez rábízható a Prolog fordítóprogramra. A Prolog rendszerek többsége ugyanis megengedi, hogy programelemként ún. definit klóz nyelvtani (Definite Clause Grammar, DCG) szabályokat szerepeltessünk. Ezeket a Prolog rendszer a program beolvasása során Prolog szabályokká alakítja.

A fenti két Prolog eljárásnak megfelelő DCG szabály:

```
% Leelemezhető egy szám
szám -->
        jegy, számmaradék.

% Leelemezhető jegyek egy esetleg üres listája.
számmaradék -->
        jegy, számmaradék.
számmaradék --> [].
```

A szabály bal- és jobboldalát tehát a `-->` operátorral kell elválasztani, az egymás után írást a `','`, az üres jelsorozatot a `[]` jelöli. Fontos megjegyezni, hogy DCG „tényállítások” nincsenek, tehát az üres törzsű nyelvtani szabályokat úgy kell a DCG jelölésre átírni, hogy a törzsbe `[]`-t írunk (lásd a `számmaradék` második klózát).

Vegyük észre, hogy a `számmaradék` eljárás nem-determinisztikus: először megpróbálja a lehető leghosszabb jegy-sorozatot leelemezni, de visszalépés esetén hajlandó ennek minden kezdőszeletét felsorolni, hiszen ezek mind megfelelnek a ⟨számmaradék⟩ szintaxisának. Ez utóbbi megoldások általában nem vezetnek eredményre, mert „normális” nyelvtan egy ⟨szám⟩ után nem enged meg ⟨jegy⟩-gyel kezdődő nem-terminálist (hiszen akkor sokértelmű lenne az elemzés). Hogy elkerüljük a felesleges visszalépéseket, célszerű a `számmaradék` első DCG szabályában egy vágót elhelyezni, amely átkerül a szabályból generált Prolog kódba is. Ezzel biztosítjuk, hogy ⟨számmaradék⟩-nak csak a maximális hosszúságú jegy-sorozatot tekintjük:

```
% Leelemezhető jegyek egy maximális esetleg üres listája.
számmaradék -->
        jegy, !, számmaradék.
számmaradék --> [].
```

A fenti DCG szabályokból a beolvasás során az előzőleg leírt Prolog eljárások keletkeznek:

```
| ?- listing.
szám(A, B) :-
        jegy(A, C),
        számmaradék(C, B).

számmaradék(A, B) :-
        jegy(A, C), !,
        számmaradék(C, B).
számmaradék(A, B) :-
        B=A.
yes
```

A DCG szabályokban megengedett a diszjunktív jelölés is, a ; használatával. Például a `számmaradék` DCG szabály így is írható (az első a vágó nélküli, a másik vágót tartalmazó változattal azonos hatású):

```
% Leelemezhető jegyek egy esetleg üres listája.
számmaradék -->
        (   jegy, számmaradék.
        ;   []
        ).

% Leelemezhető jegyek egy maximális, esetleg üres listája.
számmaradék -->
        (   jegy -> számmaradék.
        ;   []
        ).
```

Nézzük most, hogy a terminális jelet is tartalmazó ⟨jegy⟩ nyelvtani szabályt hogyan alakíthatjuk át Prolog eljárássá!

```
% jegy(L0, L): L0-ból leelemezhető egy jegy kódja, marad L.
jegy([0'0|L], L).
jegy([0'1|L], L).
```

A terminálisok elemzése nagyon egyszerű, hiszen csak ellenőrizni kell, hogy a listában az adott elem jön, és ha igen, akkor a lista farkát kell kiadni maradékként.

A DCG nyelvtanok a terminálisokra is tartalmaznak jelölést: ezeket egy vagy többelemű listaként kell a DCG szabály jobboldalán szerepeltetni:

```
% Leelemezhető egy jegy kódja.
jegy --> [0'0].
jegy --> [0'1].
```

Ezekből a DCG szabályokból egy kicsit más kód generálódik, mint amit az előbb felírtunk:

```
jegy(A, B) :-
        'C'(A, 48, B).
jegy(A, B) :-
        'C'(A, 49, B).
```

Itt `'C'/3` egy beépített eljárás (azért van ilyen furcsa neve, mert közvetlen alkalmazása a felhasználó számára nem ajánlott), amelynek definíciója:

```
'C'([C|L], C, L).
```

Fejlesszük most tovább elemzőnket, hogy `jegy`-ként decimális számjegyeket is elfogadjon. Ezt a következőképpen tehetjük meg:

```
% leelemezhető egy jegy kódja.
jegy --> [J], {jegykód(J)}.

% J egy jegy kódja.
jegykód(J):-  J >= 0'0, J =< 0'9.
```

Az `jegy` új változatából a következő Prolog klóz keletkezik:

```
jegy(L0, L) :- 'C'(L0, J, L), jegykód(J).
```

Látjuk tehát, hogy a DCG szabályban, a terminálist jelölő listában egy változót is irhatunk, és ezzel mintegy lekérdezzük a soron következő terminálist (karakter-kódot). Egy tetszőleges Prolog hívást is beszúrhatunk az elemzési folyamatba, ha azt a szabályban kapcsos zárójelben szerepeltetjük. Ez a két lehetőség így együtt lehetővé teszi, hogy egy Prolog szabályban döntsük el, hogy elfogadjuk-e a következő terminális jelet.

A DCG szabályokat argumentumokkal is elláthatjuk, így a fenti számelemzőt kiegészíthetjük úgy, hogy siker esetén adja vissza a szám értékét. A `számmaradék` eljárás az általa leelemzett karakterkód-listát adja vissza a paraméterében.

```
% leelemezhető az Sz szám
szám(Sz) -->
        jegy(J), számmaradék(M),
        {number_codes(Sz, [J|M])}.

% leelemezhető jegyek egy esetleg üres JL listája.
számmaradék([J|JL]) -->
        jegy(J), !,
        számmaradék(JL).
számmaradék([]) --> [].

% J a leelemzett jegy kódja.
jegy(J) --> [J], {jegykód(J)}.
```

Ezek a paraméteres nem-terminálisok is éppúgy kiegészülnek egy akkumulátor argumentum-párral, mint a paraméter nélküliek. A DCG $\longrightarrow$ Prolog transzformáció mindig az argumentumlista végére teszi a kiegészítő argumentumokat. Például a fenti első szabály Prolog alakja a következő lesz:

```
szám(Sz, L0, L) :-
        jegy(J, L0, L1),
        számmaradék(M, L1, L),
        number_codes(Sz, [J|M]).
```

4.12.2. A DCG nyelvtani szabályok szerkezete — összefoglalás

- Nem-terminális: tetszőleges *hívható* kifejezés (atom vagy struktúra).
- Terminális: *tetszőleges* Prolog kifejezés; a 0, 1 vagy több terminális sorozata listaként helyezhető el a DCG szabályokban.
- Feltételek: tetszőleges Prolog hívás elhelyezhető `{}` zárójelekbe zárva.
- A DCG szabály alakja: Baloldal `-->` Jobboldal .
- Baloldal: egy nem-terminális, amit opcionálisan terminálisok listája követhet.

- Jobboldal: Egymás után írás (,), diszjunkció (;), ha-akkor és ha-akkor-egyébként (->) és negáció (\+) segítségével épül fel terminálisokból, nem-terminálisokból és Prolog feltételekből. A vágó (!) „automatikusan” Prolog hívássá alakul (mintha {!} volna), de pl. true nem.

Általános példa

```
p(A,...)  -->
      q0(B,...), ...[X], qi(C,...), ..., {Cel}, ...qn(D,...)
```

lefordított alakja:

```
p(A,...,L0,L):-
      q0(B,...,L0,L1), ...Li-1 = [X|Li], qi(C,...,Li,Li+1),...
      Cel, ..., qn(D,...,Ln,L)
```

4.12.3. További példák

P24 Példa: Szótárbeolvasás

Ez a példa egy szótárprogram beolvasó-elemző részét mutatja be.

A szotar_elem_be/1 eljárás beolvas és kielemez egy sort, amelyben szavak sorozata kell álljon. A szavakban csak kis- és nagybetűk fordulhatnak elő, elválasztásukra egy vagy több szóköz szolgál. A sorban legfeljebb egy egyenlőség- vagy minusz-jel is előfordulhat. Egy később megirandó szótár-programban (P32) ez utóbbi alak szolgál majd az egymásnak megfeleltetett kifejezések (szó-sorozatok) megadására, míg az olyan sorok, amelyekben nincs = vagy - jel, a megadott szósorozat lefordítását kérik majd.

A szotar_elem_be(Kif) eljáráshívás a következő sor belső alakját adja vissza Kif-ben, ez egy atomokból álló lista, vagy két ilyen lista a - operátorral összekötve. Ha az elemzés nem sikerül, az eljárás hibát jelez, és mindaddig új sort olvas, amíg egy helyes alakot nem talál.

A soron következő karakter beolvasására a get_code(Kód) eljárás szolgál, amely a karakter kódját egyesíti a Kód argumentummal.

```
% Kif a következő sor kielemzett formája
szotar_elem_be(Kif):-
        rd_line(L),
        (   szotar_elem(Kif, L, []) -> true
        ;   format('Hibas sor: ~s~n', [L]), szotar_elem_be(Kif)
        ).

% szotar_elem(Kif, L1, L): az L1 kódlistáról
% kielemezhető egy szótar-elem, a maradék kódlista L.
szotar_elem(Elem) -->
        szokoz, szosor(Szosor1),
        (   elvalaszto ->  szosor(Szosor2), {Elem = Szosor1-Szosor2}
        ;   {Elem = Szosor1}
        ).

% elvalaszto --> leelemezhető egy elvalasztó.
elvalaszto--> [0'-], szokoz ; [0'=], szokoz.

% szosor(SL) --> leelemezhető az SL szólista
szosor([S|SL]) -->
        szo(S), szokoz,
        (   szosor(SL)
        ;   {SL = []}
        ).
```

```
% leelemezhető az S szó
szo(S) --> betu(B), szomaradek(BL), {atom_codes(S, [B|BL])}.

% szomaradek(BL) --> leelemezhető betűk egy esetleg
% üres listája, BL a leelemzett betűk kódlistája.
szomaradek([B|BL]) -->
        betu(B), !, szomaradek(BL).
szomaradek([]) --> [].

% B a leelemzett betű kódja.
betu(B) --> [B], {betukod(B)}.

% B egy betű kódja.
betukod(B):- B >= 0'a, B =< 0'z ; B >= 0'A, B =< 0'Z.

% leelemezhető szóközök esetleg üres sorozata
szokoz --> [0' ], !, szokoz ; [].

% L a következő sor karakterkódjainak listája.
rd_line(L):-
        get_code(C), rd_line(C, L).

% rd_line(C, L): L a következő sor karakterkódjainak
% listája, ha C a soronkövetkező karakter kódja.
rd_line(10, []):- !.
rd_line(C, [C|L]):-
        get_code(C1), rd_line(C1, L).

| ?- szotar_elem_be(K).
|: six = hat.
Hibas sor: six = hat.
|: six = hat

K = [six]-[hat] ? ;

no
| ?- szotar_elem_be(K).
|: Prolog clause = Prolog kloz

K = ['Prolog',clause]-['Prolog',kloz] ? ;

no
```

P25 Példa: Kifejezés kiértékelése

```
% kif(Z, L0, L): L0 kódlistából leelemezhető egy
% Z értékű aritmetikai kifejezés, marad L
kif(Z) --> tag(X), "+", kif(Y), {Z is X + Y}.
kif(Z) --> tag(X), "-", kif(Y), {Z is X - Y}.
kif(X) --> tag(X).

% tag(Z, L0, L): L0-ból leelemezhető egy Z értékű tag.
tag(Z) --> szám(X), "*", tag(Y), {Z is X * Y}.
tag(Z) --> szám(X), "/", tag(Y), {Z is X / Y}.
```

```
tag(Z) --> szám(Z).

| ?- kif(Z, "10*10-6*6", "").
Z = 64 ? ;
no
| ?- kif(Z, "10*10-6*6", L).
L = [], Z = 64 ? ;
L = [42,54], Z = 94 ? ;
L = [45,54,42,54], Z = 100 ? ;
L = [42,49,48,45,54,42,54], Z = 10 ? ;
no
```

Vegyük észre, hogy a fenti program „mohón” elemez, tehát először a leghosszabb kielemezhető kifejezést értékét adja vissza, de visszalépéskor hajlandó ennek részkifejezésként értelmezhető kezdőszeleteit is elfogadni. Ha a kifejezés után egy lezáró karaktert is elvárunk, akkor ezzel kizárhatjuk a részkifejezések elfogadását.

Vegyük észre, hogy a program az azonos prioritású műveleteket jobbról balra zárójelezi:

```
| ?- kif(Z, "4-2+1", []).
Z = 1 ?
```

Egy lehetséges javítás a kifejezés-maradék, tag-maradék stb. nyelvtani fogalmak bevezetése. Az alábbi részlet a kifejezés-elemzést mutatja be, a tagok elemzését végző szabályok analóg módon írhatók meg.

```
kif(Z) --> tag(X), kifmaradék(X, Z).

kifmaradék(X0, Z) -->
    "+", tag(X1), {X is X0 + X1}, kifmaradék(X, Z).
kifmaradék(X0, Z) -->
    "-", tag(X1), {X is X0 + X1}, kifmaradék(X, Z).
kifmaradék(X, X) --> [].
```

P26 Példa: “természetes” nyelvű beszélgetés

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

% mondat(Alany, Áll, L0, L): L0-L kielemezhető egy Alany alanyból
% és Áll állítmányból álló mondattá.
% Alany lehet első vagy második személyű névmás, vagy egyetlen szóból
% álló (harmadik személyű) alany.
mondat(én, Áll) --> perm(én, Áll).
mondat(te, Áll) --> perm(te, Áll).
mondat(Alany, Áll) --> szó(Alany), szavak(Áll).

% perm(Ki, Áll, L0, L): L0-L kielemezhető egy Ki (első vagy második személyű)
% névmásból és Áll állítmányból álló mondattá.
perm(Ki, Áll) --> névmás(Ki), létige(Ki), szavak(Áll).
perm(Ki, Áll) --> névmás(Ki), szavak(Áll), létige(Ki).
perm(Ki, Áll) --> szavak(Áll), létige(Ki), névmás(Ki).
perm(Ki, Áll) --> szavak(Áll), létige(Ki).

% névmás(Ki, L0, L): L0-L egy Ki névmás.
névmás(én) --> "én", köz.        névmás(te) --> "te", köz.
névmás(én) --> "Én", köz.        névmás(te) --> "Te", köz.
```

```
% létige(Ki, L0, L): L0-L egy Ki névmással egyeztetett létige.
létige(én) --> "vagyok", köz.    létige(te) --> "vagy", köz.

% köz(L0, L): L0-L nulla, egy vagy több szóköz.
köz --> " " -> köz ; [].

% szó(Sz, L0, L): L0-L egy Sz betűsorozatból álló (nem üres) szó.
% Betűnek számít minden a ".?" jelektől különböző karakter.
szó([B|Sz]) --> betű(B), szómaradék(Sz), köz.

% szómaradék(Sz, L0, L): L0-L egy Sz betűsorozatból álló (esetleg üres) szó.
szómaradék([B|Sz]) --> betű(B), !, szómaradék(Sz).
szómaradék([]) --> [].

% betű(K, L0, L): L0-L egy K kódú betű.
betű(K) --> [K], {non_member(K, " .?")}.

% szavak(SzL, L0, L): L0-L egy SzL szó-lista.
szavak([Sz|Szk]) --> szó(Sz),
     (   szavak(Szk)
     ;   {Szk = []}
     ).

% menet(Mondás, L0, L): Az L0-L kielemzett alakja Mondás.
menet(kijelent(Alany,Áll)) -->
    mondat(Alany, Áll), ".".
menet(kérdez(Alany)) -->
    mondat(Alany, [Szó]), "?", {kérdő(Szó)}.
menet(un) --> "Unlak.".

% kérdő(Szó): Szó egy kérdőszó.
kérdő("mi").   kérdő("ki").   kérdő("kicsoda").
kérdő("Mi").   kérdő("Ki").   kérdő("Kicsoda").

% Egy párbeszédet valósít meg.
párbeszéd :-
     repeat, rd_line(L),
     (   menet(M, L, []) -> feldolgoz(M)
     ;   write('Nem értem\n'), fail
     ), M = un, !.

% feldolgoz(Mondás): feldolgozza a felhasználótól érkező Mondás üzenetet.
feldolgoz(un) :- write('Én is.\n').
feldolgoz(kijelent(Alany, Áll)) :-
     assertz(tudom(Alany,Áll)), write('Felfogtam.\n').
feldolgoz(kérdez(Alany)) :-
     tudom(Alany, _), !, válasz(Alany).
feldolgoz(kérdez(_)) :-
     write('Nem tudom.\n').

% Felsorolja az Alany ismert tulajdonságait.
válasz(Alany) :- tudom(Alany, Áll),
     (   member(Szó, Áll), format('~s ', [Szó]), fail
     ;   nl, fail
```

```
    ).
válasz(_).
```

A fenti programban hivatkozott `rd_line/1` definícióját lásd a P24 példában.

Beszélgetős DCG példa — egy párbeszéd

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

4.12.4. DCG használata elemzésen kívül

A DCG formalizmust kényelmesen használhatjuk akkumulálást végző eljárások tömörebb írására. Listák akkumulálása esetén az elemi akkumulálási lépést a `[X]` DCG terminális jelölésként írhatjuk le.

Például a klasszikus `append` eljárást a következőképpen definiálhatjuk DCG szabályok segítségével:

```
append(L1, L2, L12) :-
        app(L1, L12, L2).

app([]) --> [].
app([X|L]) --> [X], app(L).

| ?- listing(app).
app([], A, B) :-
        B=A.
app([A|B], C, D) :-
        'C'(C, A, E),
        app(B, E, D).

| ?- append([alma,korte], [szilva], L).
L = [alma,korte,szilva] ? ;
no
```

Az alábbi példa egy listaszűrő eljárást DCG megvalósítását mutatja be.

```
% M az L lista N-nél nagyobb elemeinek listája.
nagyobb(L, N, M) :- nagyobb(L, N, M, []).

% nagyobb(L, N, M, M0): M-M0 az L lista N-nél
% nagyobb elemeinek listája.
```

```
nagyobb([], _) --> [].
nagyobb([X|L], N) --> {X > N}, !, [X], nagyobb(L, N).
nagyobb([_|L], N) --> nagyobb(L, N).
```

A DCG szabályokat használhatjuk nem csak listák, hanem tetszőleges kifejezések akkumulálására is, de ez esetben az elemi akkumulálási lépést a DCG-n kívül kell megírni:

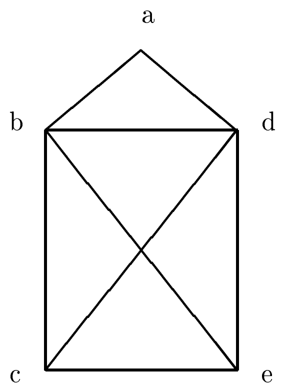
```
% Az L lista összege S.
sum(L, S) :- sum1(L, 0, S).

% sum1(L, S0, S): Az L lista összege S-S0.
sum1([]) --> [].
sum1([X|L]) --> add(X), sum1(L).

% X+S0= S.
add(X, S0, S) :- S is S0+X.
```

4.13. Nagyobb példák

Ebben az alfejezetben két közepes méretű Prolog példát ismertetünk: egy közismert feladványtípust megoldó programot, és egy rendezett bináris fákat kezelő eljárásgyűjteményt.

P27 Példa: Ábrarajzoló

Írjunk Prolog programot amely egy összefüggő vonallal megrajzolja pl. az alábbi ábrát:



Először el kell határoznunk milyen formában ábrázoljuk a program bemenetét, a gráfot és a várt eredményt, a vonalat:

megrajzolandó gráf:	élek listája
él:	egyik végpont - másik végpont a fenti gráf leírása pl. lehet: `[a-b,a-d,b-c,b-d,b-e,c-d,c-e,d-e]`
összefüggő vonal:	olyan gráfleírás (éllista), ahol a minden él végpontja a következő él kezdőpontjával azonos. pl.: `[c-b,b-a,a-d,d-b,b-e,e-c,c-d,d-e]`

Döntésünknek a következő típus-specifikációk felelnek meg:

```
% :- type  graf == list(el).
% :- type    el ---> atom-atom.
% :- type vonal == graf.
```

Típusfogalmunk nem alkalmas a vonalra vonatkozó feltétel leírására, ennek ellenére érdemes a `graf` és a `vonal` típusokat megkülönböztetni.

A feladvány-megoldó program első változata a fogalmak definícióit próbálja követni, a hatékonysággal nem törődve.

```
pelda_graf([a-b,a-d,b-c,b-d,b-e,c-d,c-e,d-e]).


% G gráfot a V vonal megrajzolja
% :- pred megrajzolja_1(graf::in, vonal::out).
megrajzolja_1(G, V):-
      azonos_graf(G, V), osszefuggo(V).

% osszefuggo(V) = a V vonal összefüggő
% :- pred osszefuggo(graf::in).
osszefuggo([_]).
osszefuggo([E|V]):-
      csatlakozik(E, V),
      osszefuggo(V).

% csatlakozik(E, V) = Az E él végpontja azonos a V vonal
% kezdőpontjával.
% :- pred csatlakozik(el::in, graf::in).
csatlakozik(_-P, [P-_|_]).

% azonos_graf(G, P) = A G gráffal azonos a P gráf (az élek
% sorrendje lehet más és az élek végpontjai felcserélődhetnek)
% :- pred azonos_graf(graf::in, graf::out).
azonos_graf([], []).
azonos_graf(G, [E|P]):-
      select(X, G, G1),
      azonos_el(X, E),
      azonos_graf(G1, P).

% azonos_el(El1, El2) = El1 és El2 azonos éleket jelölnek,
% de a végpontok fel lehetnek cserélve.
% :- pred azonos_el(el::in, el::out).
azonos_el(E, E).
azonos_el(P-Q, Q-P).
```

Ez egy ún. generál-és-ellenőriz (generate and test) típusú megoldás: az `azonos_graf(G, V)` eljárás `V`-ben felsorolja a `G` gráf összes lehetséges megadási módját, lényegében az élek sorrendjének és irányításának összes permutációját végigpróbálva. Minden egyes `V` gráfra ezután az `osszefuggo(V)` eljárással ellenőrizzük, hogy az „véletlenül" egy összefüggő vonal-e. Ez kisméretű adatokra működik, de már a fenti házikó gráfra gyakorlatilag kivárhatatlan ideig fut:

```
| ?- megrajzolja_1([a-b,c-d,d-a],V).

V = [b-a,a-d,d-c] ? ;

V = [c-d,d-a,a-b] ? ;

no
| ?- pelda_graf(G), megrajzolja_1(G,V).
^C
```

```
Prolog interruption (h for help)? a
{Execution aborted}
```

Tekintsünk most egy második megoldást, ebben lényegében a permutálás és ellenőrzés vegyítve fut le:

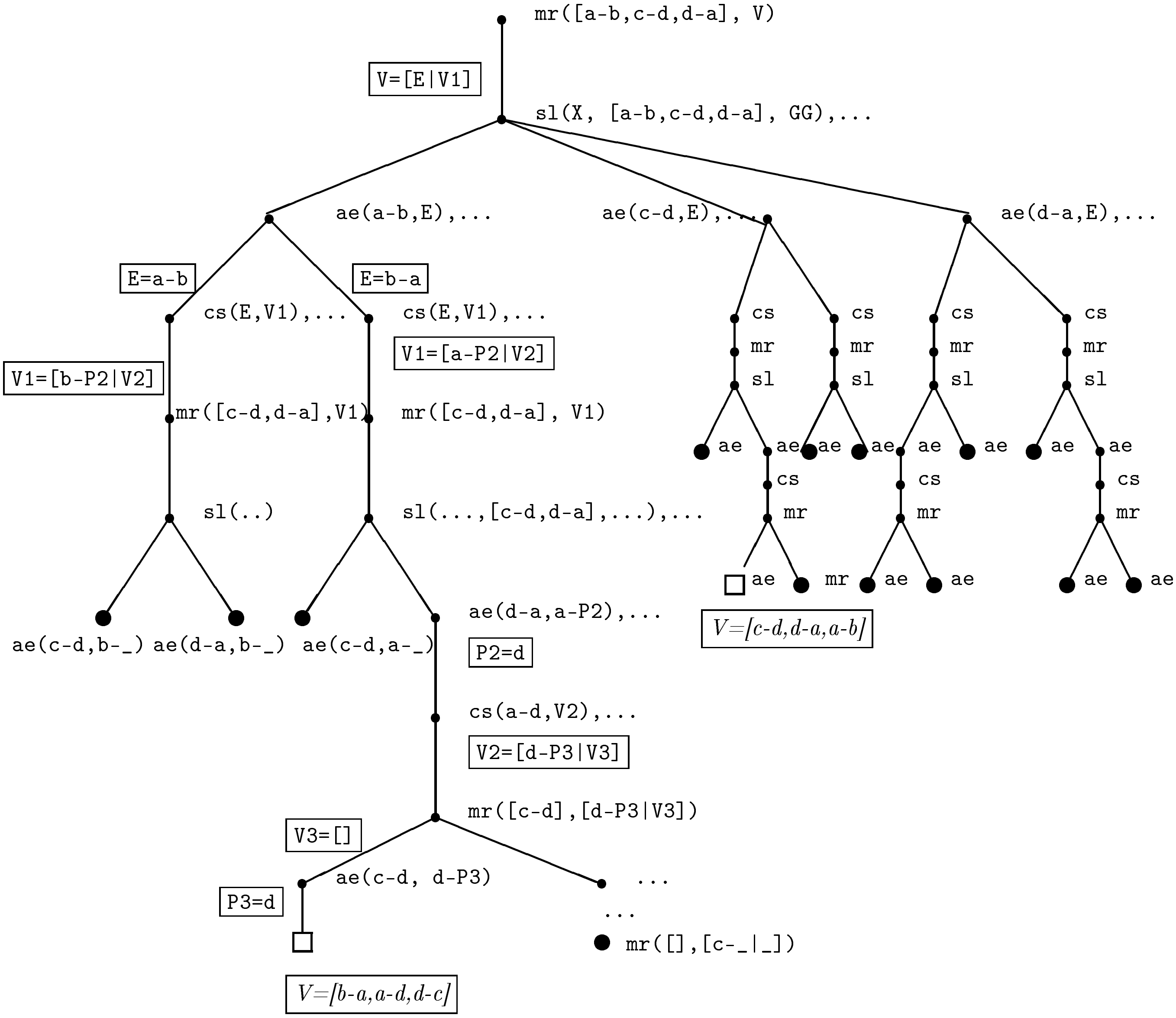
```
% :- pred megrajzolja_2(graf::in, vonal::out).
megrajzolja_2([X],[E]):- !,
      azonos_el(X, E).
megrajzolja_2(G, [E|V]):-
      select(X, G, G1),
      azonos_el(X, E),
      csatlakozik(E, V),
      megrajzolja_2(G1, V).
```

Ez a változat lényegében azonos szerkezetű, mint a 3.4.7. pontban ismertetett `útvonal_3` eljárás, csak itt nincs megadva a keresett út hossza, viszont kikötés, hogy a gráf minden élét fel kell használni. Az első klózban levő vágó egy zöld vágó, hiszen a második klóz az egyelemű `G` listára meghiúsul.

A `megrajzolja_2` eljárás futtatásakor gyakorlatilag azonnal megkapjuk a házikó feladat négy megoldását:

```
| ?- pelda_graf(G), megrajzolja_2(G,V).

G = [a-b,a-d,b-c,b-d,b-e,c-d,c-e,d-e],
V = [c-b,b-a,a-d,d-b,b-e,e-c,c-d,d-e] ? ;

G = [a-b,a-d,b-c,b-d,b-e,c-d,c-e,d-e],
V = [c-b,b-a,a-d,d-b,b-e,e-d,d-c,c-e] ? ;

G = [a-b,a-d,b-c,b-d,b-e,c-d,c-e,d-e],
V = [c-b,b-a,a-d,d-c,c-e,e-b,b-d,d-e] ? ;

G = [a-b,a-d,b-c,b-d,b-e,c-d,c-e,d-e],
V = [c-b,b-a,a-d,d-c,c-e,e-d,d-b,b-e] ?

yes
| ?-
```

Az alábbiakban az ábrarajzoló második változatának keresési fáját mutatjuk be, egy egyszerű bemenő adatra.



Jelmagyarázat

mr	megrajzolja_2	▪	sikeres futásvég
ae	azonos_el	·	zsákutca
cs	csatlakozik	$\boxed{\texttt{X=...}}$	közbenső behelyettesítés
sl	select	$\boxed{X=\ldots}$	végső behelyettesítés

Második nagyobb példaként rendezett bináris fák építésére és bejárására mutatunk Prolog programokat.

P28 Példa: Rendezett bináris fák kezelése

A kezelni kívánt fastruktúrák szerkezete a következő:

```
% :- type bfa ---> ures ; bfa(int, bfa, bfa).
```

Egy számnak, ill. egy számlistának egy bináris fába való beszúrását végzi a következő két eljárás.

```
% beszur(BF0, E, BF): E beszurása BF0 rendezett fába
% a BF rendezett fát adja
% :- pred beszur(bfa::in, int::in, bfa::out).
beszur(ures, Elem, bfa(Elem, ures, ures)).
beszur(bfa(E, B, J), Elem, Fa):-
      Elem < E, !,
      Fa = bfa(E, B1, J),
      beszur(B, Elem, B1).
beszur(bfa(E, B, J), Elem, bfa(E, B, J1)):-
      beszur(J, Elem, J1).

% lista_bfa(L, BF0, BF): L elemeit beszurva BF0-ba
% kapjuk BF-t.
% :- pred lista_bfa(list(int)::in, bfa::out).
lista_bfa([], BF, BF).
lista_bfa([E|L], BF0, BF):-
      beszur(BF0, E, BF1),
      lista_bfa(L, BF1, BF).
```

Egy rendezett bináris fa listává alakítására szolgáló eljárás következik:

```
% bfa_lista(BF, L0, L): BF elemeit L0 elé fűzve
% kapjuk az L listát.
% :- pred bfa_lista(bfa::in, list(int)::out).
bfa_lista(ures, L, L).
bfa_lista(bfa(E, B, J), L0, L):-
      bfa_lista(J, L0, L1),
      bfa_lista(B, [E|L1], L).
```

Végül a fenti eljárásokra alapozva egy lista-rendező eljárást is bemutatunk.

```
% L lista rendezettje R.
% :- pred rendez(list(int)::in, list(int)::out).
rendez(L, R):-
      lista_bfa(L, ures, BF), bfa_lista(BF, [], R).
```

4.14. Gyakorló feladatok

GY4.

Írjunk Prolog programot nem-negatív számok legnagyobb közös osztójának kiszámítására, az Euklideszi algoritmus segítségével.

Egy `A` szám `B`-vel való osztásakor képződő `M` maradék kiszámítására az `M is A mod B` aritmetikai eljárást használhatjuk.

GY5.

Írjunk egy

```
egyszerusitheto(X,Y)
```

Prolog eljárást amely felsorolja a következő tulajdonságokkal bíró `(X,Y)` számpárokat:

- `X` és `Y` tizes számrendszerben kétjegyű természetes számok

- `X` tizes számrendszerbeli alakja AB
- `Y` tizes számrendszerbeli alakja BC
- Az `X/Y` és A/C törtek (végtelen pontossággal) megegyeznek
- A, B és C páronként különböző számjegyek

(azaz az `X/Y` $= AB/BC$ tört egyszerűsíthető a B számjegyek elhagyásával).
(Forrás: 1. Prolog programozási verseny, 1994 november, Ithaca, NY)
Az `X =\= Y` beépített eljárást használhatjuk annak eldöntésére, hogy `X` és `Y` különbözőek-e.

GY6.

Az $f(i)$ Fibonacci-szerű számsorozatot a következő rekurzív módon definiáljuk:

$$f(1) = 1, f(2) = 1, f(i) = f(i-1) + 3 * f(i-2)$$

Írjunk egy `tag(I, FI)` Prolog eljárást, amely a sorozat `I`-edik tagját `FI`-ben előállítja. Pl. a `tag(3,F)` hívás eredménye `F=4` lesz.
Kiséreljünk meg olyan definíciót írni a fenti `tag(I, FI)` eljárásra, amelynek futási ideje `I`-vel arányos (lineáris `I`-ben).

GY7.

Adottnak feltételezzük a következő eljárást:

```
szuloje(Gyermek, Szulo).
```

Erre építve definiáljuk a következő eljárásokat (ehhez természetesen segédeljárásokat is definiálhatunk):

(a) `unokatestvere(A, B)` (`A` és `B` szülei testvérek)
(b) `n_ed_unokatestvere(N, A, B)`
(1. unokatestvérek = unokatestvérek
2. unokatestvérek = unokatestvérek gyermekei
stb.)
`N` adott szám, `A` és `B` változók is lehetnek.

GY8.

Tegyük fel, hogy egy súlyozott élű irányítatlan gráfot a Prolog adatbázisában tárolt következő alakú tényállításokkal adunk meg:

```
el(Honnan, Hova, Koltseg)
```

Egy ilyen állítás azt fejezi ki, hogy a gráf `Honnan` csúcsából vezet él a `Hova` csúcsba, és ennek az élnek `Koltseg` a költsége, ahol `Koltseg` egy szám. A `Honnan` és `Hova` csúcsok sorrendje nem lényeges (irányitatlan a gráf), de egy adott csúcspár csak egyszer szerepel az '`el`' relációban.
Írjunk egy `kormentes_ut(Honnan, Hova, Koltseg)` Prolog eljárást, amely azt fejezi ki, hogy a `Honnan` csúcsból el lehet a gráfban jutni a `Hova` csúcsba egy önmagát nem érintő útvonalon, amelynek összköltsége `Koltseg`.

GY9.

Írjunk egy

```
nszerese(N, L, NL)
```

Prolog eljárást, amely az `L` lista `N`-szeres egymás után füzését egyesíti `NL`-lel. (Pl.:

```
?- nszerese(3, [a,b], [a,b,a,b,a,b])
```

sikerül).
Kiséreljünk meg hatékony, jobb-rekuzív megoldást adni.

GY10.

Írjunk egy olyan `atrendez(L, L1)` Prolog eljárást, amely adott `L` (egész számokból álló) lista esetén `L1`-ben előállítja `L` átrendezett formáját a következő értelemben véve: `L1`-ben az összes páros szám megelőzi a páratlanokat, a páratlan és a páros részben a számok sorrendje megegyezik az eredetivel. Pl.:

```
atrendez([1,2,3,4,5,6], L1).
```

eredménye:

```
L1 = [2,4,6,1,3,5]
```

5. fejezet

A legfontosabb beépített eljárások

Ebben a fejezetben a legfontosabb beépített eljárásokat írjuk le. Alapvetően az ISO Prolog szabvány szerint ismertetjük ezeket, de kitérünk a SICStus Prologbeli kiterjesztésekre, illetve az ISO és a hagyományos SICStus üzemmód közötti különbségekre.[1]

5.1. A predikátumleírások formátuma

Egy predikátum leírása az eljárás nevével kezdődik. Ezt a lehetséges hívási minták követik az esetleges argumentumok típusával, majd egy rövid magyarázat következik arról, hogy mi az adott predikátum jelentése és mi a hatása. Ezután következnek a megjegyzések. A leírást az eljárás ISO szabvánnyal való kapcsolata zárja.

A mellékhatásos eljárásoknál a jelentés gyakran „Igaz.", azaz mindig sikerülnek (vagy hibát jeleznek). Ilyenkor a jelentést általában nem írjuk ki.

A hívási mintáknál a változó elé illesztett egy vagy két karakter az adott arguentum felhasználási módját jelzi. A következő karakterek használatosak:

`@` Az argumentum egy tisztán bemenő paraméter. A hívásban megadott paraméterben előforduló behelyettesítetlen változók nem kapnak értéket, kivéve, ha egy másik, nem tisztán bemenő argumentumban is előfordulnak (és ott értéket kapnak).

`+` Az argumentum egy bemenő paraméter. A paraméterben előforduló behelyettesítetlen változók értéket kaphatnak. Ha az argumentum egy atomi kifejezés, akkor nincs különbség a `@` és a `+` módok között, ezért a `@` módot csak akkor használjuk, ha az argumentum lehet összetett kifejezés.

`?` Az argumentum egy kimenő/bemenő paraméter. A eljárás végrehajtása során a paramétert egyesítik valamivel. Ha ez az egyesítés nem sikerül, akkor az eljárás meghiúsul vagy hibát jelez, ha argumentum be van helyettesítve és nem megfelelő típusú. A paramétertben előforduló behelyettesítetlen változók értéket kaphatnak.

`-` Az argumentum egy tisztán kimenő paraméter. Híváskor az értéke csak behelyettesítetlen változó lehet. Ha az eljárás sikerül, a változó értéket kap.

`:`X ahol X a fenti karakterek közül való. A predikátum egy meta-predikátum és az adott argumentum modulkiterjesztésen esik át (vesd össze modularitás, 6.1).

Ha a leírás több mint egy hívási mintát tartalmaz, akkor egy adott hívásra a legfelső olyan minta vonatkozik rá, amely kielégíti a feltételeket.

[1] A két üzemmód közötti váltásra a `set_prolog_flag/2` beépített eljárás használható, lásd 5.14.

5.2. Vezérlési eljárások

`!/0` — vágó

Hívási minta:

`!`

Jelentés:

Igaz.

Hatás:

Megszünteti az összes választási pontot egészen a szülő célig, azt is beleértve. ∎

A vágó eljárást részletesen ismertettük a 4.1.1 alfejezetben.

`call/1`

Hívási minta:

`call(:+Cél)`

Argumentumok:

`Cél` Az argumentum egy meghívható kifejezés.

Jelentés:

`call(X)` igaz, ha `X` igaz.

Hatás:

A rendszer `X`-et célként meghívja és annak sikere dönti el az eljárás sikerességét.

Megjegyzések:

Ez egy ún. meta-hívás, adatból programelem válik. Végrehajtáskor `X` már nem lehet üres változó, csak név vagy struktúra, amit célként értelmezünk. Vezérlési operátorok is (például `,` és `;`) előfordulhatnak benne. Az `X`-ben szereplő vágók hatása nem terjed túl a `call(X)` híváson. Más szavakkal a `call/1` eljárás nem átlátszó a vágó számára.

Ha egy változót célként szerepeltetünk, akkor a rendszer azt úgy kezeli, mintha egy `call/1` hívás argumentuma volna. ∎

Példa:

```
ketszer(Hivas) :-
      call(Hivas), call(Hivas).

| ?- ketszer(nl).
```

`fail/0`

Hívási minta:

`fail`

Jelentés:

Hamis. ∎

Példa:

```
p :- between(1, 5, X), write(X), fail.
```

`true/0`

Hívási minta:

`true`

Jelentés:

Igaz. ■

Példa:

```
p :-
        (   between(1, 5, X), write(X), fail
        ;   true
        ).
```

(',')/2 — konjunkció

Hívási minta:

`Első, Második`

Argumentumok:

`Első` Az argumentum egy meghívható kifejezés.
`Második` Az argumentum egy meghívható kifejezés.

Jelentés:

Igaz, ha `Első` és `Második` is igaz.

Hatás:

A rendszer először célként meghívja `Első`-t és ha sikerült, akkor célként meghívja `Második`-at.

Megjegyzések:

Ez egy vezérlési szerkezet. `(',')/2` átlátszó a `!/0` (vágó) számára. ■

(;)/2 — diszjunkció

Hívási minta:

`(Első; Második)`

Argumentumok:

`Első` Az argumentum egy meghívható kifejezés.
`Második` Az argumentum egy meghívható kifejezés.

Jelentés:

Igaz, ha `Első` vagy `Második` igaz.

Hatás:

A rendszer először célként meghívja `Első`-t, majd annak meghiúsulása esetén (visszalépéskor) meghívja `Második`-at.

Megjegyzések:

Ez egy vezérlési szerkezet. `(;)/2` átlátszó a `!/0` (vágó) számára. ■

(->)/2 — if–then

Hívási minta:

`(Ha -> Akkor)`

Argumentumok:

`Ha` Az argumentum egy meghívható kifejezés.
`Akkor` Az argumentum egy meghívható kifejezés.

Jelentés:

Igaz, ha `Ha` igaz és `Akkor` igaz `Ha` első megoldására.

Hatás:

A rendszer megkeresi a `Ha` első megoldását. Ha ez sikerül, meghívja `Akkor`-t.

Megjegyzések:

Ez egy vezérlési szerkezet. A rendszer `Ha`-nak csak az első megoldását keresi meg (nem lép vissza bele). A `Ha` rész nem átlátszó, az `Akkor` rész átlátszó a `!/0` (vágó) számára.

Kompatibilitás:

SICStus Prologban a `Ha` részben nem szerepelhet vágó. ■

`(;)/2` — if–then–else

Hívási minta:

`(Ha -> Akkor; Egyébként)`

Argumentumok:

- `Ha` Az argumentum egy meghívható kifejezés.
- `Akkor` Az argumentum egy meghívható kifejezés.
- `Egyébként` Az argumentum egy meghívható kifejezés.

Jelentés:

Igaz, ha `Ha` igaz és `Akkor` igaz `Ha` első megoldására, vagy `Ha` hamis és `Egyébként` igaz.

Hatás:

A rendszer először célként meghívja `Ha`-t. Ha az sikerül, akkor meghívja `Akkor`-t, egyébként meghívja `Egyébként`-et.

Megjegyzések:

Ez egy vezérlési szerkezet, amit az különbözteti meg a diszjunkciótól, hogy az első argumentuma egy `(->)/2` funktorú kifejezés.

A rendszer `Ha`-nak csak az első megoldását keresi meg (nem lép vissza bele). A `Ha` rész nem átlátszó, a másik két rész átlátszó a `!/0` (vágó) számára.

Kompatibilitás:

SICStus Prologban a `Ha` részben nem szerepelhet vágó. ■

A feltételes kifejezés általános formája:

```
( If1 -> Then1
; If2 -> Then2
...
; Else
)
```

pszeudo-definíciója:

```
If -> Then ; _Else :- If, !, Then.
_If -> _Then ; Else :- Else.
```

Ez az eljárás két okból nem lehet valódi definíció: egyrészt mert így a `Then` és az `Else` részekben szereplő vágó hatása lokális lenne, másrészt mert `(;)/2` nem definiálható felül.

Az if–then–else szerkezet általában kiváltható egy közönséges vágót tartalmazó segéd-definícióval.

Példa:

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

ugyanaz, mint

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

`(\+)/1` — nem bizonyítható

Hívási minta:

`\+ :+Cél`

Argumentumok:

`Cél` Az argumentum egy meghívható kifejezés.

Jelentés:

`\+ Cél` igaz, ha Cél nem igaz.

Hatás:

Meghívja `Cél`-t és ha az meghiúsul, akkor sikerül, egyébként meghiúsul.

Megjegyzések:

A végrehajtási módszerből adódóan siker esetén sem helyettesít be változókat. ■

Példa

```
member(X, [2,4,6]), \+ member(X, [1,2,3]).

X = 4;
X = 6;
no
```

de

```
\+ member(X, [1,2,3]), member(X, [2,4,6])
```

meghiúsul!
Definíciója:

```
\+ X :- X, !, fail.
\+ X.
```

azaz pl. `\+ member(X, L)` ekvivalens az alábbi `notmember(X, L)` meghívásával:

```
notmember(X,L) :- member(X,L), !, fail.
notmember(_,_).
```

A `\+` nem valódi negáció: `notmember(X,[1,2])` nem sorolja fel az összes `1`-től és `2`-től különböző kifejezést, hanem meghiúsul, mert `X` behelyettesíthető úgy, hogy az `[1,2]` lista elemévé váljék.

Használata:

A fent ismertetett okok miatt a `(\+)/1` hívást többnyire csak változómentes célok esetén szokás használni.

`repeat/0`

Hívási minta:

`repeat`

Jelentés:

Igaz.

Hatás:
Híváskor választási pontot hoz létre, majd sikerül. Ilyen módon visszalépések során végtelen sokszor képes sikerülni. ∎

Ezt az eljárást mindig vágóval párban használjuk, pl.:

```
fociklus:-
      repeat,
            read(X),
            feldolgoz(X),
      X = end_of_file, !.
```

A `repeat` értelmesen csak mellékhatásos környezetben használható.

5.3. Dinamikus adatbáziskezelés

Az ebben a szakaszban definiált beépített eljárások az ún. dinamikus eljárások módosítását teszik lehetővé. Dinamikus eljárásokhoz (szemben a statikusokkal) futási időben hozzáadhatunk, és el is vehetünk tőlük klózokat. A dinamikus eljárások általában lassaban futnak mint a megfelelő statikus eljárások.

A program módosítása nyilvánvalóan mellékhatásos, ezért kerülendő.

Egy eljárás alapértelmezés szerint statikus. Csak azok az eljárások lesznek dinamikusak, amelyket a

```
:- dynamic(Eljárásnév/Argumentumszám)
```

deklarációval látunk el, vagy a rendszerbe először a most következő eljárások egyikével kerülnek be.

A program átláthatósága érdekében célszerű az általunk használt dinamikus eljárásokra a megfelelő `dynamic` deklarációt a program szövegében szerepeltetni.

Az alábbi eljárásokban egy `F` tényállítás egy `F :- true` alakú klóznak számít.

`asserta/1`

Hívási minta:

`asserta(:@Klóz)`

Argumentumok:

`Klóz` Az argumentum egy klózként értelmezhető kifejezés.

Hatás:
A `Klóz` kifejezést klózként értelmezve felveszi a programba, mégpedig az adott predikátum első klózaként. ∎

`assertz/1`

Hívási minta:

`assertz(:@Klóz)`

Argumentumok:

`Klóz` Az argumentum egy klózként értelmezhető kifejezés.

Hatás:
A `Klóz` kifejezést klózként értelmezve felveszi a programba, mégpedig az adott predikátum utolsó klózaként. ∎

`retract/1`

Hívási minta:

`retract(:@Klóz)`

Argumentumok:

`Klóz` Az argumentum egy klózként értelmezhető kifejezés.

Jelentés:

A `retract(Klóz)` hívás igaz, ha létezik egy dinamikus eljárás, amelynek van `Klóz`-zal egyesíthető klóza.

Hatás:

Illeszti `Klóz`-zal az első megfelelő klózt az adott definícióból majd kitörli a klózt. *Többszörösen sikerülhet!* Visszalépéskor újabb klózt keres, illeszti, majd kitörli stb. ■

`abolish/1`

Hívási minta:

`abolish(@Funktor)`

Argumentumok:

`Funktor` Az argumentum egy Név/Aritás alakú kifejezés, ahol Név egy atom, Aritás egy egész.

Hatás:

Az `abolish(N/A)` hívás kitörli az `N` nevű `A` argumentumszámú eljárás összes klózát. ■

`clause/2`

Hívási minta:

`clause(:+Fej, ?Törzs)`

Argumentumok:

`Fej` Az argumentum meghívható kifejezés.
`Törzs` Az argumentum meghívható kifejezés.

Jelentés:

Igaz, ha létezik egy interpretált (F:-T) klóz, amely egyesíthető a (`Fej`:-`Törzs`) struktúrával.

Hatás:

A `(Fej :- Törzs)` klózzal illeszthető első klózt megkeresi és illeszti. *Többszörösen sikerülhet!* Visszalépéskor újabb klózt keres, illeszti stb. ■

`current_predicate/1`

Hívási minta:

`current_predicate(?Funktor)`

Argumentumok:

`Funktor` Az argumentum egy Név/Aritás alakú kifejezés, ahol Név egy atom, Aritás egy egész.

Jelentés:

Igaz, ha létezik egy F/A funktorú eljárás, és F/A egyesíthető `Funktor`-ral. Többszörösen sikerülhet. ■

5.4. Aritmetika

5.4.1. Prolog kifejezések mint aritmetikai kifejezések kiértékelése

Kétféle aritmetikai kifejezés létezik, egyszerű és összetett. Egyszerű aritmetikai kifejezések az egész és a lebegőpontos számok. Összetett aritmetikai kifejezések csak összetett Prolog kifejezések lehetnek. Az érvényes összetett aritmetikai kifejezésekkel a 5.4.2 szakasz foglalkozik.

A kifejezések kiértékelése a következőképpen történik:

1. Ha a kifejezés szám, akkor az érték önmaga.

2. Ha a kifejezés összetett, akkor az értéket úgy képezzük, hogy a kifejezés argumentumait (implementációfüggő sorrendben) kiértékeljük, majd kifejezés funktorának és az argumentumok típusának megfelelő műveletet elvégezzük rajtuk.

Ha egy argumentum név vagy változó, a rendszer hibát jelez.

5.4.2. Összetett aritmetikai kifejezések

Ebben a szakaszban azt adjuk meg, hogy milyen funktorok használhatók aritmetikai kifejezésekben.

Infix operátorok

`+`	összeadás	`mod`	modulus képzés
`-`	kivonás	`rem`	maradék képzés
`*`	szorzás	`<<`	bitenkénti balra léptetés
`/`	osztás	`>>`	bitenkénti jobbra léptetés
`**`	hatványozás	`/\`	bitenkénti és
`//`	egész osztás	`\/`	bitenkénti vagy

Prefix operátorok

`-`	negáció
`\`	bitenkénti negáció

Függvény jelölésűek

`abs/1`	abszolút érték	`ceiling/1`	felső egészrész
`sign/1`	előjel függvény	`sin/1`	szinusz
`float_integer_part/1`	lebegőpontos egészrész	`cos/1`	koszinusz
`float_fractional_part/1`	lebegőpontos törtrész	`tan/1`	tangens[2]
`float/1`	lebegőpontos konverzió	`atan/1`	arkusz tangens
`floor/1`	alsó egészrész	`exp/1`	exponenciális függvény
`truncate/1`	csonkolás	`log/1`	természetes alapú logaritmus
`round/1`	kerekítés	`sqrt/1`	négyzetgyök
`max/2`	maximum[2]	`min/2`	minimum[2]

5.4.3. Aritmetikai eljárások

Ez a szakasz felsorolja azokat a beépített eljárásokat, amelyek valamelyik argumentumukat aritmetikai kifejezésként kiértékelik.

`(is)/2`

Hívási minta:

`?X is @AKif`

Argumentumok:

`X` Az argumentum egy tetszőleges kifejezés.

`AKif` Az argumentum egy aritmetikai kifejezés.

Jelentés:

Igaz, ha `X` az `AKif` aritmetikai kifejezés értéke. ∎

[2]SICStus Prolog kiterjesztés

`(<)/2, (>)/2, (=<)/2, (>=)/2, (=:=)/2, (=\=)/2`

Jelentés:

Az eljárások szemantikáját az alábbi táblázat definiálja. A val függvényt annak jelzésére használjuk, hogy az argumentumában szereplő kifejezést a 5.4.2 szakaszban megadottak szerint ki kell értékelni.

hívás	igaz, ha
`AKif1 < AKif2`	$\mathrm{val}(\texttt{AKif1}) < \mathrm{val}(\texttt{AKif2})$
`AKif1 > AKif2`	$\mathrm{val}(\texttt{AKif1}) > \mathrm{val}(\texttt{AKif2})$
`AKif1 =< AKif2`	$\mathrm{val}(\texttt{AKif1}) \leq \mathrm{val}(\texttt{AKif2})$
`AKif1 >= AKif2`	$\mathrm{val}(\texttt{AKif1}) \geq \mathrm{val}(\texttt{AKif2})$
`AKif1 =\= AKif2`	$\mathrm{val}(\texttt{AKif1}) \neq \mathrm{val}(\texttt{AKif2})$
`AKif1 =:= AKif2`	$\mathrm{val}(\texttt{AKif1}) = \mathrm{val}(\texttt{AKif2})$

Ezen eljárások minden argumentuma tisztán bemenő.

Argumentumok:

`AKif1` Az argumentum egy aritmetikai kifejezés.
`AKif2` Az argumentum egy aritmetikai kifejezés.

■

5.5. Kifejezések osztályozása

`var/1, nonvar/1, integer/1, float/1, number/1, atom/1, atomic/1, compound/1`

Jelentés:

Az eljárások szemantikáját az alábbi táblázat definiálja.

hívás	sikerül, ha `X`	hívás	sikerül, ha `X`
`var(X)`	változó	`nonvar(X)`	nem változó
`integer(X)`	egész	`atom(X)`	név
`float(X)`	valós	`atomic(X)`	konstans (szám v. név)
`number(X)`	szám (egész v. valós)	`compound(X)`	összetett

Ezen eljárások argumentuma tisztán bemenő.

Argumentumok:

`X` Az argumentum egy tetszőleges kifejezés.

■

`ground/1, callable/1`

Jelentés:

Az eljárások szemantikáját az alábbi táblázat definiálja.

hívás	sikerül, ha `X`-ben	hívás	sikerül, ha `X`
`ground(X)`	nem szerepel üres változó	`callable(X)`	név vagy összetett

Ezen eljárások argumentuma tisztán bemenő.

Argumentumok:

`X` Az argumentum egy tetszőleges kifejezés.

Kompatibilitás:

Mindkét eljárás SICStus Prolog kiterjesztés. ■

Ezek logikailag nem tiszta eljárások, hiszen pl.

```
var(X), X=1
```

sikeresen lefut, míg ha a két hívást felcseréljük:

```
X=1, var(X)
```

a célsorozat meghiúsul.

P29 Példa: intelligens `between`

```
between(N, M, I):-
      var(I), !, N =< M, between1(N, M, I).
between(N, M, I):-
      integer(I), N =< I, I =< M.

between1(N, M, N).
between1(N, M, I):-
      N < M, N1 is N+1,
      between1(N1, M, I).
```

5.6. Kifejezések összehasonlítása és egyesítése

`(==)/2, (\==)/2, (@<)/2, (@=<)/2, (@>)/2, (@>=)/2`

Jelentés:

Az eljárások szemantikáját az alábbi táblázat definiálja:

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

Ezen eljárások minden argumentuma tisztán bemenő.

Argumentumok:

`Kif1` Az argumentum egy tetszőleges kifejezés.
`Kif2` Az argumentum egy tetszőleges kifejezés.

■

A fenti táblázatban használt $\prec$ relációjel a 4.7.4 fejezetben definiált szabványos rendezés.

SICStus Prologban a változókat az életkoruk szerint rendezik, azok a változók amelyekkel a rendszer előbb találkozott megelőzik azokat, amelyekkel később.

SICStus Prologban előállíthatók végtelen (ciklikus) kifejezések, amelyekre a fenti rendezés nem érvényes.

`(=)/2` — egyesítés

Hívási minta:

`?Kif1 = ?Kif2`

Argumentumok:

`Kif1` Az argumentum egy tetszőleges kifejezés.

`Kif2` Az argumentum egy tetszőleges kifejezés.

Jelentés:

Igaz, ha `Kif1` és `Kif2` egyesíthető.

Megjegyzések:

Az eljárás definíciója: `X = X.` ■

`(\=)/2` — nem egyesíthető

Hívási minta:

`@Kif1 \= @Kif2`

Argumentumok:

`Kif1` Az argumentum egy tetszőleges kifejezés.
`Kif2` Az argumentum egy tetszőleges kifejezés.

Jelentés:

Igaz, ha `Kif1` és `Kif2` nem egyesíthető. ■

`unify_with_occurs_check/2`

Hívási minta:

`unify_with_occurs_check(?Kif1, ?Kif2)`

Argumentumok:

`Kif1` Az argumentum egy tetszőleges kifejezés.
`Kif2` Az argumentum egy tetszőleges kifejezés.

Jelentés:

Igaz, ha `Kif1` és `Kif2` előfordulás-ellenőrzéssel egyesíthető. (Lásd 3.4.8.) ■

Példák:

```
?- X == Y.
no

?- append([], X, Y), X == Y.
yes

?- X \= 1.
no

?- X \== 1.
yes

| ?- unify_with_occurs_check(X, f(X)).
no
```

`compare/3`

Hívási minta:

`compare(?Rel, @Kif1, @Kif2)`

Argumentumok:

`Rel` Az argumentum egy tetszőleges kifejezés.
`Kif1` Az argumentum egy tetszőleges kifejezés.
`Kif2` Az argumentum egy tetszőleges kifejezés.

Jelentés:

Igaz, ha `Kif1` és `Kif2` adott `Rel` relációban van a kifejezések szabványos sorrendjében.

`Rel` értéke `=`, ha `Kif1` azonos `Kif2`-vel; `<`, ha `Kif1` megelőzi `Kif2`-t; `>`, ha `Kif2` előzi meg `Kif1`-et.

Kompatibilitás:

SICStus Prolog kiterjesztés. ■

P30 Példa: Tetszőleges kifejezéslisták rendezése

```
% rendez(XL, ZL): az XL lista compare szerinti rendezése ZL
% (ismétlődések kiszűrésével).
rendez([X|XL], ZL) :-
      rendez(XL, YL),
      beszur(X, YL, ZL).
rendez([], []).

% beszur(X, YL, ZL): az YL rendezett listába beszúrva X-et kapjuk ZL-t
beszur(X, [Y|YL], ZL) :-
      compare(Rel, X, Y),
      beszur(Rel, X, Y, YL, ZL).
beszur(X, [], [X]).

% beszur(Rel, X, Y, YL, ZL): az [Y|YL] rendezett listába beszúrva X-et
% kapjuk ZL-t, feltéve, hogy X és Y relációja Rel.
beszur(<, X, Y, YL, [X,Y|YL]).
beszur(=, X, _, YL, [X|YL]).
beszur(>, X, Y, YL, [Y|ZL]) :-
      beszur(X, YL, ZL).
```

Futása:

```
| ?- rendez([f(1), f(1,_), f(2), f(_,2), _, 1, 1.2, 0.9, 1.0], L).

L = [_A,0.9,1.0,1.2,1,f(1),f(2),f(_B,2),f(1,_C)] ?
```

5.7. Listakezelés

`length/2`

Hívási minta:

`length(?L, +N)`
`length(@L, -N)`

Argumentumok:

- `L` Az argumentum egy tetszőleges kifejezés.
- `N` Az argumentum egy egész szám.

Jelentés:

Igaz, ha `L` lista hossza `N`.

Megjegyzések:

Hajlandó adott hosszúságú, csupa különböző változóból álló listát létrehozni.

Kompatibilitás:

SICStus Prolog kiterjesztés. ∎

A `length/2` definícióját lásd a P12 példában.

`sort/2`

Hívási minta:

`sort(@L, ?S)`

Argumentumok:

- `L` Az argumentum tetszőleges kifejezések listája.
- `S` Az argumentum egy tetszőleges kifejezés.

Jelentés:

Igaz, ha `L` lista `@<` szerinti rendezése `S`, (`==/2` szerint azonos elemek ismétlődését kiszűrve).

Megjegyzések:

Eredménye azonos a P30 példában leírt `rendez/2` eljárásával.

Kompatibilitás:

SICStus Prolog kiterjesztés. ∎

`keysort/2`

Hívási minta:

`keysort(@L, ?S)`

Argumentumok:

- `L` Az argumentum `-/2` funktorú kifejezések listája.
- `S` Az argumentum egy tetszőleges kifejezés.

Jelentés:

Igaz, ha az `L` lista `Key-Value` párokból áll és `S` az `L` lista `Key` értékei szerinti szabványos rendezése.

Kompatibilitás:

SICStus Prolog kiterjesztés. ∎

Példa:

```
?- keysort([[new,york]-amerikai, budapest-magyar, varso-lengyel,
                 amszterdam-holland], L).

L = [amszterdam - holland,budapest - magyar,
        varso - lengyel,[new,york] - amerikai]
```

5.8. Kifejezések szétszedése és összereakása

Itt olyan szétszedő illetve összerakó eljárásokat mutatunk, amelyek nem helyettesíthetők egyesítéssel. Ezekre például akkor van szükség, amikor egy olyan struktúrát szeretnénk részeire bontani, amelynek a program írásakor még nem ismerjük a nevét.

5.8.1. Struktúrák szétszedése és összerakása

`functor/3`

Hívási minta:

```
functor(-Kif, +Nev, +ArgSzam)
functor(+Kif, ?Nev, ?ArgSzam)
```

Argumentumok:

`Kif` Az argumentum egy összetett kifejezés, név vagy szám.
`Nev` Az argumentum egy név vagy egy szám.
`ArgSzam` Az argumentum egy egész.

Jelentés:

Igaz, ha `Kif` egy `Nev`/`ArgSzam` funktorú kifejezés.

Megjegyzések:

Ha `Kif` struktúra, `Nev` illesztődik a (rekord)névvel, `ArgSzam` az argumentumszámmal. Ha `Kif` konstans, `Nev` = `Kif`, `ArgSzam` = 0. Ha `Kif` változó, akkor a `Nev` névből `ArgSzam` argumentumú struktúra épül, csupa különböző új változóval argumentumként, és ez helyettesítődik `Kif`-be. ∎

arg/3

Hívási minta:

`arg(+Sorszam, +Kif, ?Arg)`

Argumentumok:

`Sorszam` Az argumentum egy egész.
`Kif` Az argumentum egy összetett kifejezés.
`Arg` Az argumentum egy tetszőleges kifejezés.

Jelentés:

A `Kif` struktúra `Sorszam`-adik argumentuma `Arg`. ∎

(=..)/2

Hívási minta:

```
+Kif =..  ?Lista
-Kif =..  +Lista
```

Argumentumok:

`Kif` Az argumentum egy tetszőleges kifejezés.
`Lista` Az argumentum egy lista, az első eleme egy név vagy egy szám, a többi eleme tetszőleges kifejezés. A lista első eleme csak akkor lehet szám, ha több eleme már nincsen.

Jelentés:

Igaz, ha `Kif = rekord(`A_1`,..., `A_n`)` és `Lista = [rekord,`A_1`,...  `A_n`]`. ∎

copy_term/2

Hívási minta:

`copy_term(?Kif1, ?Kif2)`

Argumentumok:

`Kif1` Az argumentum egy tetszőleges kifejezés.
`Kif2` Az argumentum egy tetszőleges kifejezés.

Jelentés:

Igaz, ha `Kif2` a `Kif1` kifejezés egy olyan másolata, amelyben az összes `Kif1`-ben előforduló változót szisztematikusan lecseréltünk egy teljesen új változóra. ∎

Példa:

```
| ?- copy_term(X+X+Y, A+B+B).

B = A ?

yes
```

5.8.2. Konstansok szétszedése és összerakása

Itt azokat a beépített eljárásokat mutatjuk be, amelyekkel például egy atomot (nevet) tudunk *karakterkódokra* bontani.

A karakterkódok egész számok, méretüket az ISO szabvány nem szabályozza, SICStus Prologban akár 31 bitesek is lehetnek. SICStus Prologban a 0–127 értékek az ASCII kódnak megfelelő karaktereket jelentik. (Átlagos magyar terminálon a 255-nél nagyobb kódok nem jeleníthetők meg, ilyenkor csonkolás történik.)

Az ISO Prolog bevezette a karakter fogalmát; egy karakter az egy hosszúságú (egyetlen jelből álló) atom.

Ezzel kapcsolatban egy sajnálatos konfliktushelyzet keletkezett: míg a hagyományos Prolog rendszerekben (és így a SICStus Prolog `sicstus` üzemmódjában is) az `atom_chars/2` eljárás az atomot karakterkódokra bontja, addig a szabvány szerint ugyanez az eljárás karakterek listáját állítja elő (és természetesen így viselkedik a SICStus Prolog `iso` üzemmódban). Szerencsére a szabvány minden olyan eljárás mellé, amely karakterekkel dolgozik, definiál egy variánst, amely karakterkódokkal működik.[3] Például: az `atom_chars/2` megfelelője az `atom_codes/2`. A jegyzetben mi mindig az utóbbi, a kódokkal dolgozó eljárásokat használjuk, a félreértések elkerülése érdekében.

`char_code/2`

Hívási minta:

```
char_code(+Karakter, ?Kód)
char_code(-Karakter, +Kód)
```

Argumentumok:

`Karakter` Az argumentum egy karakter.
`Kód` Az argumentum egész karakterkód.

Jelentés:

Igaz, ha a `Karakter` karakter karakterkódja `Kód`. ∎

`atom_chars, atom_codes/2`

Hívási minta:

```
atom_chars(+Atom, ?KarakterLista)
atom_chars(-Atom, +KarakterLista)
atom_codes(+Atom, ?SzámLista)
atom_codes(-Atom, +SzámLista)
```

Argumentumok:

`Atom` Az argumentum egy atom.
`KarakterLista` Az argumentum karakterek listája.
`SzámLista` Az argumentum karakterkódok listája.

Jelentés:

Igaz, ha `Atom` egyes karaktereinek (azok kódjának) a listája `KarakterLista` (`SzamLista`).

Megjegyzések:

Ha híváskor `Atom` ismert, és a $c_1c_2...c_n$ karakterekből áll, akkor a rendszer ezt szétszedi egy $[c_1, c_2, ..., c_n]$ karakterlistává illetve egy $[k_1, k_2, ..., k_n]$ számlistává, ahol k_i a c_i karakterkódja, és ezt egyesíti az eljárás második argumentumával. Ha `Atom` változó, akkor a `KarakterLista`-ból illetve a `SzamLista` karakterkód-listából összerak egy nevet, és azt írja be `Atom`-ba. ∎

`number_chars, number_codes/2`

Hívási minta:

```
number_chars(+Szám, ?KarakterLista)
number_chars(-Szám, +KarakterLista)
```

[3] Ez alól a `char_code/2` eljárás kivétel.

`number_codes(+Szám, ?SzámLista)`
`number_codes(-Szám, +SzámLista)`

Argumentumok:

`Szám` Az argumentum egy number.
`KarakterLista` Az argumentum karakterek listája.
`SzámLista` Az argumentum karakterkódok listája.

Jelentés:

Igaz, ha `Szám` tizes számrendszerbeli alakjában az egyes karaktereknek (azok kódjának) a listája `KarakterLista` (`SzamLista`).

Megjegyzések:

Ha `Szám` tizes számrendszerbeli alakja a $c_1 c_2 ... c_n$ karakterekből áll, akkor `KarakterLista = [`$c_1, c_2, ..., c_n$`]` lesz, illetve `SzamLista = [`$k_1, k_2, ..., k_n$`]` lesz, ahol k_i a c_i karakterkódja. Ha `Szám` változó, akkor a `KarakterLista`-ból illetve a `SzamLista` karakterkód-listából összerak egy számot, és azt írja be `Szám`-ba.

Valójában abban az esetben, amikor karakter(kód) listából készítünk számot, nem csak tizes számrendszerbeli alakot fogad el, de ezt nem érdemes kihasználni. ■

`atom_length/2`

Hívási minta:

`atom_length(+Atom, ?Hossz)`

Argumentumok:

`Atom` Az argumentum egy atom.
`Hossz` Az argumentum egy egész.

Jelentés:

Igaz, ha `Hossz` az `Atom` karaktereinek a száma. ■

`atom_concat/3`

Hívási minta:

`atom_concat(?Atom1, ?Atom2, +Atom12)`
`atom_concat(+Atom1, +Atom2, -Atom12)`

Argumentumok:

`Atom1` Az argumentum egy atom.
`Atom2` Az argumentum egy atom.
`Atom12` Az argumentum egy atom.

Jelentés:

Igaz, ha `Atom2`-őt `Atom1` mögé fűzve `Atom12`-t kapjuk. ■

`sub_atom/5`

Hívási minta:

`sub_atom(+Atom, ?Előtt, ?Hossz, ?Után, ?Rész)`

Argumentumok:

`Atom` Az argumentum egy atom.
`Előtt` Az argumentum egy egész.
`Hossz` Az argumentum egy egész.
`Után` Az argumentum egy egész.
`Rész` Az argumentum egy atom.

Jelentés:

Igaz, ha `Atom` szétbontható három folytonos részre, úgy hogy azok hossza rendre `Előtt`, `Hossz` és `Után` valamint a `Rész` atom a középső darab. ■

5.9. Összes megoldás keresése

`findall/3`

Hívási minta:

`findall(?Gyüjtő, :+Cél, ?Lista)`

Argumentumok:

`Gyüjtő` Az argumentum egy tetszőleges kifejezés.
`Cél` Egy meghívható kifejezés.
`Lista` Tetszőleges kifejezések listája.

Jelentés:

A `Cél` összes megoldására `Gyüjtő` értéke listába gyüjtve `Lista`.

Hatás:

A `Cél` hívás összes megoldását visszaléptetéssel keresi meg. `Lista` elemei abban a sorrendben vannak a listában, ahogy `Cél` különböző megoldásai előállítják `Gyüjtő` értékeit. ∎

Példa:

```
findall(X, (member(Y,[1,2,3]), X is Y*Y), L)

L = [1,4,9]
```

`bagof/3`

Hívási minta:

`bagof(?Gyüjtő, :+Cél, ?Lista)`

Argumentumok:

`Gyüjtő` Az argumentum egy tetszőleges kifejezés.
`Cél` Egy meghívható kifejezés.
`Lista` Tetszőleges kifejezések listája.

Jelentés:

`Lista` az összes olyan `Gyüjtő` behelyettesítés nem üres listája, amely a `Cél` egy megoldását adja.

Hatás:

A `Cél` hívás összes megoldását visszaléptetéssel keresi meg.

Megjegyzések:

Ha `Cél`-ban nincs más üres változó mint `Gyüjtő`-ben, akkor lényegében azonos `findall(Gyüjtő, Cél, Lista)`-val, csak az a különbség, hogy meghiúsul, ha `Cél`-nak nincs megoldása. Ha `Cél`-ban van további üres változó, amely nincs a ^ operátorral lekötve, akkor ezen további változók minden lehetséges értékkombinációjára külön gyüjti ki a `Gyüjtő`-értékeket `Lista`-ba, és ezeket visszalépéskor sorolja fel.

Az, hogy a szabad változók különböző behelyettesítései milyen sorrendben fordulnak elő, nem definiált. ∎

Példa:

```
varos(becs, osztrak).
varos(budapest, magyar).
varos(graz, osztrak).
varos(pecs, magyar).
varos(pozsony, szlovak).
varos(szeged, magyar).

?- bagof(Varos, Orszag^ varos(Varos, Orszag), L)
```

```
               /* Y ^ Cél olvasd:  létezik olyan Y hogy Cél igaz*/

    L = [becs,budapest,graz,pecs,pozsony,szeged]
```

de

```
    ?- bagof(Varos, varos(Varos, Orszag), L).

    L = [becs,graz]
    Orszag = osztrak ;

    L = [pozsony]
    Orszag = szlovak ;

    L = [budapest,pecs,szeged]
    Orszag = magyar ;

    No
```

`setof/3`

Hívási minta:

`setof(?Gyüjtő, :+Cél, ?Lista)`

Argumentumok:

`Gyüjtő` Az argumentum egy tetszőleges kifejezés.
`Cél` Egy meghívható kifejezés.
`Lista` Tetszőleges kifejezések listája.

Jelentés:

Ugyanaz mint `bagof`, de `Lista` rendezett (ld. `sort/2`), ismétlődések nélkül. ■

5.10. Kiírás

A kifejezések írása és olvasásakor fontos szerepet játszanak az operátorok. Fontos, hogy ügyeljünk arra, hogy a kiíráskor élő operátorok beolvasáskor is éljenek, különben a beolvásás nem (vagy nem megfelelően) sikerül.

Operátorok deklarálása

`op/3`

Hívási minta:

`op(+Prec, +Típus, +Név)`

Argumentumok:

`Prec` Az argumentum egy 0–1200 közötti egész.
`Típus` Az argumentum az `fx`, `fy`, `xfx`, `xfy`, `yfx` nevek valamelyike.
`Név` Az argumentum egy név.

Hatás:

Ha `Prec` > 0, akkor `Név`-et felveszi az operátortáblába `Prec` precedenciával és `Tipus` típussal. Ha `Prec` $= 0$, akkor eltávolítja `Név`-et az operátortáblából. ■

`current_op/3`

Hívási minta:

`current_op(?Prec, ?Típus, ?Név)`

Argumentumok:

`Prec` Az argumentum egy 0–1200 közötti egész.

`Típus` Az argumentum az `fx`, `fy`, `xfx`, `xfy`, `yfx` nevek valamelyike.

`Név` Az argumentum egy név.

Jelentés:

Igaz, ha a `Név` operátor `Prec` precedenciával és `Tipus` típussal szerepel az operátortáblában. Többszörösen sikerülhet. ■

Kifejezések kiírása:

`write/1, write_term/2`

Hívási minta:

```
write(@X)
write_term(@X, @Opciók)
```

Argumentumok:

`X` Az argumentum egy tetszőleges kifejezés.

`Opciók` Az argumentum írási opciók listája.

Hatás:

Kiírja `X`-et az `Opciók` listának megfelelően. A `write/1` ha szükséges operátorokat, zárójeleket használ.

Megjegyzések:A `write(`*X*`)` hívás hatása megegyezik a `write(`*X*`, [])` hívásával.

Az írási opciók:

`quoted(`*B*`)` Ha *B* `true`, akkor minden nevet egyszeres idézőjelek között ír ki amennyiben ez szükséges ahhoz, hogy a `read/1` eljárás be tudja olvasni.

`ignore_ops(`*B*`)` Ha *B* `true`, akkor az összetett kifejezéseket funkcionális jelöléssel írja ki, sem operátorokat sem listajelölést nem használ.

`numbervars(`*B*`)` Ha *B* `true`, akkor a `'$VAR'(`*N*`)` alakú kifejezéseket (ahol *N* egy egész szám) egy nagybetű és számok sorozataként írja ki. A nagybetű az angol ABC (N mod 26) + 1 sorszámú betűje lesz, a követő szám pedig $N//26$ lesz, feltéve, hogy ez nem 0.

■

`writeq/1`

Hívási minta:

```
writeq(@X)
```

Argumentumok:

`X` Az argumentum egy tetszőleges kifejezés.

Hatás:

Mint `write(X)`, csak gondoskodik, hogy szükség esetén a nevek idézőjelek közé legyenek téve, hogy a kiírt kifejezés `read`-del visszaolvasható legyen (feltéve, hogy olvasáskor a használt operátorok deklarálva vannak).
■

`write_canonical/1`

Hívási minta:

```
write_canonical(@X)
```

Argumentumok:

`X` Az argumentum egy tetszőleges kifejezés.

Hatás:

Mint `writeq(X)`, csak operátorok nélkül, minden struktúra szabványos alakban jelenik meg. ∎

Példa:

```
| ?- write_canonical(1+2).
+(1, 2)
yes
| ?- write_canonical([1,2]-[]).
-(.(1,.(2,[])),[])
yes
```

`print/1`

Hívási minta:

`print(@X)`

Argumentumok:

`X` Az argumentum egy tetszőleges kifejezés.

Hatás:

Alapértelmezésben azonos `write`-tal. Ha a felhasználó definiál egy `portray/1` eljárást, akkor a rendszer minden a `print`-tel kinyomtatandó részkifejezésre meghívja `portray`-t. Ha ez a hívás sikerül, akkor feltételezi, hogy a `portray` elvégezte a szükséges kiírást, ha meghiúsul, akkor maga írja ki a részkifejezést.

Kompatibilitás:

SICStus Prolog kiterjesztés. ∎

`portray/1`

Hívási minta:

`portray(@Kif)`

Jelentés:

Igaz, ha `Kif` kifejezést a Prolog rendszernek nem kell kiírnia.

Hatás:

Alkalmas formában kiírja a `Kif` kifejezést.

Megjegyzések:

Ez egy felhasználó által definiálandó (*kampó*) eljárás (angolul hook predicate). ∎

Pl., ha felvesszük a következő klózt:

```
portray(szemely(Nev,_,_)):-
      write(szemely(Nev,...)).
```

akkor a `print/1` a következőképen működik:

```
| ?- print([szemely(kiss, istvan, 1960),szemely(nagy, gabor, 1945)]).
[szemely(kiss, ...),szemely(nagy, ...)]
Yes
```

Formázott kifejezés-kiírás

`format/2`

Hívási minta:

`format(@Formátum, @AdatLista)`

Argumentumok:

`Formátum` Az argumentum egy név vagy karakterkódok listája.
`AdatLista` Az argumentum tetszőleges kifejezések listája.

Hatás:

A `Formátum`-nak megfelelő módon kiírja `AdatLista`-t.
A formázójelek alakja: ~<*szám esetleg*> < *formázójel*>.

Kompatibilitás:

SICStus Prolog kiterjesztés. ∎

A `format/2` legfontosabb formázójelei:

	Adattal		*Adat nélkül*
`d`	(decimális) egész szám	`t`	tabuláció
`D`	(decimális) szám, csoportosítva	`n`	újsor
`f`	lebegőpontos szám	`|`	abszolút tabulátorpozició
`w`	tetszőleges kifejezés, mint write	`+`	relativ tabulátorpozició
`q`	tetszőleges kifejezés, mint writeq		
`p`	tetszőleges kifejezés, mint print		

Példa:

```
simple_statistics :-
      <obtain statistics>                % left to the user
      format('~tStatistics~t~72|~n~n'),
      format('Runtime:  ~`.t ~2f~34| Inferences:  ~`.t ~D~72|~n',
            [RunT, Inf]),
      ...
```

Eredménye:

```
                           Statistics


 Runtime: .................. 3.45  Inferences: .......... 60,345
```

Karakterek és byte-ok kiírása:

Karaktereket csak szöveges üzemmódban megnyitott csatornákra írhatunk és ilyenekről olvashatunk, byte-okat pedig csak bináris csatornákkal használhatunk. (Lásd `open/4`.)

`put_char/1, put_code/1, put_byte/1`

Hívási minta:

`put_char(@Karakter)`
`put_code(@Kód)`
`put_byte(@Byte)`

Argumentumok:

`Karakter` Az argumentum egy karakter.
`Kód` Az argumentum egy karakterkód.
`Byte` Az argumentum egy byte.

Hatás:

Kiírja az adott (karakterkódú) karaktert vagy byte-ot. ∎

`tab/1`

Hívási minta:

`tab(@N)`

Argumentumok:

`N` Az argumentum egy egész szám.

Hatás:

Kiír `N` szóközt feltéve, hogy `N` > 0.

Kompatibilitás:

SICStus Prolog kiterjesztés. ■

`nl/0`

Hívási minta:

`nl`

Jelentés:

Igaz.

Hatás:

Kiír egy soremelést. ■

5.11. Beolvasás

Kifejezések beolvasása:

`read/1, read_term/2`

Hívási minta:

```
read(?Kif)
read_term(?Kif, +Opciók)
```

Argumentumok:

`Kif` Az argumentum egy tetszőleges kifejezés.
`Opciók` Az argumentum olvasási opciók listája.

Jelentés:

Igaz, ha a beolvasott kifejezés egyesíthető a `Kif` kifejezéssel és az `Opciók` listában szereplő opciók egyesíthetők a kifejezéshez tartozó opcókkal.

Hatás:

Beolvas egy ponttal lezárt kifejezést, egyesíti `Kif`-fel és kitölti a megadott opció-listát.

Megjegyzések:

File végénél `Kif = end_of_file`. A `read(X)` hívás megegyezik `read_term(X, [])` hívással.

Olvasási opciók:

`variables(`*`V`*`)` *`V`* a beolvasott kifejezésben szereplő változók listája (balról jobbra haladva).

`variable_names(`*`VN`*`)` *`VN`* `A = V` alakú párok listája, ahol `V` a beolvasott kifejezésben szereplő nem névtelen változó, `A` pedig egy atom amely nyomtatott alakja megegyezik a változó nevével.

`singletons(`*`Sz`*`)` *`Sz`* `A = V` alakú párok listája, ahol `V` egy a beolvasott kifejezésben pontosan egyszer előforduló nem névtelen változó, `A` pedig egy atom amely nyomtatott alakja megegyezik a változó nevével.

■

`char_conversion/2`

Hívási minta:

`char_conversion(+KarBe, +KarKi)`

Argumentumok:

`KarBe` Az argumentum egy karakter.
`KarKi` Az argumentum egy karakter.

Hatás:

A `KarBe` → `KarKi` párt hozzáveszi az aktuális karakterkonverziós leképezéshez. Ezután egy kifejezés beolvasása során `KarBe` összes idézőjelek közé nem tett előfordulását lecseréli a `KarKi` karakterre.

Ha a két argumentum megegyezik, akkor a korábbi `KarBe` karakterhez tarozó párt eltávolítja a leképezésből.
■

`current_char_conversion/2`

Hívási minta:

`current_char_conversion(?KarBe, ?KarKi)`

Argumentumok:

`KarBe` Az argumentum egy karakter.
`KarKi` Az argumentum egy karakter.

Jelentés:

Igaz, ha a `KarBe` → `KarKi` pár szerepel az aktuális karakterkonverziós leképezésben. Többszörösen sikerülhet.
■

Karakterek beolvasása:

`get_char/1, get_code/1, get_byte/1`

Hívási minta:

```
get_char(?Karakter)
get_code(?Kód)
get_byte(?Byte)
```

Argumentumok:

`Karakter` Az argumentum egy karakter.
`Kód` Az argumentum egy karakterkód.
`Byte` Az argumentum egy byte.

Jelentés:

Igaz, ha a beolvasott karakter/karakterkód/byte `Karakter`/ `Kód`/`Byte`.

Hatás:

Beolvas egy karaktert/byte-ot, és (karakterkódját) egyesíti `Karakter`-rel (`Kód`-dal) illetve `Byte`-tal.

Megjegyzések:

File végénél `Karakter = end_of_file`, `Kód = -1`, `Byte = -1`. ■

`get/1`

Hívási minta:

`get(?Kar)`

Argumentumok:

`Kar` Az argumentum egy tetszőleges kifejezés.

Jelentés:

Igaz, ha a következő látható (nem-layout) karakter karakterkódja `Kar`.

Hatás:

Beolvassa a következő látható (nem-layout) karaktert és karakterkódját egyesíti `Kar`-ral.

Megjegyzések:

File végénél `Kar = -1`.

Kompatibilitás:

SICStus Prolog kiterjesztés. ■

`peek_char/1, peek_code/1, peek_byte/1`

Hívási minta:

```
peek_char(?Karakter)
peek_code(?Kód)
peek_byte(?Byte)
```

Argumentumok:

`Karakter` Az argumentum egy karakter.
`Kód` Az argumentum egy karakterkód.
`Byte` Az argumentum egy byte.

Jelentés:

Igaz, ha a beolvasható karakter/karakterkód/byte `Karakter`/`Kód`/`Byte`.

Hatás:

Megnézi a soronkövetkező karaktert/byte-ot, és (karakterkódját) egyesíti `Karakter`-rel (`Kód`-dal) illetve `Byte`-tal. A karaktert nem távolítja el a bemenetről.

Megjegyzések:

File végénél `Karakter = end_of_file`, `Kód = -1`, `Byte = -1`. ■

P31 Példa: Számbeolvasás

```
% szambe(Kar, Szam, Kov) beolvassa a kovetkező számot az
% input-folyamból, Kar-t tekintve a soronkövetkező karakternek.  Kov-ben
% visszaadja a szamot követő első karaktert.

szambe(Kar, Szam, Kov):-
      szamjegy(Kar, Ertek), get_code(Kar1),
      szambe(Kar1, Ertek, Szam, Kov).

szambe(Kar, Szam0, Szam, Kov):-
      szamjegy(Kar, E), !,
      get_code(Kar1), Szam1 is Szam0*10+E,
      szambe(Kar1, Szam1, Szam, Kov).
szambe(Kar, Szam, Szam, Kar).

szamjegy(Kar, Ertek):-
      Kar >= 0'0, Kar =< 0'9, Ertek is Kar - 0'0.
```

5.12. Bevitel/kiírás szervezése

Az alábbi eljárásokkal kiviteli/beviteli csatornákat nyithatunk meg, zárhatunk be illetve tehetünk aktuális kiviteli/beviteli csatornává.

Csatorna szinte minden lehet, amiből olvasni vagy amibe írni lehet, például egy hálózati kapcsolat, egy mindent elnyelő objektum de leggyakrabban egy file. A különféle objektumok megnyitására természetesen külön eljárások vannak, írni vagy olvasni azonban mindegyiket ugyanúgy kell.

Minden időpontban van egy aktuális kimenet és egy aktuális bemenet. Az előzőleg az 5.10–5.11 szakaszokban leírt beviteli/kiviteli eljárások mindig az aktuális csatornára vonatkoznak.

open/3, open/4

Hívási minta:

`open(@Filenév, @Mód, -Csatorna)`
`open(@Filenév, @Mód, -Csatorna, @Opciók)`

Argumentumok:

`Filenév` Az argumentum egy atom, ami egy fájl nevét adja.
`Mód` Az argumentum a `read`, `write`, `append` atomok valamelyike.
`Csatorna` Az argumentum egy csatorna.
`Opciók` Az argumentum megnyitási opciók listája.

Hatás:

Megnyitja a `Filenév` nevű file-t `Mód` módban, az `Opciók` listának megfelelően. A `Csatorna` argumentumban visszaadja a megnyitott csatorna nevét.

Megjegyzések:

A legfontosabb opció: `type(`*T*`)`, ahol *T* `text` vagy `binary`. Az első esetben szöveges, a másodikban bináris üzemmódot írunk elő. ■

set_input/1, set_output/1

Hívási minta:

`set_input(@Csatorna)`
`set_output(@Csatorna)`

Argumentumok:

`Csatorna` Az argumentum egy csatorna.

Hatás:

A `Csatorna` lesz a jelenlegi beviteli/kiviteli csatorna. ■

current_input/1, current_output/1

Hívási minta:

`current_input(?Csatorna)`
`current_output(?Csatorna)`

Argumentumok:

`Csatorna` Az argumentum egy csatorna.

Jelentés:

Igaz, ha a jelenlegi beviteli/kiviteli csatorna `Csatorna`. ■

close/1, close/2

Hívási minta:

`close(@Csatorna)`
`close(@Csatorna, @Opciók)`

Argumentumok:

`Csatorna` Az argumentum egy csatorna.
`Opciók` Az argumentum csatornazárási opciók listája.

Hatás:

Az `Opciók` opcióknak megfelelően lezárja a `Csatorna` csatornát.

Megjegyzések:

Csatornazárási opciók:

`force(`*B*`)` Ha *B* `true`, akkor a rendszer a csatorna bezárása közben keletkező hibákat figyelemen kívül hagyva lezárja a csatornát és az eljárás sikerül. Ha *B* `false`, akkor hiba esetén a csatornát nem zárja le és hibát jelez. Ez utóbbi az alapértelmezés.

■

`flush_output/1, flush_output/0`

Hívási minta:

`flush_output(?Csatorna)`
`flush_output`

Argumentumok:

`Csatorna` Az argumentum egy csatorna.

Hatás:

A rendszer kiírja a `Csatorna`, illetve a 0 argumentumú változat esetében az aktuális csatorna által pufferelt adatot. ■

`stream_property/2, at_end_of_stream/1, at_end_of_stream/0`

Hívási minta:

`stream_property(?Csatorna, ?Tulajdonság)`
`at_end_of_stream(+Csatorna)`
`at_end_of_stream`

Argumentumok:

`Csatorna` Az argumentum egy csatorna.
`Tulajdonság` Egy csatorna-tulajdonságot leíró kifejezés.

Jelentés:

`stream_property(Csatorna, Tulajdonság)` igaz, ha a `Csatorna` rendelkezik a `Tulajdonság` tulajdonsággal.

`at_end_of_stream(Csatorna)` illetve `at_end_of_stream` igaz, ha `Csatorná`-t illetve az aktuális csatornát már végigolvastuk.

Megjegyzések:

A fontosabb csatorna-tulajdonságok:

`input` A csatorna egy forrás.

`output` A csatorna egy nyelő.

`position(`*`P`*`)` A csatorna aktuális pozíciója *`P`*.

`reposition(`*`B`*`)` *`B`* értéke `true`, ha a csatorna pozíciója állítható, `false`, ha nem.

`type(`*`T`*`)` A csatorna típusa *`T`*. Ez lehet `text` (szöveges) vagy `binary` (bináris).

■

`set_stream_position/2`

Hívási minta:

`set_stream_position(?Csatorna, @Pozíció)`

Argumentumok:

`Csatorna` Az argumentum egy csatorna.
`Pozíció` Egy csatorna-pozíciót leíró kifejezés.

Hatás:

`Pozíció` lesz a `Csatorna` aktuális pozíciója.

Megjegyzések:

`Pozíció` mindig egy (az implementáció által definiált) változómentes kifejezés, amely egyértelműen azonosítja a csatorna egy pozícióját. ■

see/1, tell/1

Hívási minta:

```
see(@Filenév)
tell(@Filenév)
```

Argumentumok:

`Filenév` Az argumentum egy atom.

Hatás:

Első hívásakor megnyitja a `Filenév` file-t olvasásra/írásra és a jelenlegi beviteli/kiviteli csatornává teszi. Későbbi híváskor csak a jelenlegi csatornává teszi.

Kompatibilitás:

Az eljárás DEC10 kompatibilis. ■

seeing/1, telling/1

Hívási minta:

```
seeing(?Filenév)
telling(Filenév)
```

Argumentumok:

`Filenév` Az argumentum egy tetszőleges kifejezés.

Jelentés:

Igaz, ha `Filenév` a jelenlegi beviteli/kiviteli csatorna neve.

Kompatibilitás:

Az eljárás DEC10 kompatibilis. ■

seen/0, told/0

Hívási minta:

```
seen
told
```

Hatás:

Lezárja a jelenlegi beviteli/kiviteli csatornát, a terminál lesz a jelenlegi beviteli/kiviteli csatorna.

Kompatibilitás:

Az eljárás DEC10 kompatibilis. ■

A korábban ismertetett be/kiviteli eljárások mindegyikének van egy eggyel több argumentumú változata, ahol az első argumentumban explicit módon megadható a csatorna, amin a be/kivitelt végezni kell.

write/2, write_term/3, writeq/2, write_canonical/2, print/2, read/2, read_term/3, format/3, put_char/2, put_code/2, put_byte/2, tab/2, nl/1, get_char/2, get_code/2, get_byte/2, get/2, peek_char/2, peek_code/2, peek_byte/2

Hívási minta:

```
write(@Csatorna, @Kif)
write_term(@Csatorna, @Kif, @Opciók)
writeq(@Csatorna, @Kif)
write_canonical(@Csatorna, @Kif)
print(@Csatorna, @Kif)
format(@Csatorna, @Formátum, @AdatLista)
read(@Csatorna, ?Kif)
read_term(@Csatorna, ?Kif, +Opciók)
put_char(@Csatorna, @Karakter)
```

```
put_code(@Csatorna, @Kód)
put_byte(@Csatorna, @Byte)
tab(@Csatorna, @N)
nl(@Csatorna)
get_char(@Csatorna, ?Karakter)
get_code(@Csatorna, ?Kód)
get_byte(@Csatorna, ?Byte)
get(@Csatorna, ?Kód)
peek_char(@Csatorna, ?Karakter)
peek_code(@Csatorna, ?Kód)
peek_byte(@Csatorna, ?Byte)
```

Hatás:

Azonos az első argumentumok elhagyásával keletkező eljárásokéval azzal az eltéréssel, hogy nem az aktuális ki/bemenetet használják, hanem az első argumentumban megadottat. ■

5.13. Egy összetettebb példa

Az alábbi példa bemutatja egyes beolvasási, kiírási és a bevitel/kivitel szervezésére szolgáló eljárások használatát. A példában meghívott `szotar_elem_be` eljárást korábban, a P24 példában (90. oldal) vezettük be.

P32 Példa: Szótárazás

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

% A szotar.txt állományban tárolt szótár lekérdezésére,
% bővítésére és visszaírására ad lehetőséget.
szotar :-
      szotar_be(Sz),
      write('Bevitel: Magyar - Angol.'), nl,
      szotaraz(Sz),
      szotar_ki(Sz).

% Létrehoz egy üres szótárat
ures_szotar :-
      szotar_ki(_).

% beolvassa a szotar.txt file-t
szotar_be(Sz) :-
      see('szotar.txt'),
      read(Kif),
      szotar_be(Kif, Sz),
      seen.

% szotar_be(Kif, Sz): Feldolgozza a Kif beolvasott szótársort, és az
% utána következő szótarsorokat az Sz nyílt végű listába. A 'vege.'
% sor jelzi a szótár végét.
szotar_be(vege, _Sz) :- !.
szotar_be(end_of_file, _Sz) :- !.
szotar_be(Kif, Sz) :-
      memberchk(Kif, Sz),
      read(UjKif),
      szotar_be(UjKif, Sz).
```

```
% kiírja a szotar.txt file-t
szotar_ki(Sz) :-
      tell('szotar.txt'),
      szotar_kiir(Sz),
      write('vege.'), nl,
      told.

% kiírja az Sz nyílt végű lista elemeit külön sorokba
szotar_kiir(Sz) :-
      var(Sz), !.
szotar_kiir([Kif|Sz]) :-
      writeq(Kif), write('.'), nl,
      szotar_kiir(Sz).

% Az Sz nyílt végű listával szótáraz
szotaraz(Sz) :-
      szotar_elem_be(Kerdes),
      feldolgoz(Kerdes, Sz), !, szotaraz(Sz).
szotaraz(_).

% feldolgoz(Kerdes, Sz): Egy beolvasott Kerdes kérdést feldolgoz az Sz
% szótárra vonatkozóan. Meghiúsul, ha a feldolgozásnak vége kell
% legyen.
feldolgoz([vege], _):- !, fail.
feldolgoz(M-A, Sz) :- !,
      memberchk(M-A, Sz),
      write('OK'), nl.
feldolgoz(Szo, Sz) :- !,
      (  kikeres(X-Szo, Sz, magyarul, X) -> true
      ;  kikeres(Szo-X, Sz, angolul,  X) -> true
      ;  write('Nem tudom. '), nl
      ).
feldolgoz(_, _) :-
      write('Nem ertem. '), nl.

% Ha sikerül Par-t az Sz szótárban megtalálni,  kiírja a
% "Valasz: Szosor" szöveget.
kikeres(Par, Sz, Valasz, Szosor) :-
      memberchk(Par, Sz), nonvar(Szosor),
      write(Valasz), write(': '), szosor_ki(Szosor), nl.

% szosor_ki(Szavak): kiírja a Szavak szó-listát.
szosor_ki([]).
szosor_ki([Szo|Szosor]) :-
write(Szo), write(' '), szosor_ki(Szosor).
```

5.14. Programfejlesztés

Az alábbi eljárások nem szabványosak, bár a legtöbb Prolog rendszerben megtalálhatók.

Néhány eljárás *eljárás specifikációt* vár valamelyik argumentumaként. Egy ilyen specifikáció a következő formák valamelyikét öltheti.

- *név* Minden predikátum, aminek ez a neve, tetszőleges aritással.

- *név*/*aritás* A *név* nevű, *aritás* aritású eljárás.
- *név*/*alsó*-*felső* Minden *név* nevű eljárás aminek az aritása *alsó* és *felső* közé esik.
- *modul*:*spec* Minden *modul* modulbeli a *spec* eljárás specifikációnak megfelelő eljárás.
- [$spec_1$,...,$spec_n$] Minden a $spec_i$ specifikációk által meghatározott eljárás.

set_prolog_flag/2

Hívási minta:

`set_prolog_flag(+Jelző, @Érték)`

Argumentumok:

`Jelző` Az argumentum egy Prolog jelző.
`Érték` Az argumentum egy a jelzőnek megfelelő kifejezés.

Hatás:

`Jelző` értkékét `Érték`-re állítja. ■

current_prolog_flag/2

Hívási minta:

`current_prolog_flag(?Jelző, ?Érték)`

Argumentumok:

`Jelző` Az argumentum egy Prolog jelző.
`Érték` Az argumentum egy tetszőleges kifejezés.

Jelentés:

Igaz, ha `Jelző` egy érvényes Prolog jelző és pillanatnyi értkéke `Érték`. ■

A legfontosabb Prolog jelzők:

- `language` Lehetséges értékei: `sicstus` (ez az alapértelmezés) vagy `iso`. Azt adja meg, hogy éppen milyen módban vagyunk.
- `argv` Csak olvasható, értéke a parancssorban kapott argumentumok listája. Például, ha a SICStus Prologot a `% sicstus -a hello world 2001` paranccsal indítottuk, akkor az érték a `[hello,world,'2001']` lista lesz.
- `unknown` Azt adja meg, hogy ha a rendszer egy definiálatlan eljárást hív meg, mit tegyen. Lehetséges értékei:
  - `trace` Elindítja a nyomkövető rendszert.
  - `fail` Meghiúsul.
  - `error` Hibát jelez. (Ez az alapértelmezés.)
- `source_info` Lehetséges értékei: `on`, `off`. Azt adja meg, hogy a rendszer gyűjtsön-e forrás szintű nyomkövetési információt (`on`) vagy se (`off`). Használata elsősorban a GNU Emacs környezetben kényelmes, ilyenkor a rendszer a forrásfile megfelelő sorát szép zölden kivilágítja és jelzi, hogy milyen kapunál járunk.
- `double_quotes` Az idézőjelek (`"`) közé zárt karaktersorozatok háromféle értelmezését teszi lehetővé:
  1. `codes` — karakterkódok listája (alapértelmezés),
  2. `chars` — karakterek (egy hossszú atomok) listája,
  3. `atom` — az adott karakterekből álló atom.

consult/1, '.'/2

Hívási minta:

`consult(@Files)`
`[@File,...]`

Argumentumok:

`Files` Az argumentum egy név vagy nevek listája.
`File` Az argumentum egy név.

Hatás:

Beolvassa a `File`(ok)at.

Megjegyzések:

SICStusban interpretált alakot hoz létre. ■

compile/1

Hívási minta:

`compile(@Files)`

Argumentumok:

`Files` Az argumentum egy név vagy nevek listája.

Hatás:

Beolvassa a `File`(ok)at, leforditott alakot hozva létre. ■

expand_term/2

Hívási minta:

`expand_term(@Kif1, ?Kif2)`

Argumentumok:

`Kif1` Az argumentum egy tetszőleges kifejezés.
`Kif2` Az argumentum egy tetszőleges kifejezés.

Jelentés:

Igaz, ha `Kif1` kifejtése `Kif2`.

Hatás:

A `Kif1` kifejezést kifejti `Kif2`-vé, az alábbi értelemben.

Az `expand_term/2` eljárás először a meghívja a `term_expansion(Kif1,Kif2)` felhasználó által definiálható eljárást. Ha ez sikerül, akkor a `Kif2`-ben visszaadott alakot tekinti az `expand_term` értékének. Ha nem sikerült, megkisérli a DCG nyelvtankiterjesztést alkalmazni (lásd 4.12). Ha ez sem sikerül, `Kif1`-et adja vissza `Kif2`-ként.

Megjegyzések:

Minden beolvasott klóz keresztülmegy ezen a kifejtésen. ■

listing/0, listing/1

Hívási minta:

`listing`
`listing(@EljárásSpec)`

Argumentumok:

`EljárásSpec` Az argumentum egy eljárás specifikáció.

Hatás:

Kiírja az összes/megnevezett interpretált eljárás(oka)t az aktuális kimenetre. ■

trace/0

Hívási minta:

`trace`

Hatás:

Elindítja az interaktív nyomkövetést. ■

`debug/0, zip/0`

Hívási minta:

`debug`
`zip`

Hatás:

Elindítja a szelektív nyomkövetést (spion-pontok, lásd alább).

Megjegyzések:

A két eljárás között annyi a különbség, hogy `zip` módban a rendszer gyorsabb (majdnem olyan gyors, mint ha nem is lenne nyomkövetés), de nem gyűjt annyi információt mint `debug` módban. ■

`nodebug/0, notrace/0, nozip/0`

Hívási minta:

`nodebug`
`notrace`
`nozip`

Hatás:

Leállítja a nyomkövetést. ■

`spy/1`

Hívási minta:

`spy(@EljárásSpec)`

Argumentumok:

`EljárásSpec` Az argumentum egy eljárás specifikáció.

Hatás:

Spion-pontot tesz az `EljárásSpec` által megadott eljárásokra. ■

`nospy/1`

Hívási minta:

`nospy(@EljárásSpec)`

Argumentumok:

`EljárásSpec` Az argumentum egy eljárás specifikáció.

Hatás:

Megszünteti az `EljárásSpec` által megadott eljárásokra kiadott Spion-pontokat. ■

`nospyall/0`

Hívási minta:

`nospyall`

Hatás:

Az összes spion-pontot megszünteti. ■

`leash/1`

Hívási minta:

`leash(@Kapulista)`

Argumentumok:

`Kapulista` Az argumentum kapuk listája.

Hatás:

A `Kapulista` meghatározza, hogy teljes nyomkövetéskor mely kapuknál álljon meg a rendszer.

Megjegyzések:

A listában a következő nevek szerepelhetnek: `call`, `exit`, `redo`, `fail`, `exception` (lásd 3.3). Alapértelmezésben a rendszer minden kapunál megáll. ■

`statistics/0`

Hívási minta:

`statistics`

Hatás:

Különféle statisztikákat ír ki az aktuális kimenetre. ■

`statistics/2`

Hívási minta:

`statistics(?Fajta, ?Ertek)`

Argumentumok:

`Fajta` Az argumentum egy mennyiség neve.
`Ertek` Az argumentum egy tetszőleges kifejezés.

Jelentés:

Igaz, ha `Ertek` a `Fajta` fajtájú mennyiség pillanatnyi értéke. ■

Pl.:

> `statistics(runtime, E)` eredménye
> `E=[Tdiff, T]`
> ahol `Tdiff` az előző lekérdezés óta eltelt idő, `T` a rendszerindítás óta eltelt idő, mindkettő ezredmásodpercben.

`break/0`

Hívási minta:

`break`

Hatás:

Egy új interakciós szintet hoz létre. ■

`abort/0`

Hívási minta:

`abort`

Hatás:

Kilép a legkülső interakciós szintre. ■

`halt/0, halt/1`

Hívási minta:

`halt`
`halt(+Üzenet)`

Argumentumok:

`Üzenet` Az argumentum egy egész.

Hatás:

Kilép a Prolog rendszerből, az esetleges argumentumot átadva a hívó rendszernek.

Kompatibilitás:

ISO szabvány szerinti eljárás. ■

5.15. Hibakezelés (kivételkezelés)

Alapvetően kétféle hibakezelés van a Prolog rendszerekben:

- Kapd el és dobd (Catch and throw)

  Hiba esetén a rendszer visszalép (felgöngyölíti a vermeit) ameddig egy megfelelő hibakezelő eljáráshívásig nem ér, majd ott folytatja.

- helyi hibakezelés

  Hiba esetén a hibát okozó eljáráshivás helyébe lép a hibakezelő eljárás meghívása. Megszakító jellegű hiba esetén a hibakezelő beszúródik a hívásfolyamba. Erre példa az `unknown` Prolog jelző, amivel előírhatjuk, hogy a nem definiált eljárás hívása esetén mit csináljon a rendszer. A helyi hibakezelési lehetőségeket nem tárgyaljuk részletesebben.

`throw/1, raise_exception/1`

Hívási minta:

```
throw(@HibaKif)
raise_exception(@HibaKif)
```

Jelentés:

Se nem igaz, se nem hamis.

Hatás:

Kiváltja a `HibaKif` hibahelyzetet.

Kompatibilitás:

A két eljárás szinonima, de `raise_exception/1` SICStus Prolog specifikus. ■

`catch/3, on_exception/3`

Hívási minta:

```
catch(:+Cél, ?Minta, :+Hibaág)
on_exception(?Minta, :+Cél, :+Hibaág)
```

Argumentumok:

`Cél` Az argumentum egy meghívható kifejezés,
`Minta` Az argumentum egy tetszőleges kifejezés,
`Hibaág` Az argumentum egy meghívható kifejezés,

Jelentés:

Igaz, ha `call(Cél)` igaz, vagy ha a hívását megszakította `throw/1` (vagy `raise_exception/1`) hívása egy olyan argumentummal, ami egyesíthető `Minta`-val és `call(Hibaág)` igaz.

Hatás:

Ha `Cél` végrehajtása során hibahelyzet nem fordul elő, futása azonos `Cél`-lal. Ha `Cél`-ban hiba van, a hibát leíró kifejezést egyesíti `Mintá`-val. Ha ez sikeres, meghívja a `Hibaág`-at. Ellenkező esetben tovább göngyölíti a Prolog eljárásvermét további körülvevő `catch/1` (`on_exception/1`) hívásokat keresve és ezekre megismétli az eljárást.

Hibahelyzetet beépített eljárás rossz argumentumokkal való meghívása, vagy a `throw/1` (`raise_exception/1`) eljárás válthat ki.

Kompatibilitás:

`on_exception/3` SICStus Prolog kiterjesztés. ■

5.15.1. Hibakifejezések

Az ISO szabvány előírja, hogy a beépített eljárások egy `error(Hiba, Részletes)` alakú hibakifejezést „dobjanak”, ahol `Hiba` a szabvány által előírt alakú, míg `Részletes` rendszerfüggő lehet. SICStus Prologban a `Részletes` kifejezés a rendszer által előállított összes hibainformációt tartalmazza, míg `Hiba` ennek a szabvány által előírt kivonata.

Példák beépített hibakifejezésekre:

```
| ?- prolog_flag(language, _, sicstus).

yes
| ?- catch(X is 1+Y, E, true).

E = instantiation_error(_A is 1+_B,2) ?

yes
| ?- prolog_flag(language, _, iso).

yes
| ?- catch(X is 1+Y, E, true).

E = error(instantiation_error,instantiation_error(_A is 1+_B,2)) ?

yes
| ?- X is 1*blabla.
{DOMAIN ERROR: _32 is 1*blabla - arg 2: expected expression, found blabla}
| ?- catch(X is 1*blabla, error(IsoErr,SpErr), true).

SpErr = domain_error(_A is 1*blabla,2,expression,blabla),
IsoErr = type_error(evaluable,blabla) ?

yes
| ?- catch(blabla, error(IsoErr,SpErr), true).

SpErr = existence_error(blabla,0,procedure,user:blabla/0,0),
IsoErr = existence_error(procedure,user:blabla/0) ?

yes
| ?- catch(atom_codes(1+2,X), error(IsoErr,SpErr), true).

SpErr = type_error(atom_codes(1+2,_A),1,atom,1+2),
IsoErr = type_error(atom,1+2) ?

yes
```

5.16. Gyakorló feladatok

GY11.

Tegyük fel, hogy tetszőlegesen nagy egész számok kezelésére van szükségünk, ezért az ilyen számokat a tizes számrendszeri alakjukból képzett számlistával ábrázoljuk, pl. 1256 ábrázolása `[1,2,5,6]`. Írjunk egy `osszead(L1, L2, L3)` eljárást, ahol `L1` és `L2` adott számlisták. Az eljárás állítsa elő azt az `L3` számlistát, amely az `L1` és `L2` által jelölt számok összegét ábrázolja. Pl.:

```
osszead([9,2,5,6], [7,5,8], L)
```

eredménye:

```
L = [1,0,0,1,4].
```

(Vigyázat: nem feltételezhető, hogy a listákkal ábrázolt összeadandó számok a Prolog számtartományán belül vannak)

GY12.

Tegyük fel, hogy a dátumokat egy '`d(Ev,Ho,Nap)`' 3-argumentumú rekordstruktúrával ábrázoljuk. Írjunk egy

```
masnap(Datum, KovDatum)
```

Prolog eljárást, amely adott `Datum` eseten meghatározza a következő nap dátumát `KovDatum`-ban (`Datum`-ról feltételezhetjük, hogy érvényes dátum). A szökőéveket a Gregorián naptár szerint vegyük figyelembe (a 100-zal osztható de 400-zal nem osztható évek nem szökőévek, a többi néggyel osztható év szökőév). Példák:

```
?- masnap(d(1900, 2, 28), Kov).          Kov = d(1900, 3, 1) ;
?- masnap(d(2000, 2, 28), Kov).          Kov = d(2000, 2, 29) ;
```

GY13.

Írjunk egy `gyakorisaga(Nev)` Prolog eljárást, amely a `Nev` atomban előforduló összes különböző karaktert pontosan egyszer, előfordulási gyakoriságával együtt kiírja. Pl.

```
?- gyakorisaga(abbabbac).
```

eredménye lehet a következő:

```
b gyakorisaga 4 a gyakorisaga 3 c gyakorisaga 1
```

(A kiirás sorrendje tetszőleges lehet).

GY14.

Írjunk egy `titkosit(Szoveg, Eltolas, Titkositott)` Prolog eljárást. Ennek hívásakor `Szoveg` egy adott név (atom) lesz, `Eltolas` pedig egy 1 és 25 közé eső egész szám. Az eljárás feladata a `Szoveg`-et karakterenként átalakítani és eredményként egy új nevet létrehozni a `Titkositott` argumentumban. A titkosítás abból áll, hogy a `Szoveg`-ben szereplő minden kisbetű helyett az ABC-ben `Eltolas` hellyel utána következő kisbetűt kell tenni (az eltolás ciklikus, azaz az ABC-ben a z betű után ismét az a betű jön). A nem kisbetű karaktereket a `Szoveg`-ben változatlanul kell hagyni. Példa:

```
titkosit('Zizi zuz?', 6, T).
```

eredménye:

```
T = 'Zofo faf?'
```

GY15.

Írjunk egy `egyszerusit(Kif, UjKif)` Prolog eljárást, amely a `Kif` tetszőleges Prolog kifejezésben előforduló `A+B` alakú részkifejezéseket, ahol `A` és `B` egyaránt számok, az `A` és `B` számok összegére cseréli, a többi részkifejezést változatlanul hagyva. Pl.

```
?- egyszerusit(
      f(a+1,[Y,1+2,Y],3+4),
      Ujkif).
```

eredménye:

```
Ujkif = f(a+1, [Y,3,Y], 7)
```

Kiséreljünk meg olyan megoldást adni, amely a cserét a kifejezésfában alulról felfelé végzi, és ezáltal a kifejezés egyszeri bejárásával azt tovább nem egyszerűsíthető alakra hozza, azaz pl.

```
?- egyszerusit(f(1+2+3+4), Ujkif).
```

eredménye:

```
Ujkif = f(10)
```

GY16.

Írjunk egy olyan `melysege(Kif, N)` Prolog eljárást, amely tetszőleges adott `Kif` Prolog kifejezés esetén `N`-ben visszadja annak mélységét. Egy kifejezés mélységén a neki megfelelő fastruktúra magasságát értjük, másképpen: a nem összetett kifejezések mélysége 0, egy összetett kifejezés mélysége a legnagyobb mélységű argumentumának mélységénél eggyel nagyobb. Például:

```
:- melysege((a+b)*(f(1)+_V), N)          N = 3
:- melysege([1+2,3+4,5+6], N)            N = 4
```

GY17.

Tegyük fel, hogy a repülőtársaságok járataira vonatkozóan a Prolog adatbázisában a következő alakú tényállításokkal rendelkezünk:

```
jarat(Honnan, Hova)
```

Egy ilyen állítás azt fejezi ki, hogy van repülőjárat `Honnan` városból `Hova` városba és vissza. Írjon egy

```
elerheto(N, Honnan, CélLista)
```

Prolog eljárást, amely adott `N` és `Honnan` esetén `CélLista`-ban előállítja a `Honnan` városból *legfeljebb* `N` járattal elérhető városok ismétlődés nélküli listáját.

GY18.

Tegyük fel, hogy a repülőtársaságok járataira vonatkozóan a Prolog adatbázisában a következő alakú tényállításokkal rendelkezünk: `jarat(Honnan, Ind, Hova, Erk)`. Egy ilyen állítás azt fejezi ki, hogy indul repülőjárat `Honnan` városból `Ind` időpontban, amely `Hova` városba `Erk` időpontban érkezik. Az időpontokat `Ora-Perc` alakú Prolog struktúra-kifejezéssel ábrázoljuk (ahol `Ora` és `Perc` egész számok). Írjunk egy olyan `menetrend(Honnan, Hova)` Prolog eljárást, amely adott `Honnan` és `Hova` városok esetén kiírja a két város közötti közvetlen, vagy egyetlen átszállással járó összes utazás adatait. Az átszállóhelyen legalább 45 percet, de legfeljebb 2 órát kell tölteni (és még ugyanaznap tovább kell utazni). A kiírás tartalmazza az (összes) érkezési és indulási időpontot és az átszállóhely nevét is, ha van ilyen, pl.:

```
  bud   waw
  ind   erk   atszallohely   erk   ind
18-00 19-30   NONSTOP
12-00 16-00   prg           13-10 15-10
```

(Megj: nem szükséges a kiírást táblázatba rendezni, csak a fenti információ legyen benne.)

GY19.

Írjunk meg egy `kigyujt(File, Nev)` Prolog eljárást, amelynek feladata, hogy az adott `File`-ban levő sorok közül kiírja a képernyőre azokat, amelyek az adott `Nev` atomot a sorban bárhol tartalmazzák. A sorok végét egyetlen 10-es kódú karakter jelzi (Unix konvenció). A `File` minden sora legfeljebb egyszer íródjék ki.

6. fejezet

Fejlettebb nyelvi és rendszerelemek

6.1. Modularitás

Alapvetően kétféle modulfogalom lehetséges:

- név-alapú
- eljárás-alapú

6.1.1. Név-alapú modell (pl. MProlog, LPA Prolog)

Egy név minden előfordulása (eljárás, konstans, struktúra) vagy látható (visible) vagy lokális (local) az adott modulban.

A lokális nevek csak az adott modulban láthatóak, azaz más modulban nem nevezhetők meg. Ez legegyszerűbben úgy képzelhető el, hogy egy *r* lokális névnek egy *m* modulban való minden előfordulása átneveződik pl. az '*m:r*' névvé. [MPrologban a lokális neveket szisztematikusan ún. kódolt nevekre (#0, #1 stb.) nevezheti át a rendszer.]

Előnyök:

- egyszerű modell
- metahívások „automatikusan" jól kezelődnek, pl.:

```
module m1.
export(p/1).
...
p(X) :- ..., X, ....
...
```

```
module m2.                    module m2.
import(p/1).                  import(p/1).
...                           ...
q :- p(r).                    q :- p('m2:r').
r:- ...                       'm2:r':- ...
```

Hátrányok:

- nem lehetséges pl. p/1-et exportálni, de p/2-t lokálissá tenni.

- az adatnevek (struktúra/konstans) és a velük azonos alakú eljárásnevek láthatósága egymáshoz van kötve.
- az adatnevek alaphelyzetben általában lokálisak, láthatóságukat deklarálni kell.

6.1.2. Eljárás-alapú modell (pl. SWI, SICStus, Quintus)

A kivülről való láthatóságot eljárásokhoz és nem nevekhez kötjük.
Az adatnevek általában minding láthatóak.

Előnyök:

- lehetséges pl. p/1-et exportálni, és p/2-t lokálissá tenni.
- az adatnevek (struktúra/konstans) és a velük azonos alakú eljárásnevek láthatósága nincs egymáshoz kötve.

Hátrányok:

- bonyolultabb modell
- metahívások kezelése külön mechanizmust igényel:

  a. modul-környezet (context-module) nyilvántartás (SWI):
  Futás közben a rendszer állandóan nyilvántartja mely modulban vagyunk. Ez közönséges eljárások esetén az eljárás definícióját tartalmazó modul. A meta-eljárásokat (pl. `p/1` alább) azonban átlátszóaknak kell deklarálni:
  `:- module_transparent p/1.`
  Az ilyen eljárások esetén a modul-környezet a hívótól öröklődik.
  A modul-környezetet használjuk annak megállapítására, hogy egy meta-hívást mely modulban értelmezzünk.
  Például:

```
:- module(m1, p/1).                    % az m1 modul exportálja
                                       % a p/1 eljárást
:- module_transparent p/1.

p(X) :- ...., X, ...

...

:-module m2.

q :- p(r), ...

r:- ...
```

  A `p` eljárás meghívásakor `m2` marad a modul-környezet, mert `p/1` átlátszó, így `X` hívásakor az `X=r` eljárást `m2`-ben keressük.

  b. meta-argumentumok nyilvántartása (Quintus, SICStus):
  A meta-eljárásoknál meg kell nevezni a meta-argumentumpoziciókat, pl. egy olyan `p/3` esetén amelynek a 3. argumentuma eljárás:
  `:- meta_predicate p(+,+,:).`
  A `meta_predicate` deklarációt minden olyan modulban szerepeltetni kell, ahol az adott eljárást meghívjuk!
  A fordítóprogram az adott argumentumpoziciót minden hívásban kiegészíti, pl.:

```
:- module(m2).
:- meta_predicate p(:).

q:- p(r).  ----> q:-p(m2:r)
```

6.1.3. A SICStus modulfogalma

Modul-file első direktívája a

```
:- module( <név>, [<exportált eljárás>,...]).
```

modul-direktíva
Importálni elsősorban a

```
:- use_module( <file>, [<importált eljárás>, ...]).
```

direktívával lehet, amely betölti a `<file>`-t (ha még nincs betöltve), majd abból a kurrens modulba importálja a megadott eljárásneveket. A

```
:- use_module(<file>).
```

direktíva az összes a `<file>`-ban exportált eljárást importálja.

Az ISO Prolog szabvány első része a modulfogalmat nem tárgyalja. A szabvány második része, amely a modulfogalommal foglalkozik, jelenleg kidolgozás alatt áll.

6.2. Külső nyelvi interfész

Hagyományos nyelvű, pl. C nyelven írt programrészek meghívására alapvetően kétféle módszert alkalmaznak:

- A Prolog rendszer elvégzi az átalakítást a Prolog alak és a külső nyelvi alak között. Kényelmesebb, biztonságosabb mint a másik módszer, de kevésbé hatékony. Többnyire csak egyszerű adatokra (egész, valós, atom). (MProlog)
- A külső nyelvi rutin pointereket kap Prolog adatstruktúrákra, valamint hozzáférési algoritmusokat ezek kezelésére. Nehézkesebb, veszélyesebb, de jóval hatékonyabb mint az előző megoldás. Összetett adatok adásvételére is jó. (SWI, SICStus)

Alább egy egyszerű példát közlünk a SICStus külső interfészének használatára. A példa forrása két részből áll, egy Prolog file-ból, ahol deklaráljuk a C-ben megírt eljárást és a C file-ból, ahol megírjuk azt.

Prologban az `index_keys(`*`Spec`*`, `*`Kif`*`, `*`Kulcs`*`, `*`Szám`*`)` eljárást szeretnénk meghívni, aminek a jelentése:

- Ha *`Spec`* és *`Kif`* különböző funktorú kifejezések, akkor *`Szám`* `= -1` és *`Kulcs`* `= []`.
- Egyébként, ha *`Spec`* valamelyik argumentuma `+` és *`Kif`* megfelelő argumentuma változó, akkor *`Szám`* `= -2` és *`Kulcs`* `= []`.
- Egyébként *`Szám`* a *`Spec`* argumentumaként előforduló `+` atomok száma, *`Kulcs`* pedig *`Kif`* megfelelő argumentumok *kivonatából* képzett lista. A kivonat lényegében az argumentum funktora, azzal az eltéréssel, hogy a konstansok kivonata maga a konstans, struktúrák esetén pedig a struktúra neve és az aritása külön elemként kerül a kivonat-listába.

Példa az eljárás használatára:

```
| ?- [ixtest].
| ?- index_keys(f(+, -, +, +), f(12.3, _, s(1, _, z(2)), t), L, X).
L = [12.3,s,3,t], X = 3 ?
yes
```

Az `ixtest.pl` file.

```
foreign(ixkeys, index_keys(+term, +term, -term, [-integer])).
foreign_resource(ixkeys, [ixkeys]).
:- load_foreign_resource(ixkeys).
```

Az `ixkeys.c` file.

```
#include <sicstus/sicstus.h>

#define NA -1                   /* not applicable */
#define NI -2                   /* instantiatedness */

long ixkeys(SP_term_ref spec, SP_term_ref term, SP_term_ref list)
{
  unsigned long sname, tname, plus;
  int sarity, tarity, i;
  long ret = 0;
  SP_term_ref arg = SP_new_term_ref(), tmp = SP_new_term_ref();

  SP_get_functor(spec, &sname, &sarity);
  SP_get_functor(term, &tname, &tarity);
  if (sname != tname || sarity != tarity) return NA;

  plus = SP_atom_from_string("+");
  for (i = sarity; i > 0; --i) {
    unsigned long t;

    SP_get_arg(i, spec, arg);
    SP_get_atom(arg, &t);       /* no error checking */
    if (t != plus) continue;

    SP_get_arg(i, term, arg);
    switch (SP_term_type(arg)) {
    case SP_TYPE_VARIABLE:
      return NI;
    case SP_TYPE_COMPOUND:
      SP_get_functor(arg, &tname, &tarity);
      SP_put_integer(tmp, (long)tarity);
      SP_cons_list(list, tmp, list);
      SP_put_atom(arg, tname);
      break;
    }
    SP_cons_list(list, arg, list);
    ++ret;
  }
  return ret;
}
```

A C programot elő kell készíteni a Prolog számára az `splfr` eszköz segítségével:

```
splfr ixkeys ixtest.pl +o ixkeys.o
```

6.3. Füzérek (string) kezelése

SICStus Prologban alaphelyzetben egy füzér a karakterei kódjának listájává alakul:

```
"abc" == [97,98,99]
```

Az ISO szabvány a füzérek háromféle értelmezését teszi lehetővé a `double_quotes` jelző értékétől függően (lásd 5.14).

6.4. További hasznos lehetőségek SICStus Prologban

Tetszőleges nagyságú egész számok

pl.:

```
| ?- fakt(100,F).

F = 9332621544394415268169923885626670049071596826438162146859296389521759999322
29915608941463976156518286253697920827223758251185210916864000000000000000000000
0000 ?
```

Gyakoriságmérés

Bekapcsolása (a program fordítása előtt):

```
| ?- prolog_flag(compiling, _, profiledcode).
```

Az ezután lefordított állományokban szereplő eljárásokról különböző fajta statisztikákákat gyűjt. Ezek lekérdezése a következő eljárással történik:

```
profile_data(Állományok, Fajta, Pontosság, Adatok).
```

A

```
profile_reset(Állományok).
```

újrainicializálja a statisztika-számlálókat.

Globális változók (Blackboard)

Az alábbiakban `Kulcs` egy (kis) egész szám vagy atom lehet.

`bb_put(Kulcs, Érték)`
A `Kulcs` kulcs alatt eltárolja `Érték`-et, az előző értéket, ha van, törölve.

`bb_get(Kulcs, Érték)`
Előhívja `Érték`-be a `Kulcs` értékét.

`bb_delete(Kulcs, Érték)`
Előhívja `Érték`-be a `Kulcs` értékét, majd kitörli.

Visszaléptethető módon változtatható kifejezések

`create_mutable(Adat, ValtKif)`

`Adat` kezdőértékkel létrehoz egy új változtatható kifejezést, ez lesz `ValtKif`. `Adat` nem lehet üres változó.

`get_mutable(Adat, ValtKif)`

`Adat`-ba előveszi `ValtKif` pillanatnyi értékét.

`update_mutable(Adat, ValtKif)`

A `ValtKif` változtatható kifejezés új értéke `Adat` lesz. Ez a változtatás visszalépéskor visszacsinálódik. `Adat` nem lehet üres változó.

Takarító eljárás

`call_cleanup(Hivas, Tiszito)`

Jelentése megegyezik `call(Hivas)` jelentésével. Hatása: meghívja `call(Hivas)`-t és ha az véglegesen befejezte futását, meghívja `Tiszito`-t. Egy eljárás akkor fejezte be véglegesen a futását, ha további alternatívák nélkül sikerült, meghiúsult vagy kivételt dobott.

6.5. Fejlett vezérlési lehetőségek SICStusban

6.5.1. Blokk-deklaráció

Példa:

```
:- block p(-, ?, -, ?, ?).
```

Jelentése: ha a `p` eljárás meghívásakor az első és a harmadik argumentum is behelyettesítetlen változó (blokkolási feltétel), akkor a hívást a rendszer felfüggeszti. A hívás csak azután folytatódik, ha a blokkolási feltétel megszűnik, azaz a példában vagy az első, vagy a harmadik argumentum nem-változó behelyettesítést kap. Egy futás végén, ha maradt végre nem hajtott felfüggesztett hívás, akkor azt a rendszer figyelmeztetésként kiírja.

A blokk-deklarációban ugyanarra az eljárásra több feltétel is szerepelhet, ezek vagylagosan értendők. Például a

```
:- block p(-, ?), p(?, -).
```

deklaráció esetén, ha vagy az első, vagy a második argumentum változó, akkor a hívás blokkolódik.

A blokk-deklaráció alkalmazásait az alábbi pontok vázolják.

Végtelen választási pontot létrehozó hívások kiküszöbölése

Listakezelő eljárások, pl. az `append` esetén láttuk, hogy végtelen választási pont jön létre ha nem kellően behelyettesített argumentumokkal hívjuk. Ezt az esetet kizárhatjuk egy megfelelő blokk-deklarációval. Például:

```
:- block sel(?, -, -).
sel(X, [X|L], L).
sel(X, [Y|L], [Y|L1]):-
      sel(X, L, L1).
```

```
:- block app(-, ?, -).
app([], L, L).
app([X|L1], L2, [X|L3]):-
      app(L1, L2, L3).
```

Ezek az eljárások, jelentésüket tekintve azonosak az `append` ill. `select` könyvtári eljárásokkal, de nem hoznak létre végtelen választási pontot.

Programok gyorsítása korutinszervezéssel

A generál-és-ellenőriz típusú programok áltában túl sok visszalépést használnak, hiszen a generáló rész „buta” módon nagyon sok felesleges ágat is bejár.

Például a `megrajzolja_1` eljárás a P27 példában kivárhatatlanul lassú volt, az összes lehetséges gráfpermutációt generálta, majd ellenőrizte, hogy az így kapott gráf egy összefüggő vonalat ír le. Cseréljük fel ebben az eljárásban a generáló és ellenőrző részt:

```
% :- pred megrajzolja_3(graf::in, vonal::out).
megrajzolja_3(G, V):-
      osszefuggo(V), azonos_graf(G, V).
```

Így a tesztelő rész (`osszefuggo`) került előre. Ennek idő előtti végrehajtásának elkerülésére egy blokkdeklarációt kell az `osszefuggo` eljárásra tenni:

```
% osszefuggo(V) = a V vonal összefüggő
% :- pred osszefuggo(graf::in).
:- block osszefuggo(-).
osszefuggo([_]).
osszefuggo([E|V]):-
      csatlakozik(E, V),
      osszefuggo(V).
```

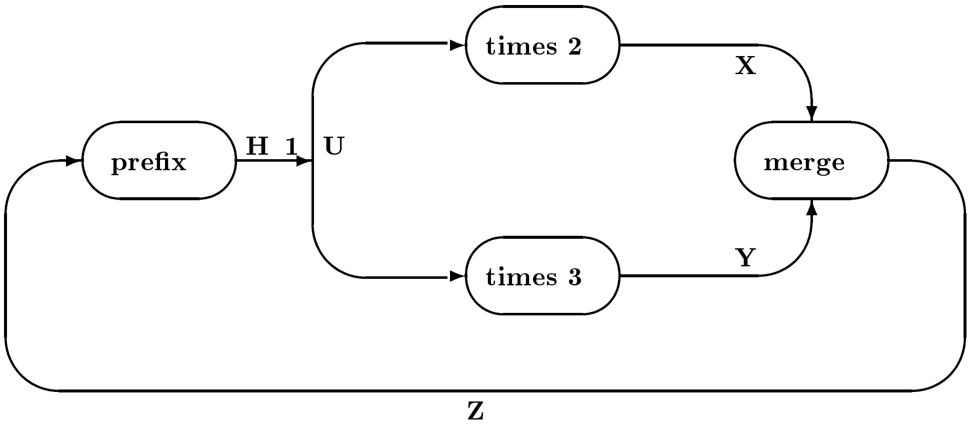
Ezzel a módosítással elérjük, hogy mihelyst a generáló rész a vonal elejét kitölti, annak összefüggő volta azonnal ellenőrződik. Ezzel el tudjuk kerülni a keresési tér felesleges bejárását és a program rövid idő alatt lefut.

Korutinszervezésre épülő programok

P33 Példa: Egyszerűsített Hamming feladat

Tekintsük azokat az 1-nél nagyobb természetes számokat, amelyek csak 2-t és 3-at tartalmazzák prímtényezőként. A feladat az, hogy ezek közül az első `N` darabot nagyság szerint rendezve előállítsuk.

Az alábbi megoldás az ún. „stream-and-parallelism” közelítésmódot használja ki. Az egyes predikátumok adatfeldolgozó egységeknek felelnek meg, a argumentumaikban szereplő változók pedig az ezeket összekapcsoló adatfolyamoknak (amiket Prolog listákkal ábrazolunk). Az alábbi ábra mutatja a példa adatáramlási hálózatát.




```
% A H lista az első N olyan > 1 számot tartalmazza növekvő
% sorrendben, amelyek prímtényezői között csak 2 és 3 szerepel.
hamming(N, H) :-
      U = [1|H], times(U, 2, X), times(U, 3, Y), merge(X, Y, Z),
      prefix(N, Z, H).

% times(X, M, Z): A Z lista az X elemeinek M-szerese
:- block times(-, ?, ?).
times([A|X], M, Z) :-
      B is M*A, Z = [B|U], times(X, M, U).
times([], _, []).

% merge(X, Y, Z): A Z lista az X és Y rendezett listák
% rendezést megőrző összefésülése.
:- block merge(-, ?, ?), merge(?, -, ?).
% Csak akkor fusson, ha az első két argumentum nem változó
merge([A|X], [B|Y], V) :-
      A < B, !, V = [A|Z], merge(X, [B|Y], Z).
merge([A|X], [B|Y], V) :-
      B < A, !, V = [B|Z], merge([A|X], Y, Z).
merge([A|X], [A|Y], [A|Z]) :-
      merge(X, Y, Z).
merge([], X, X) :- !.
merge(_, [], []).

% prefix(N, X, Y): Az X lista első N eleme Y.
prefix(0, _, []) :- !.
prefix(N, [A|X], [A|Y]) :-
      N > 0, N1 is N-1, prefix(N1, X, Y).
```

6.5.2. Korutinszervező eljárások

`freeze(X, Hivas)`

Hivast felfüggeszti mindaddig, amig X behelyettesítetlen változó.

`frozen(X, Hivas)`
Az X változó miatt felfüggesztett hívás(oka)t egyesíti Hivas-sal.

`dif(X, Y)`
X és Y nem egyesíthető. Mindaddig felfüggesztődik, amig ez el nem dönthető.

`call_residue(Hivas, Maradék)`
Hivas-t végrehajtja, és ha a sikeres lefutás után maradnak felfüggesztett hívások, akkor azokat visszaadja Maradékban. Pl.

```
| ?- call_residue(dif(X, f(Y)), Maradek).

Maradek = [[X]-(prolog:dif(X,f(Y)))] ?

yes
| ?- call_residue((dif(X, f(Y)), X=f(Z)), Maradek).

X = f(Z),
Maradek = [[Y,Z]-(prolog:dif(f(Z),f(Y)))] ?

yes
| ?-
```

6.6. SICStus könyvtárak

A SICStus Prolog részét képezi több könyvtár:

- `arrays` Logaritmikus elérési idejű kiterjeszthető tömbök megvalósítását tartalmazza.
- `assoc` AVL fák segítségével valósítja meg az „asszociációs listák”, azaz véges Prolog kifejezéshalmazokon definiált kiterjeszthető leképezések fogalmát.
- `atts` tetszőleges attributumokat enged a Prolog változókhoz rendelni, ezeket tárolórekeszként és a Prolog egyesítési mechanizmusának módosítására is engedi használni.
- `heaps` A bináris kazal (heap) fogalmát valósítja meg, amely főként prioritásos sorok (priority queue) megvalósítására használható.
- `lists` Biztosítja a listakezelő alapműveleteket.
- `terms` Különböző kifejezéskezelő eljárásokat tartalmaz.
- `ordsets` Halmazműveleteket definiál, ahol a halmazokat a Prolog szabványos rendezése szerint (`compare`) rendezett listákkal ábrázolja.
- `queues` Sorokra (queue, FIFO store) vonatkozó műveleteket definiál.
- `random` Egy véletelenszám-generátort tartalmaz.
- `system` Különféle operációsrendszer-szolgáltatások elérését biztosítja.
- `trees` Az `arrays` könyvtárhoz hasonló, de nem-kiterjeszthető logaritmikus elérési idejű tömbfogalmat valósít meg, bináris fák segítségével (kicsit hatékonyabb mint az `arrays` könyvtár).
- `ugraphs` Irányított és irányítatlan gráffogalmat valósít meg, élcimkék nélkül.

- `wgraphs` Olyan irányított és irányítatlan gráffogalmat valósít meg, ahol minden él egy egészértékű súllyal rendelkezik.
- `sockets` A socket-ek kezelésére szolgáló eljárásokat biztosít.
- `linda/client` és `linda/server` Linda-szerű processzkommunikációs eszközöket ad.
- `db` Felhasználó által definiált többszörös indexelést lehetővé tevö, Prolog kifejezések lemezen való tárolására szolgáló adatbázis-rendszer.
- `clpb` Boole-értékekre vonatkozó feltétel-megoldó (constraint solver).
- `clpq` és `clpr` Feltétel-megoldó a Q (racionális számok) ill. R (valós számok) tartományán.
- `clpfd` Véges tartományokra vonatkozó feltétel-megoldó (constraint solver).
- `objects` A logikai és objektum-orientált paradigmák kombinációját biztosítja.
- `gcla` A Prolog ún. GCLA (Generalized Horn Clause Language) általánosításán alapuló specifikációs eszköz.
- `tcltk` A *Tcl/Tk* nyelv és eszközkészlet elérését biztosítja.
- `gauge` Prolog programok a profilirozására szolgáló, a `tcltk` -n alapuló grafikus interfésszel rendelkező eszköz.
- `charsio` Karaktersorozatból olvasó ill. abba író be- és kiviteli eljárások gyüjteménye.
- `flinkage` Segédprogram a külső nyelvi interfész összekötő kódjának generálására.
- `timeout` Lehetőséget ad arra, hogy célok futási idejét korlátozzuk.
- `xref` A nyomkövetés és a program-analízis segítésére használható keresztreferencia készítő program.

Egy könyvtár betöltése az alábbi módon tőrténhet:

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

7. fejezet

Fordítóprogram-írás Prologban

Ez a fejezet egy nagyobb teljes példaprogramot tartalmaz, a [13] cikk alapján.

A példa egy teljes fordítóprogram megvalósítása. A fordítóprogram teljes szövege a hálózatról letölthető:

```
http://www.inf.bme.hu/ dp/prolog/compiler.zip
```

Az alábbiakban, amikor forrás-állományokról beszélünk, akkor ennek a becsomagolt könyvtárnak az állományaira hivatkozunk.

7.1. A forrásnyelv és a célnyelv

Forrásnyelv:

egyszerű Algol-szerű nyelv

- értékadás
- IF utasítás
- WHILE utasítás
- READ és WRITE utasítások

Például a következő forrásprogram szolgálhat a faktoriális kiszámítására:

```
READ value;
count := 1;
result := 1;
WHILE count < value DO
      (count := count+1;
      result := result*count);
WRITE result
```

Célnyelv:

egyszerű egycímes gépi nyelv

- Aritmetikai stb utasitások
  (literális operandussal):

```
ADDC, SUBC, MULC, DIVC, LOADC
```

(memória operandussal):

```
ADD, SUB, MUL, DIV, LOAD, STORE
```

- Ugró utasitások:

```
JUMP, JUMPEQ, JUMPNE, JUMPLT, JUMPGT, JUMPLE, JUMPGE
```

- I/O etc:

```
READ, WRITE, HALT
```

Peldául a fenti programból a következő kód generálódik: (A 3. és 4. oszlop a generált kód, a többi csak a megértést könnyítő megjegyzés.)

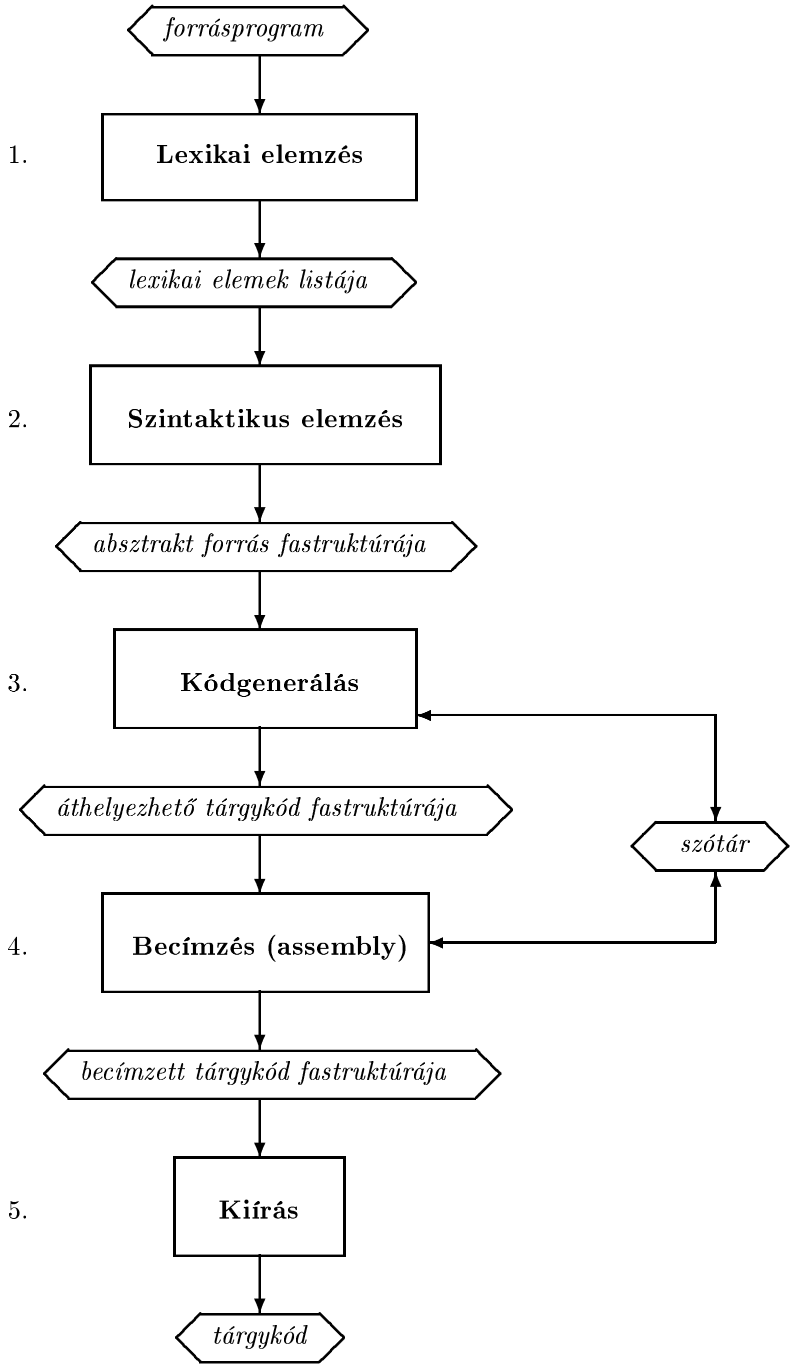
	1:	`READ`	21	value
	2:	`LOADC`	1	
	3:	`STORE`	19	count
	4:	`LOADC`	1	
	5:	`STORE`	20	result
L1	6:	`LOAD`	19	count
	7:	`SUB`	21	value
	8:	`JUMPGE`	16	L2
	9:	`LOAD`	19	count
	10:	`ADDC`	1	
	11:	`STORE`	19	count
	12:	`LOAD`	20	result
	13:	`MUL`	19	count
	14:	`STORE`	20	result
	15:	`JUMP`	6	L1
L2	16:	`LOAD`	20	result
	17:	`WRITE`	0	
	18:	`HALT`	0	
count	19:	`BLOCK`	3	
result	20:			
value	21:			

A forrásnyelv (konkrét) szintaxisa:

```
<program> ::=            <statements>

<statements> ::=         <statement> ; <statements> |
                         <statement>

<statement> ::=          <name> :=  <expr> |
                         if <test> then <statement> else <statement> |
                         while <test> do <statement> |
                         read <name> |
                         write  <expr> |
                         (  <statements> )

<test> ::=               <expr> <comparison_op> <expr>

<expr> ::=               <expr> <op2> <expr1> |
                         <expr1>

<expr1> ::=              <expr1> <op1> <expr0> |
                         <expr0>

<expr0> ::=              <name> | <number> | ( <expr> )

<comparison_op> ::=      = | < | > | =< | >= |\=

<op2> ::=                + | -

<op1> ::=                * | /
```

7.2. A fordítóprogram szerkezete és adatstruktúrái

7.2.1. A fordítás fázisai



7.2.2. A forrásnyelv absztrakt szintaxisa

(azaz adatstruktúra-terve Prologban)

```
<program> ::=            <statement>

<statement> ::=          assign(<name>, <expr>) |
                         if(<test>, <statement>, <statement>) |
                         while(<test>, <statement>) |
                         read( <name> ) |
                         write(<expr>) |
                         <statement>; <statement>

<test> ::=               test(<comparison_op>, <expr>, <expr>)

<expr> ::=               expr(<op>, <expr>, <expr>) |
                         const(<number>) |
                         name(<name>)

<comparison_op> ::=      = | < | > | =< | >= | \=

<op> ::=                 + | - | * | /
```

Ugyanez Mercury-szerű típusdefiníciókkal:

```
:- type program == statement.

:- type statement --->  assign(name, expr)
        ;               if(test, statement, statement)
        ;               while(test, statement)
        ;               read(name)
        ;               write(expr)
        ;               { statement ; statement } .
```

Az előző sorban a kapcsos zárójeleket azért használtuk, hogy a ; adatépítő névként való használatát jelezzük.

```
:- type test --->       test(comparison_op, expr, expr).

:- type expr --->       expr(op, expr, expr)
        ;               const(int)
        ;               name(atom).

:- type comparison_op ---> = ; < ; > ; =< ; >= ; \= .

:- type op --->         + ; - ; * ; / .
```

Példa:

```
while count<value do
      (count := count+1;
      result := result*count)
```

absztrakt alakja

```
while(
      test(<, name(count), name(value)),
      (assign(name(count), expr(+, name(count), const(1))) ;
      assign(name(result), expr(*, name(result), name(count)))
      )
)
```

7.2.3. A (becímzetlen) tárgykód struktúrája

<target> ::=	<instr>
<instr> ::=	`instr(`<mnem>`, `<addr>`)` |
	`label(`<label>`)`|
	`block(`<integer>`)`|
	<instr>; <instr>
<mnem> ::=	`LOAD` | `STORE` | ...
<addr> ::=	<integer>
<label> ::=	<integer>

Ugyanez Mercury-szerű típusdefiníciókkal:

```
:- type target == instr .

:- type instr --->      instr(mnem, addr)
        ;               label(label)
        ;               block(int)
        ;               { instr ; instr } .

:- type mnem --->       load ; store ; ... .

:- type addr == int .

:- type label == int .
```

A kód generálása során az `addr` és `label` címek még nem ismertek, ezért a megfelelő üres változók szerepelnek a kódban.

Példa (a fenti kóddarabnak megfelelő tárgykód):

```
label(L1) ;
      instr(load, CountAddr) ;
      instr(sub, ValueAddr) ;
      instr(jumpge, L2) ;
      instr(load, CountAddr) ;
      instr(addc, 1) ;
      instr(store, CountAddr) ;
      instr(load, ResultAddr) ;
      instr(mul, CountAddr) ;
      instr(store, ResultAddr) ;
      instr(jump, L1) ;
label(L2)
```

ahol `CountAddr, ResultAddr, ValueAddr, L1, L2` behelyettesítetlen változók.

7.2.4. A szótár struktúrája

(rendezett bináris fa)

<dictionary> ::= `dic(`<name>`,` <addr>`,` <dictionary>`,` <dictionary>`)` |
void

(A `dic` struktúra harmadik argumentuma a <name>-nél kisebb elemek részfája, míg a negyedik argumentum a nála nagyobb elemek részfája (bal- ill. jobboldali részfa).)

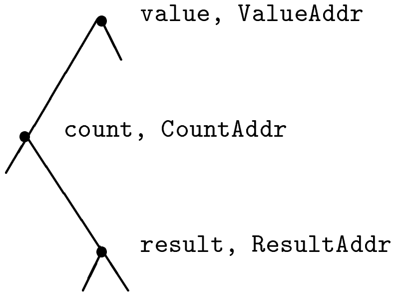
Ugyanez Mercury-szerűen:

```
:- type dictionary ---> dic(name, addr, dictionary, dictionary)
                  ;      void.
```

A kód generálása során az `addr` mezők behelyettesítetlen változók.
Példa (a fenti kóddarabnak megfelelő szótár):

```
      dic(value, ValueAddr,
dic(count, CountAddr, _,
            dic(result, ResultAddr, _, _)), _)
```

A fastruktúra grafikus képe:



Keresés/bevitel a szótárba:

```
% lookup(Name, Dict, Value):  First solution of the goal:
% the (Name, Value) pair is in the dictionary Dict.
% :- pred lookup(name::in, dictionary, addr).

lookup(Name, dic(Name, Value, _, _), Value):-!.
lookup(Name, dic(Name1, _, Before, _), Value):-
      Name @< Name1, lookup(Name, Before, Value).
lookup(Name, dic(Name1, _, _, After), Value):-
      Name @> Name1, lookup(Name, After, Value).
```

Megjegyzés:
Ugyanez a viselkedés megvalósítható egszerűbben, de kevésbé hatékonyan, nyílt végű listák segitségével (a `memberchk` eljárás definícióját lásd a P4 példában):

```
lookup(Name, Dict, Value) :-
      memberchk(Name-Value, Dict).
```

7.3. Kódgenerálás

Absztrakt szintaxisú forrás áthelyezhető tárgykóddá való fordítása.

7.3.1. Értékadás fordítása

Forrás: `assign(name(X), Expr)` X := Expr

Tárgykód: <`Expr` kódja>
`STORE` <`X` címe>

```
% encode_statement(St, Dict, Code):  Code is the translated form of
% statement St with respect to dictionary Dict.
% :- pred encode_statement(statement::in, dictionary, instr::out).

encode_statement(assign(name(X), Expr), Dict,
        (ExprCode;
         instr(store, Addr))
             ):-
      lookup(X, Dict, Addr),
      encode_expr(Expr, Dict, ExprCode).
```

7.3.2. Kifejezések fordítása

```
% encode_expr(Expr, Dict, Code):  Code is the translated form of
% expression Expr with respect to dictionary Dict.
% :- pred encode_expr(expr::in, dictionary, instr::out).

encode_expr(Expr, Dict, Code):-
      encode_subexpr(Expr, 0, Dict, Code).
```

A bevezetett `encode_subexpr` eljárásnak egy további paramétere van, a kifejezés fordításakor nem-használható (már elhasznált) segédváltozók száma.

Az egyes kifejezésfajták fordítása a következő:

Forrás: `const(C)` Tárgykód: `LOADC` <`C` értéke>
Forrás: `name(X)` Tárgykód: `LOAD` <`X` címe>

```
% encode_subexpr(Expr, N, Dict, Code):  Code is the translated form of
% expression Expr with respect to dictionary Dict, with auxiliary
% variables below N not used in the code.
% encode_subexpr(expr::in, int::in, dictionary, instr::out).

encode_subexpr(const(C), _, _, instr(loadc, C)).
encode_subexpr(name(X), _, Dict, instr(load, Addr)):-
      lookup(X, Dict, Addr).
```

Forrás: `expr(Op,Expr1,Expr2)`
Tárgykód: <`Expr1` kódja>
<`Op`-végző utasítás> <`Expr2` vagy `Expr2` címe>
feltéve, hogy `Expr2` egyszerű (szám vagy név)

Példa: `c*x-1` ilyen, de `1-c*x` nem ilyen.

```
encode_subexpr(expr(Op, Expr1, Expr2), N, Dict,
        (Expr1Code;
         Instruction)
            ):-
      apply(Op, Expr2, Dict, Instruction),
      encode_subexpr(Expr1, N, Dict, Expr1Code).


% apply(Op, Expr, Dict, Instruction):  Expr is a simple expression
% such that the operation 'Acc := Acc Op Expr' can be encoded into a
% single instruction Instruction, with respect to dictionary Dict.
% :- pred apply(op::in, expr::in, dictionary, instr::out).

apply(Op, const(C), _, instr(Opcode, C)):-
      literalop(Op, Opcode).
apply(Op, name(Name), Dict, instr(Opcode, Addr)):-
      memoryop(Op, Opcode),
      lookup(Name, Dict, Addr).


% literalop(Op, Mnem):  Mnem is the mnemonic of the instruction
% Acc := Acc Op <Literal Operand>.
% :- pred literalop(op::in, mnem::out).

literalop(+, addc).
literalop(-, subc).
literalop(*, mulc).
literalop(/, divc).


% memoryop(Op, Mnem):  Mnem is the mnemonic of the instruction
% Acc := Acc Op <Memory Operand>.

memoryop(+, add).
memoryop(-, sub).
memoryop(*, mul).
memoryop(/, div).
```

Forrás: `expr(Op,Expr1,Expr2)`
Tárgykód: <Expr2 kódja>
`STORE` <Segédváltozó>
<Expr1 kódja>
<`Op`-végző utasítás> <Segédváltozó>
feltéve, hogy `Expr2` összetett

```
encode_subexpr(expr(Op, Expr1, Expr2), N, Dict,
        (Expr2Code;
         instr(store, Addr);
         Expr1Code;
         instr(Opcode, Addr))
            ):-
      complex(Expr2),
      lookup(N, Dict, Addr),
      encode_subexpr(Expr2, N, Dict, Expr2Code),
      N1 is N+1,
```

```
      encode_subexpr(Expr1, N1, Dict, Expr1Code),
      memoryop(Op, Opcode).


% complex(Expr):  Expr is a complex expression

complex(expr(_,_,_)).
```

7.3.3. Feltételes utasítások fordítása:

Forrás:	`if(Test,Then,Else)`
Tárgykód:	<`Test` kódja>
	<`Then` kódja>
	`JUMP` <cimke_2>
	<cimke_1>:
	<`Else` kódja>
	<cimke_2>:

ahol <`Test` kódja> a <cimke_1>-re ugrik, ha a vizsgálat hamis eredményt ad.

```
encode_statement(if(Test, Then, Else), Dict,
        (TestCode;
         ThenCode;
         instr(jump, L2);
        label(L1);
         ElseCode;
        label(L2))
            ):-
      encode_test(Test, Dict, L1, TestCode),
      encode_statement(Then, Dict, ThenCode),
      encode_statement(Else, Dict, ElseCode).


% encode_test(Test, Dict, Label, Code):  Code is the translated form
% of test Test with respect to dictionary Dict.  Code jumps to Label
% if Test is not satisfied.

encode_test(test(Op, Arg1, Arg2), Dict, Label,
        (ExprCode;
         instr(JumpIf, Label))
         ):-
      encode_expr(expr(-, Arg1, Arg2), Dict, ExprCode),
      unlessop(Op, JumpIf).


% unlessop(Op, Mnem):  Mnem is the mnemonic of the jump instruction
% jumping if the not(Acc Op 0) relation holds.

unlessop(=, jumpne).
unlessop(<, jumpge).
unlessop(>, jumple).
unlessop(\=, jumpeq).
unlessop(=<, jumpgt).
unlessop(>=, jumplt).
```

7.3.4. További utasítások fordítása

```
encode_statement(while(Test, Do), Dict,
        (label(L1);
          TestCode;
          DoCode;
          instr(jump, L1);
         label(L2))
            ):-
      encode_test(Test, Dict, L2, TestCode),
      encode_statement(Do, Dict, DoCode).
encode_statement(read(name(Name)), Dict, instr(read, Addr)):-
      lookup(Name, Dict, Addr).
encode_statement(write(Expr), Dict,
        (ExprCode;
         instr(write, 0))
            ):-
      encode_expr(Expr, Dict, ExprCode).
encode_statement((S1;S2), Dict, (Code1;Code2):-
      encode_statement(S1, Dict, Code1),
      encode_statement(S2, Dict, Code2).
```

7.4. Becímzés

Az áthelyezhető tárgykód és a szótar alapján becímzett tárgykódot kell előállítani, továbbá a szótárban található változóknak címet kell kiosztani és helyet kell foglalni számukra.

```
% compile(Source, Code):  Code is the compiled form of Source.
% :- pred compile(program::in, target::out).

compile(Source,
        (Code;
         instr(halt, 0);
         block(L))
      ):-
      encode_statement(Source, Dict, Code),
      assemble(Code, 1, N0),
      N1 is N0+1,
      allocate(Dict, N1, N),
      L is N-N1.


% assemble(Code, N0, N): Code can be positioned so that it starts at
% location N0 and the first unused location is N.
% :- pred assemble(instr::in, int::in, int::out).

assemble((Code1;Code2), N0, N):-
      assemble(Code1, N0, N1),
      assemble(Code2, N1, N).
assemble(instr(_, _), N0, N):-
      N is N0+1.
assemble(label(N), N, N).
```

```
% allocate(Dict, N0, N): Locations can be assigned to variables in
% Dict in such a way that the first variable is assigned N0, the next
% N0+1, etc, and the first unused location is N.
% :- pred allocate(dictionary, addr::in, addr::out).

allocate(void, N, N):-
      !.
allocate(dic(_Name, N1, Before, After), N0, N):-
      allocate(Before, N0, N1),
      N2 is N1+1,
      allocate(After, N2, N).
```

7.5. Nyelvtani elemzés

Több lehetőség van:

a. a Prolog elemző használata, megfelelő operátordeklarációkkal — gyors, de rugalmatlan (lásd a `parse_op.pl` forrás-állományt),

b. saját elemző írása a Definite Clause Grammar (DCG) formalizmust használva (lásd a `parse_dg.pl` forrás-állományt).

7.5.1. A Prolog elemzőre épülő nyelvtani elemző

Ez az út, gyors prototípuskészítésre ajánlható. Korlátai: csak a Prolog operátoros kifejezések szintaxisába beleilleszkedő nyelvre jó, (pl. `a[i]` tömbkifejezés ebbe nem fér bele, de apró kényelmetlenség árán, pl. `a^[i]`-ként „bepréselhető”).

A fenti példában lényegében problémamentesen használható.

```
:-op(990, xfx, :=).
:-op(995, fy, if).
:-op(995, xfy, then).
:-op(995, xfy, else).
:-op(995, fy, while).
:-op(995, xfy, do).
:-op(995, fx, read).
:-op(995, fx, write).
```

operátordeklarációk bevezetése után a forrásprogramok Prolog kifejezésekként beolvashatók:

```
read_program(Prog):-
      read(Term),
      parse_statement(Term, Prog).
```

`parse_statement(Term, Prog)` a Prolog kifejezést alakítja át a korábban specifikált absztrakt forrásalakra, pl.:

```
parse_statement((V:=E), assign(name(V), Expr)):-
      parse_expr(E, Expr).
```

Az elemző teljes szövege a `parse_op.pl` forrás-állományban található.

Példa egy apró kényelmetlenségre ezen módszer használatakor:

```
if a=1 then if b=1 then c:=1 else c:=2 else c:=3
```

ezzel a módszerrel nem elemezhető, helyette

```
if a=1 then (if b=1 then c:=1 else c:=2) else c:=3
```

irandó. Ennek az az oka, hogy az `if`, `then` és `else` operátorok azonos prioritásúak, így a skatulyázást a Prolog elemző nem tudja felismerni.

7.5.2. Definite Clause Grammars

Célszerű a lexikai elemzést a nyelvtani elemzésről leválasztani. Ehhez feltételezünk egy `read_tokens(L)` eljárást, amely lexikai elemek listajává alakitja az input-folyamot (lásd az `rdtok.pl` forrás-állományt).

Egy ilyen input-folyam-változó kezelésével könnyen írható elemző program Prologban a korábban ismertetetett DCG szabályok segítségével.

Tekintsük az elemzendő nyelv szintaxisát:

<statement> ::=	<name> `:=` <expr> |
	`if` <test> `then` <statement> `else` <statement> |
	...

Ezek a szintaktikus szabályok nagyon egyszerűen átalakíthatók a lexikai elemek listáját elemző DCG szabályokká:

```
statement -->
      name, [:=], expr.
statement -->
      [if], test, [then], statement, [else], statement.
...
```

Az elemző program könnyen kiegészíthető faépitő argumentumokkal:

```
statement(assign(V,Expr)) -->
      name(V),[:=],expr(Expr).
statement(if(Test,Then,Else)) -->
      [if], test(Test), [then], statement(Then), [else], statement(Else).
```

A DCG elemző teljes szövege a `parse_dg.pl` forrás-állományban található.

7.6. Lexikai elemzés

Egy egyszerű lexikai elemző található az `rdtok.pl` forrás-állományban.

7.7. Kiírás

A kiirató modul az `output.pl` forrás-állományban található.

8. fejezet

Új irányzatok a logikai programozásban

8.1. Párhuzamos megvalósítások

Két irányzat:

- explicit párhuzamosság (a programozó vezérlésével), pl.

Parlog	(Clark, Gregory, Imperial College, Anglia)
Concurrent Prolog	(Shapiro, Weizmann Inst, Izrael)
FGHC	(Ueda, ICOT, Japán)
CS-Prolog	(Futó Iván, ML, Magyarország)

  Az első három ún. *committed choice* nyelv, csak az ún. *don't care* nemdeterminizmust valósítják meg.

- implicit párhuzamosság (a programozó „háta mögötti párhuzamosítás") ugyanazt a programot futtatjuk multiprocesszoron, mint monoprocesszoron, csak a program *gyorsabban* fut, pl.

Aurora	(Warren-Lusk-Haridi, Gigalips projekt)
Muse	(Ali, SICS, Svédo.)
Andorra-I	(Warren, Bristol, Anglia)
&-Prolog	(Hermenegildo, UPM, Madrid)

Példa:

```
furdoszoba(Szin, Mosdo, Kad):-
      mosdo(Szin, Mosdo),
      kad(Szin, Kad).

szinvalasztek(feher).
szinvalasztek(bezs).
...

mosdo(feher, ...).
...

kad(feher, ...).
...
```

A párhuzamosság fajtái:

- vagy-párhuzamosság (or-parallelism):

```
?- szinvalasztek(Szin), furdoszoba(Szin, Mosdo, Kad).
```

A különböző `Szin`-behelyettesítésekhez tartozó kereséseket külön processzorok végez(het)ik.

- és-párhuzamosság (and-parallelism)
  - független (independent and-parallelism):

    ```
    ?- furdoszoba(feher, Mosdo, Kad).
    ```

    Az ebből keletkező: `mosdo(feher, Mosdo)` és `kad(feher, Kad)` hívások egymástól függetlenül párhuzamosan futhatnak, az egyik minden megoldását a másik mindegyikével párosítani kell.
  - függő (dependent and-parallelism):

    ```
    ?- furdoszoba(Szin, Mosdo, Kad).
    ```

    Az alapvető nehézséget a párhuzamos és-futások közben létrejött választási pontok jelentik. A 80-as évek elején/közepén született committed choice nyelvek ezt a problémát a don't care nemdeterminizmust segítségével oldották meg, lényegében a visszalépés kiküszöbölésével, pl. a

    ```
    mosdo(Szin, M), kad(Szin, K)
    ```

    hívásban, mind a mosdó, mind a kad eljárás csak úgy helyettesítheti be `Szin`-t, ha „elkötelezí' magát a behelyettesítés mellett.
    Újabb megvalósításokban nem zárják ki a nemdeterminizmust, lásd pl. Andorra-I.

8.2. Az Andorra-I rendszer rövid bemutatása

Vagy- és (függő) és-párhuzamos rendszer a *teljes* Prolog támogatásával, részben az Aurorá-ra épül.

Egy bonyolult, fordításidejű programanalizátort tartalmaz, annak felderítésére, hogy mely eljárások milyen paraméterezés mellett lesznek determinisztikusak.

A Prologtól némileg eltérő végrehajtási mechanizmusa van, az ún. Basic Andorra Model:

8.2.1. Basic Andorra Model

Adott egy $g_1, ..., g_n$ célsorozat:

1. Kiválasztjuk azokat a g_i célokat, amelyek legfeljebb egy klózzal illeszthetők (determinisztikus vagy meghiúsuló hívások)
2. Redukáljuk a kiválasztott g_i célokat (azaz helyettesítjük őket a megfelelő klóztörzzsel). Ezek a redukciós lépések párhuzamosan (egyidejűleg) végezhetők.
3. Amíg az 1. lépésben ki tudunk választani részcélokat, ismételjük az 1. és 2. lépéseket. A végrehajtás ezen részét *determinisztikus szakasznak* hívjuk.
4. Ha nincsenek determinisztikus vagy meghiusuló részcélok, akkor válasszuk ki a baloldali részcélt, illesszük a lehetséges klózokkal, és mindegyik választásra végezzük el a megfelelő redukciós lépést. Ezek az alternatív redukciók párhuzamosan végezhetők. Ez a végrehajtási fázis az ún. *nemdeterminisztikus szakasz*. Ezután folytassuk az 1. pontnál az összes alternatív célsorozatra párhuzamosan.

Egy egyszerű példa a modell alkalmazására:

```
ertelme([],[]).
ertelme([Szo0|Szavak0], [Szo|Szavak]):-
      elirt_szo(Szo0, Szo),
      szotar(Szo),
      ertelme(Szavak0, Szavak).
```

```
elirt_szo([], []).
elirt_szo([Betu0|Betuk0], [Betu|Betuk]):-
      elirhato(Betu, Betu0),
      elirt_szo(Betuk0, Betuk).

elirhato(a, o).
elirhato(l, t).
elirhato(t, l).
elirhato(B, B).

szotar([a]).
...
szotar([t,a,t,a]).
...
szotar([e,l,m,e,n,t]).
```

Futása:

```
:-ertelme([[a],[l,o,l,a],[e,t,m,e,n,t]],Mondat).
```

... < csupa determinisztikus redukció >

```
:- Mondat = [[A], [L1,O,L2,A2], [E1,T1,M,E2,N,T2]],
   elirhato(A, a),
   elirhato(L1, l), elirhato(O, o), elirhato(L2, l), elirhato(A2, a),
   elirhato(E1, e), elirhato(T1, t), elirhato(M, m), elirhato(E2, e),
   elirhato(N, n), elirhato(T2, t),
   szotar([A]), szotar([L1,O,L2,A2]), szotar([E1,T1,M,E2,N,T2]).
```

... < még mindig csupa determinisztikus redukció >

```
:- Mondat = [[a], [L1,O,L2,a], [e,T1,m,e,n,T2]],
   elirhato(L1, l), elirhato(O, o), elirhato(L2, l),
   elirhato(T1, t), elirhato(T2, t),
   szotar([a]), szotar([L1,O,L2,a]), szotar([e,T1,m,e,n,T2]).
```

... < ezen a ponton a `szotar` hívások egyike válhat determinisztikussá
ha nem, akkor egy nemdeterminisztikus szakasszal folytatjuk >
...

8.2.2. Egy egyszerű interpreter az Andorra alapmodellre

Korlátai

- a beépítettek mind nemdeterminisztikusak
- a vágót nem kezeli
- csak a fejillesztés alapján dönt a determinisztikusságról

Sebessége

- kb. 2 nagyságrenddel lassítja a „rendes” programokat
- ennek ellenére kb. 2 nagyságrenddel gyorsít bizonyos generál-és-ellenőriz típusú programokat

Futási idők

	megrajzolja_1		megrajzolja_2	
	Hívások	Idő	Hívások	Idő
Prolog	32,558,297	214.47s	3,964	0.03s
Prolog interp.	689,561,949	8440.39s	68,046	0.80s
Andorra interp.	271,392	3.64s	202,510	2.76s

8.2.3. Az Andorra interpreter

```
% The Goals goal sequence reduces to "true" in the basic Andorra model
solve(Goals):-
        add_body(Goals, [], Gs), solve_andorra(Gs).

% The Gs0 goal list reduces to "true" in the basic Andorra model
solve_andorra(Gs0):-
        solve_det(Gs0, Gs1),
        (   Gs1 = [] -> true
        ;   nondet_reduction(Gs1, Gs2),
            solve_andorra(Gs2)
        ).

% nondet_reduction(Gs0, Gs): Gs is the result of a nondeterministic
% reduction on the nonempty goal list Gs0.
nondet_reduction([G|Gs], Gs):-
        predicate_property(G, built_in), !,
        call(G).
nondet_reduction([G|Gs0], Gs):-
        clause(G, B), add_body(B, Gs0, Gs).

% add_body(B, Gs0, Gs): Gs is the list of goals
% in body B with Gs0 appended.
add_body(true, Gs, Gs):- !.
add_body(fail, _, _):- !, fail.
add_body((G1,G2), Gs0, Gs):-
        !, add_body(G2, Gs0, Gs1),
        add_body(G1, Gs1, Gs).
add_body(G, Gs, [G|Gs]).

% If one defined solve_det as an identity function:
% solve_det(Gs, Gs).
% the above interpreter would simulate normal Prolog.
% The result of deterministic phase reduction of Gs0 is Gs.
solve_det(Gs0, Gs):-
        solve_det1(Gs0, Gs1),
        (    Gs0 = Gs1 -> Gs = Gs1
        ;    solve_det(Gs1, Gs)
        ).

% solve_det1(+Gs, -RedGs): performing a deterministic
% reduction step on all semidet subgoals of the Gs
% goal list yields RedGs.
solve_det1([], []).
solve_det1([G|GL], Gs):-
```

```
        semidet(G, B), !,
        solve_det1(GL, Gs1),
        add_body(B, Gs1, Gs).
solve_det1([G|GL], [G|Gs]):-
        solve_det1(GL, Gs).

% Goal G matches at most one clause head.
% B is the body of this clause or 'fail'.
semidet(G, B):-                               % builtins are
        \+ predicate_property(G, built_in),  % considered nondet
        findall(one, clause(G, _B), L),
        (   L=[one] -> clause(G, B), !
        ;   L=[] -> B=fail
        ).
```

8.3. A Mercury nagyhatékonyságú LP megvalósítás

Célok, alapelvek

- Tiszta deklaratív logikai programozás
- Nagy, megbízható, hatékony valódi alkalmazások létrehozásának támogatása
- A programozási munka hatékonyságának növelése

Fő tulajdonságok

- erős polimorfikus típus-rendszer

```
:- type tree(K,V) ---> empty ; node(K,V,tree(K,V),tree(K,V)).
:- pred lookup(tree(K, V), K, V).
```

- erős mód-rendszer (argumentumok be- és kimenő voltának jelzése)

```
:- mode lookup(in, in, out).
```

- erős determinizmus-rendszer

```
:- mode lookup(in, in, out) is semidet.
```

det	pontosan egy megoldás
semidet	legfeljebb egy megoldás
multi	legalabb egy megoldás
nondet	akárhány megoldás

- modul-rendszer
- magasabbrendű szerkezetek (`solutions` hasonló `bagof`-hoz)
- teljesen deklaratív (még a be- és kivitel is)

P34 Példa: File-név-illesztő program Mercury nyelven

A feladat olyan Mercury program elkészítése, amely az operációs rendszerek file-név-illesztéséhez hasonló funkciót valósít meg.

Illesztéskor azt kell eldönteni, hogy egy *minta* mikor illeszthető egy adott karaktersorozatra. A minta karaktereinek értelmezése a következő:

- A `?` egy tetszőleges karakterrel illeszthető.
- A `*` egy tetszőleges (esetleg üres) karakter-sorozattal illeszthető.
- A *c* karakter-pár a *c* karakterrel illeszthető (ha egy minta \\-re végződik, az illesztés meghiúsul).
- Bármely más karakter csak önmagával illeszthető.

A Mercury program hívási formája:

```
match Pattern1 Name Pattern2
```

A `Pattern1` és `Pattern2` argumentumok a fenti értelemben vett minták, és bennük a `*` és `?` karaktereknek azonos számban és azonos elrendezésben kell előfordulniuk. (Pontosabban: ha a `Pattern1` és `Pattern2` argumentumokból kihagyjuk az összes `*`-tól és `?`-től különböző karaktert, akkor két azonos jelláncot kell kapnunk.)

A program funkciója a következő: A `Pattern1` mintára (az összes lehetséges módon) illeszti a `Name` nevet, a `*` és `?` karakterek helyébe kerülő szövegeket a `Pattern2` mintába rendre behelyettesíti, és az így kapott neveket kiírja.

A program listája:

```
:- module match.
/*-----------------------------------------------------*/
:- interface.

:- import_module io.
:- pred main(io__state::di, io__state::uo) is det.

/*-----------------------------------------------------*/
:- implementation.
:- import_module list, int, std_util, string.

main -->
    io__command_line_arguments(Args),
    (
        { Args = [P1,N1,P2] } ->
        { solutions( match(P1, N1, P2), Sols) },
        io__write_string("Pattern '"), io__write_string(P1),
        io__write_string("' matches '"), io__write_string(N1),
        io__write_string("' as '"), io__write_string(P2),
        io__write_string("' matches the following:\n\n"),
        write_sols(Sols)
    ;
        io__write_string("Usage: match <p1> <n1> <p2>\n")
    ).

:- pred write_sols(list(string)::in, io__state::di,
            io__state::uo) is det.
write_sols([])-->
    io__write_string("\n*** No (more) solutions\n\n").
write_sols([S|Ss])-->
    io__write_string("    "), io__write_string(S),
    io__write_string("\n"), write_sols(Ss).
:- pred match(string::in, string::in, string::in,
```

```
      string::out) is nondet.
match(Pattern1, Name1, Pattern2, Name2):-
    string__to_char_list(Pattern1, Ps1),
    string__to_char_list(Name1, Cs1),
    string__to_char_list(Pattern2, Ps2),
    match_list(Ps1, Cs1, L),
    match_list(Ps2, Cs2, L),
    string__from_char_list(Cs2, Name2).

:- type subst ---> any(list(char))  ;  one(char).

:- pred match_list(list(char), list(char), list(subst)).
:- mode match_list(in, in, out) is nondet.
:- mode match_list(in, out, in) is nondet.
match_list([], [], []).
match_list([?|Ps], Cs, [one(X)|L]):-
    Cs = [X|Cs1],
    match_list(Ps, Cs1, L).
match_list([*|Ps], Cs, [any(X)|L]):-
    list__append(X, Cs1, Cs),
    match_list(Ps, Cs1, L).
match_list([\, C|Ps],  [C|Cs], L):-
    match_list(Ps, Cs, L).
match_list([C|Ps], [C|Cs], L):-
    C \= (*), C \= (?), C \= (\),
    match_list(Ps, Cs, L).
```

A Mercury példaprogram fordítása és futása

```
> ls -l match.*
-rw-r--r--   1 szeredi  group        1908 Nov 14 12:22 match.m
> mmake match.depend
mc --generate-dependencies  match.m
rm -f match_init.c
> mmake match
rm -f match.c
mc --compile-to-c -s asm_fast.gc  match.m > match.err2 2>&1
mgnuc -sasm_fast.gc  -c match.c -o match.o
c2init  match.m > match_init.c
mgnuc -sasm_fast.gc  -c match_init.c -o match_init.o
ml -s asm_fast.gc  -o match match_init.o match.o
> match
Usage: match <p1> <n1> <p2>
> match '*a*b*' abbabab '*-*-*'
Pattern '*a*b*' matches 'abbabab' as '*-*-*' matches the following:

    --babab
    -b-abab
    -bba-ab
    -bbaba-
    abb--ab
    abb-ba-
    abbab--
*** No (more) solutions
```

```
> ls -l match*
-rwxr-xr-x   1 szeredi  group      311300 Nov 14 12:28 match*
-rw-r--r--   1 szeredi  group       47270 Nov 14 12:28 match.c
-rw-r--r--   1 szeredi  group         426 Nov 14 12:28 match.d
-r--r--r--   1 szeredi  group        1518 Nov 14 12:28 match.dep
-rw-r--r--   1 szeredi  group           0 Nov 14 12:28 match.err2
-rw-r--r--   1 szeredi  group        1908 Nov 14 12:22 match.m
-rw-r--r--   1 szeredi  group       15254 Nov 14 12:28 match.o
```

8.4. CLP (Constraint Logic Programming)

A Prolog egyesités/mintaillesztés művelete helyébe egy általánosabb feltételrendszer-megoldó kerül.

	LP		CLP
	egyesítés	⟶	constraint solving
állapot:	célsorozat	⟶	célsorozat □ konzisztens feltétel-halmaz
klóztörzsben:	célok	⟶	célok vagy (forrás-)feltételek

Példák feltételekre (constraint-ekre)

```
X in 1..10
X #< Y + 2
```

Kétféle redukciós lépés

- cél helyettesítése klóztörzzsel
- forrás-feltétel átvitele a feltétel-halmazba (esetleg csak figyelembevétele)

Két alapvető közelítésmód

- teljes konzisztencia (pl. CLP(R))
- részleges konzisztencia (pl. clp_fd)

Constraint rendszerek tartományai:

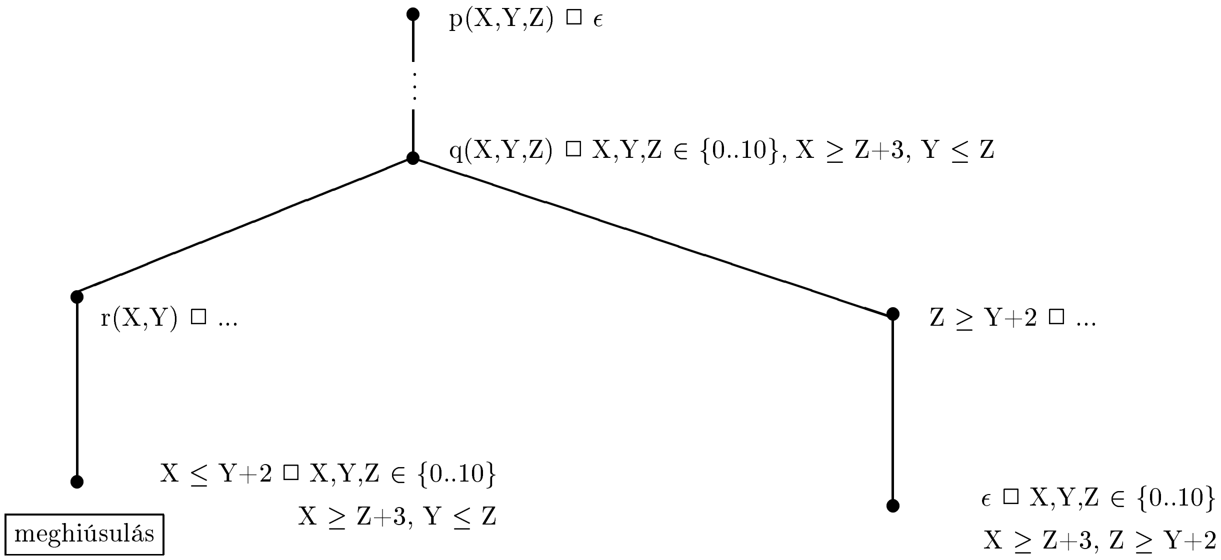
- logikai értékek
- valósak
- racionálisok
- véges tartományok (nem-negatív egészek)
- intervallumok
- ...

8.4.1. CLP végrehajtási mechanizmus

P35 Példa:

```
p(X, Y, Z):-
      X,Y,Z ∈ {0..10}
      X ≥ Z+3, Y ≤ Z, q(X, Y, Z).

q(X, Y, Z):-
      r(X, Y).
q(X, Y, Z):-
      Z ≥ Y+2.

r(X, Y):-
      X ≤ Y+2.
```

A példa végrehajtási fája



8.4.2. Példa-párbeszéd a SICStus clpr kiterjesztésével

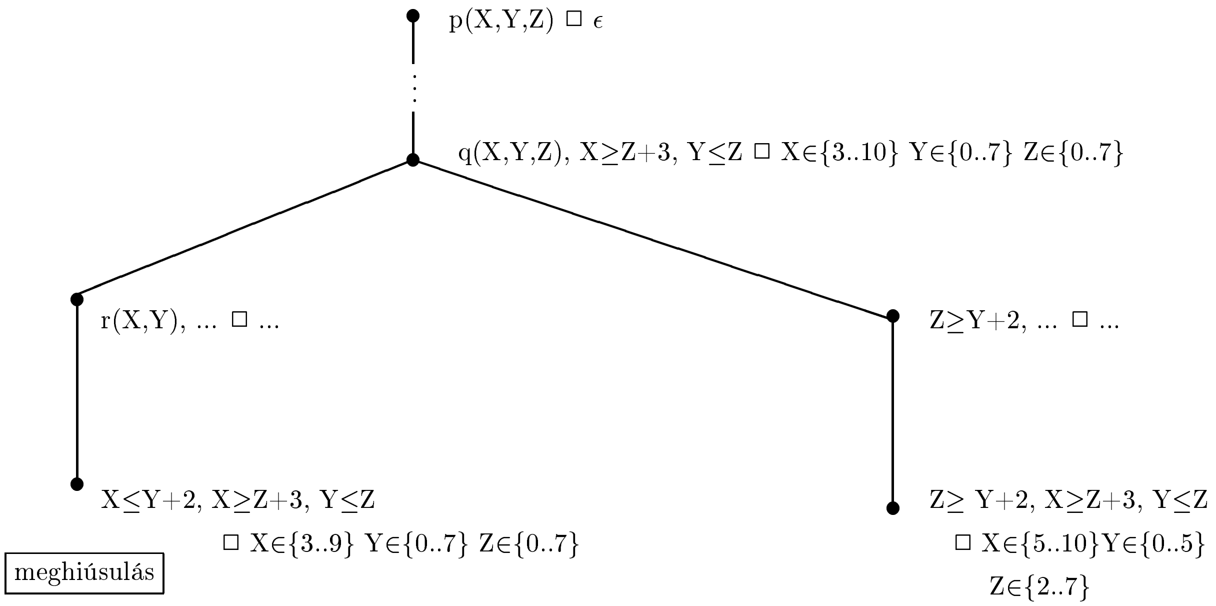
```
| ?- use_module(library(clpr)).
{loading /usr/local/lib/sicstus/library/clpr.ql...}
...
{loaded /usr/local/lib/sicstus/library/clpr.ql in module linear, 750 msec 670816 bytes}

yes
| ?- [user].
| portray(X):-
     number(X), X =:= integer(X), !, format('~d', [X]).
                                          % egészek tizedes rész nélkül,
| portray(X):-
     float(X), format('~4f', [X]).        % a lebegőpontosak 4 tizedessel nyomtatódnak.
| {user consulted, 0 msec 16 bytes}

yes
| ?- {X = Y + 4, Y = Z - 1, Z = 2}.       % lineáris egyenletek

X = 5,
```

```
Y = 1,
Z = 2 ?

yes
{X = Y + 4, Y = Z - 1, Z = 2*X - 9}.

X = 6,
Y = 2,
Z = 3 ?

yes
| ?- {X+Y+9 < 4*Z, 2*X =Y+2, 2*X+4*Z = 36}% az = reláció mellett
                                          % < stb.  is használható
{X=1.0000+0.5000*Y},
{Z=8.5000-0.2500*Y},
{Y<9.6000} ?

yes
| ?- {(Y+X)*(X+Y)/X = Y*Y/X+100}.         % lineárissá egyszerüsíthető

{X=100-2*Y} ?

yes
| ?- {(Y+X)*(X+Y) = Y*Y+100*X}.           % így már nem...

nonlin:{2*(X*Y)-100*X+X^2=0} ?

yes
| ?- [user].                              % komplex számok szorzása
| zmul(c(R1, I1), c(R2, I2), c(R3, I3)):-
          {R3 = R1 * R2 - I1 * I2, I3 = R1 * I2 + R2 * I1}.
| {user consulted, 0 msec 400 bytes}

yes
| ?- zmul(c(1, 1), c(2, 2), Z).

Z = c(0,4) ?

yes
| ?- zmul(c(1, 1), Y, c(0, 4)).           % osztásként is működik

Y = c(2,2) ?

yes
| ?- zmul(X,X,c(0,I)).                    % a nem-lineáris egyenleteket nem oldja meg

X = c(_A,_B),
nonlin:{-(_A^2)+_B^2=0},
nonlin:{I-2*(_B*_A)=0} ?

yes
| ?- zmul(X,X,c(0,I)), X = c(R,1).        % ha egy egyenlet lineárissá válik,
                                          % akkor azt megoldja
```

```
X = c(R,1),
{I=2*R},
nonlin:{1-R^2=0} ?

yes
| ?- [user].
% hiteltörlesztés számítása
% P hitelt Time hónapon át évi IntRate kamat mellett
% havi MP részletekben törlesztve Bal a maradványösszeg
| mortgage(P, Time, IntRate, Bal, MP):-
             {Time > 0, Time =< 1,
             Bal = P * (1 + Time * IntRate / 1200) - Time * MP}.
| mortgage(P, Time, IntRate, Bal, MP):-
             {Time > 1},
             mortgage(P * (1 + IntRate / 1200) - MP, Time-1, IntRate, Bal, MP).
| {user consulted, 20 msec 160 bytes}

yes
| ?- mortgage(100000,180,12,0,MP).       % 100000 Ft hitelt 180 hónapon
                                         % keresztül teljesen törleszt 12%-os
                                         % kamat mellett, mennyi a havi részlet?
MP = 1200.1681 ?

yes
| ?- mortgage(P,180,12,0,1200).          % ugyanez visszafelé

P = 99985.9968 ?

yes
| ?- mortgage(100000,Time,12,0,1300).    % 1300 Ft törlesztőrészlet esetén
                                         % mennyi a törlesztési idő?
Time = 147.3645 ?

yes
| ?- mortgage(P,180,12,Bal,MP).          % Mi az összefüggés a hitelösszeg (P),
                                         % s maradványösszeg (Bal) és a havi részlet (MP)
                                         % között 180 havi 12%-os hitel esetén?

{MP=0.0120*P-0.0020*Bal} ?

yes
| ?- mortgage(P,180,12,Bal,MP), ordering([P,Bal,MP]).
                                         % Ugyanaz, csak P-t fejezzük ki.

{P=0.1668*Bal+83.3217*MP} ?

yes
```

8.4.3. CLP végrehajtás véges tartományokon alapuló rendszerekben

A 8.4.1-beli példa végrehajtása



8.4.4. Példa-párbeszéd a SICStus clpfd kiterjesztésével

```
| ?- use_module(library(clpfd)).
{loading /usr/local/lib/sicstus/library/clpfd.ql...}
...
{loaded /usr/local/lib/sicstus/library/clpfd.ql in module clpfd, 650 msec 379344 bytes}

yes
| ?- X #= Y+4, Y #= Z - 1, Z #= 2.

X = 5,
Y = 1,
Z = 2 ?

yes
| ?- X #= Y + 4, Y #= Z - 1, Z #= 2*X - 9.

X in inf..sup,
Y in inf..sup,
Z in inf..sup ?                        % megszorítás nélkül nem képes következtetni.

yes
| ?- X #= Y + 4, Y #= Z - 1, Z #= 2*X - 9, domain([X,Y,Z], 0, 10000000).

X = 6,
Y = 2,
Z = 3 ?

yes
```

```
| ?- X+Y+9 #< 4*Z, 2*X #=Y+2, 2*X+4*Z #= 36, domain([X,Y,Z], 0, 1000000).

Y in 2..10,
Z in 2..18,
X in 4..8 ?                              % csak közelítő következtetést végez.

yes
| ?- X+Y+9 #< 4*Z, 2*X #=Y+2, 2*X+4*Z #= 36, domain([X,Y,Z], 0, 1000000), indomain(X).
                                         % A változó-értékeket az indomain/1 ill.
                                         % labeling/2 eljárásokkal fel kell soroltatni.

X = 2,
Y = 2,
Z = 8 ?  ;

X = 4,
Y = 6,
Z = 7 ?  ;

no
```

8.4.5. Két egyszerű clpfd példaprogram

P36 Példa: SEND + MORE = MONEY feladvány

```
penzkuldes(SEND, MORE, MONEY):-
      LD = [S,E,N,D,M,O,R,Y],
      domain(LD, 0, 9), all_different(LD),
      S #> 0, M #> 0,
      SEND #= 1000*S+100*E+10*N+D,
      MORE #= 1000*M+100*O+10*R+E,
      MONEY #= 10000*M+1000*O+100*N+10*E+Y,
      SEND+MORE #= MONEY,
      labeling([], LD).
```

P37 Példa: Királynők a sakktáblán

```
% A Qs lista N királynő biztonságos elhelyezését
% mutatja egy N*N-es sakktáblán.  Ha a lista i.  eleme j,
% akkor az i.  királynőt az i.  sor j.  oszlopába kell
% helyezni.
queens(N, Qs):-
      length(Qs, N), domain(Qs, 1, N),
      safe(Qs),
      labeling([ff],Qs).                 % first-fail principle

% safe(Qs):  A Qs lista a királynők biztonságos
% elhelyezését írja le.
safe([]).
safe([Q|Qs]):-
      no_attack(Qs, Q, 1),
      safe(Qs).
```

```
% no_attack(Qs, Q, I): A Qs lista által leírt
% királynők egyike sem támadja a Q oszlopban levő
% kiralynőt, feltéve hogy Q és Qs távolsága I.
no_attack([],_,_).
no_attack([X|Xs], Y, I):-
      no_threat(X, Y, I),
      I1 is I+1, no_attack(Xs, Y, I1).

% Az X és Y oszlopokban I sortávolságra levő
% királynők nem támadják egymást.
no_threat(X, Y, I) :-
      Y #= X, Y #= X-I, Y #= X+I.
```

Hatékonyabb megoldás ún. indexikálisok használatával:

```
no_threat(X, Y, I) +:
      X in -({Y}  {Y+I}  {Y-I}),
      Y in -({X}  {X+I}  {X-I}).
```

A függelék

Fogalom-tár

argumentum (argument)

Lásd struktúra.

argumentumszám, aritás (arity)

Egy eljárás vagy egy struktúra argumentumainak a száma.

atom (atom)

Egy névkonstans. A kisbetűvel kezdődő és alfanumerikus karakterekkel folytatódó sorozatot, a csupa „speciális” karakterből álló sorozatot és az aposztrófok közé zárt tetszőleges sorozatot atomnak tekinti a rendszer. Az atomok egyesítése gyors, ábrázolásuk tömör, viszont drága a létrehozásuk és szétszedésük.

atomi kifejezés (atomic term)

Lásd konstans.

behelyettesítés (binding)

Amikor egy változót egyesítünk valamivel (akár egy másik változóval is), akkor azt mondjuk, hogy a változót behelyettesítettük.

beépített eljárás (built-in predicate)

Egy rendszer által definiált reláció.

cél (goal)

Egy eljáráshívás. Egy cél vagy sikerül, vagy meghiúsul. Egy cél többféleképpen is sikerülhet.

deklaráció (declaration)

A fordítónak szóló állítások, amelyek a program értelmezését befolyásolják. Példa: operátor-deklaráció.

determinisztikus (determinate)

Egy hívás determinisztikus, ha legfeljebb egyféleképpen sikerülhet. Egy eljárás determinisztikus, ha tetszőleges hívása determinisztikus.

dinamikus eljárás (dynamic predicate)

Olyan eljárás, amelyhez futási időben adhatunk és amelyből futási időben vehetünk el klózokat.

diszjunkció (disjunction)

Vagy kapcsolat. Prologban a vagy kapcsolatban levő relációkat pontosvesszővel elválasztva egymás után írjuk, és általában zárójelekkel vesszük körbe.

egyesítés (unification)

Kifejezések mintaillesztéssel történő azonos alakra hozása.

egyhosszúságú atom (one-char atom)

Egyetlen jelből álló atom. Ezeket nevezzük karakternek.

egyszerű kifejezés (simple term)

Egy nem struktúra kifejezés, azaz változó vagy konstans.

eljárás-doboz (procedure box)

A Prolog programok végrehajtásának leírására használt (fogalmi) segédeszköz.

eljárás (procedure)

Egy beépített vagy egy felhasználó által definiált eljárás.

előfordulás-ellenőrzés (occurs-check)

Egy változó és egy struktúra egyesítésekor annak ellenőrzése, hogy a változó nem szerepel-e a struktúrában.

exportál (export)

Egy modul azokat az eljárásokat exportálja, amelyeket a modulon kívülről is elérhetővé akar tenni.

értékadás (instantiation)

Amikor egy változót egy nem változóval egyesítünk, akkor az értéket kap. (Az értékadás lehet részleges is, amennyiben a kifejezés, amivel egyesítettünk nem tömör.)

értékkel bíró (instantiated)

Egy változó értékkel bír, ha egyesítettük egy nem változó kifejezéssel.

értékkel nem bíró (unbound)

Egy változó, amit még soha nem egyesítettünk változótól különböző kifejezéssel.

fej (head)

Egy klóz következmény része.

felhasználó által definiált eljárás (user defined procedure)

Mindazon klózok összessége, amelyek feje adott nevű és argumentumszámú.

fordítás (compile)

Az a folyamat, amikor a program szövegéből lefordított alakot állítunk elő. A lefordított alak gyorsabban fut, mint az interpretált alak.

funktor (functor)

Egy struktúra nevéből és argumentumszámából képzett pár. Írásban általában `/` jellel elválasztva írjuk. Például a `rendező('Luis Buñuel', 1900, 1983)` struktúra funktora `rendező`/3.

füzér (string)

Karakterkódok listája.

importál (import)

Ha fel akarunk használni egy (másik) modulban definiált eljárást, akkor a használat előtt azt importálni kell.

indexelés (indexing)

Arra szolgál, hogy egy adott hívás esetén a rendszer ne próbálkozzon olyan klózokkal, amelyek semmiképp nem sikerülhetnek. Az indexelést a rendszer automatikusan végzi.

kampó eljárás (hook predicate)

Olyan eljárás, amit a felhasználó definiál, de közvetlenül a rendszer hívja meg.

karakterkód (character code)

Egy karakter numerikus kódja.

kifejezés (term)

A Prologban előforduló adatfajták gyűjtőneve.

klóz (clause)

Egy tényállítás vagy egy szabály.

konjunkció (conjunction)

És kapcsolat. Prologban az és kapcsolatban levő relációkat vesszővel elválasztva egymás után írjuk.

konstans (constant)

Egy egész vagy egy lebegőpontos szám, vagy egy atom. Az `atomic`/1 beépített eljárás pontosan ezekre igaz.

konzultálás (consult)

Az a folyamat, amikor a program szövegéből interpretált alakot állítunk elő. Az interpretált alak lassabban fut, mint a lefordított alak, de kilistázható (`listing`) és egy kicsit részletesebben nyomkövethető.

kérdés, célsorozat (query)

Célok konjunkciója. Ha egy célsorozatot közvetlenül az interpreternek adjuk oda, kérdésnek is nevezzük.

lista (list)

Vagy a `'[]'` (nil) atom, vagy egy `'.'`/2 struktúra, melynek második argumentuma egy lista. Ha egy lista vége nem nil, hanem változó, akkor nyitott végű listáról beszélünk.

meghívható kifejezés (callable term)

Egy atom vagy egy struktúra. A `callable`/1 eljárás pontosan ezekre igaz.

mellékhatás (side-effect)

Logikailag nem értelmezhető hatás.

meta eljárás (meta-predicate)

Olyan eljárás, amely meghívja valamelyik argumentumát.

meta hívás (meta-call)

Egy Prolog kifejezés hívásként való értelmezése. Lásd a `call/1` eljárást!

modul (module)

Eljárások olyan (legbővebb) halmaza, amelyek azonos modul-deklaráció hatókörébe esnek.

névtelen változó (anonymous variable)

A névtelen változó egy minden más változótól különböző változó. Jelölése: `_`.

operátor (operator)

Olyan név, amelyet (operátor-deklaráció) segítségével prefix, infix vagy postfix alakban használhatóvá tettünk.

összetett kifejezés (compound term)

Lásd struktúra.

parancs (directive)

Fordítási időben végrehajtandó Prolog hívás.

precedencia (precedence)

Az a szám, amely megmondja egy operátorról, hogy mennyire szorosan köt.

predikátum (predicate)

Lásd eljárás.

program (database)

A felhasználó által definiált eljárások halmaza.

pár (pair)

A `'-'/2` struktúra. Általánosan bármely kétargumentumú struktúrát nevezhetünk párnak, de ilyenkor meg kell mondani mi a neve.

redukciós lépés (reduction step)

A végrehajtásnak az a lépése, amikor egy hívást egy illeszkedő fejű klóz törzsével helyettesítünk.

rekurzió (recursion)

Egy eljárás rekurzív, ha végrehajtásához szükség van önmaga (esetleg közvetett) meghívására.

statikus eljárás (static predicate)

Egy nem dinamikus eljárás.

struktúra (structure)

Egy Prolog kifejezés. A strúktúráknak van egy neve és egy vagy több argumentuma. Például a `személy('Weöres Sándor', 1913, 1989)` struktúra neve `személy` és három argumentuma van. A `compund/1` eljárás pontosan ezekre igaz.

szabály (rule)

Egy klóz egy vagy több feltétellel. Egy szabály azt fejezi ki, hogy a törzséből következik a feje. Például a `jókedvű(attila) :- kertész(attila).` szabály azt fejezi ki, hogy ha Attila kertész, akkor jókedvű.

Változókat használva általánosabb szabályokat is felírhatunk:
`jóhírű(X) :- tejet_iszik(X), pipázik(X).`
Ezzel azt állítjuk, hogy minden pipázó és tejivó X-re igaz, hogy jó a híre.

szemantika (semantics)

A Prolog kifejezések „értelme”.

szemétgyűjtés (garbage collection)

Az elérhetetlenné vált adatok által foglalt memóriaterületek újra felhasználhatóvá tétele.

szintaxis (syntax)

A nyelvtan azon része, amely azt írja le, hogyan állnak elő szimbólumokból érvényes Prolog kifejezések.

szülő (parent)

Az a hivás, amelynek redukciója során a kérdéses cél belekerült a célsorozatba. Más szóval az a hívás, amely a kérdéses célt tartalmazó klóz fejével illesztődött.

tényállítás (fact)

Egy klóz, amelynek nincs feltétel része, azaz a törzse üres. Egy tényállítás azt fejezi ki, hogy az adott reláció fennáll az argumentumaira. Példák tényállításokra (és lehetséges interpretációjukra):

```
király(henrik, anglia).   % Henrik Anglia királya volt.
csőre_van(madár).         % A madaraknak van csőrük.
alkalmazott(lilla, adatfeldolgozás, 55000).
                          % Lilla az adatfeldolgozási osztályon dolgozik,
                          % keresete 55000.
```

tömör (ground)

Olyan Prolog kifejezés, amelyben nem szerepel értékkel nem bíró változó.

törzs (body)

Egy szabály feltétel része.

visszalépés (backtracking)

Ha egy egyszer már kielégített célt más módon újra megpróbálunk kielégíteni, akkor visszalépésről beszélünk.

változó (variable)

Egy név ami egy (esetleg ismeretlen) értéket jelöl a programban. Prologban a változókat nagybetűvel vagy aláhúzással kezdődő és alfanumerikus karaktersorozattal folytatódó szöveggel jelöljük.

B függelék

A gyakorló feladatok megoldásai

Ebben a függelékben a gyakorló feladatok megoldásait közöljük. Egyes megoldásokban felhasználtuk a `lists` könyvtár eljárásait is. Az időméréshez használt `time/2` eljárás definicióját a függelék végén közöljük.

GY1. feladat

Egy célsorozat keresési fája egy olyan irányított gráf, amelynek csúcsaiban célsorozatok vannak és két csúcs között akkor megy él, ha a kiinduló csúcs célsorozatából egyetlen (Prolog) redukciós lépéssel eljuthatunk a cél csúcs célsorozatába. Az élek a redukció során alkalmazott klóz sorszámával vannak címkézve. A fa gyökerében a teljes kiindulási célsorozat van, az üres célsorozatot egy négyzettel jelöljük.
Rajzolja fel az alábbi célsorozatok keresési fáját! A fa nem üres célsorozatot tartalmazó leveleinél jelezze a visszalépést! (A `select`, `write` és `append` szavakat rövidítheti (`s`, `w`, `a`).)

1. `select(A, [4,2,3,8], [4,2|B]), write(A-B).`
2. `select(U, [b,c], V]), write(U-V).`
3. `append([A|B], C, [1,2]), write([A|B]-C).`
4. `append(A, [B|C], [d,t]), write(A-[B|C]).`

Megoldás

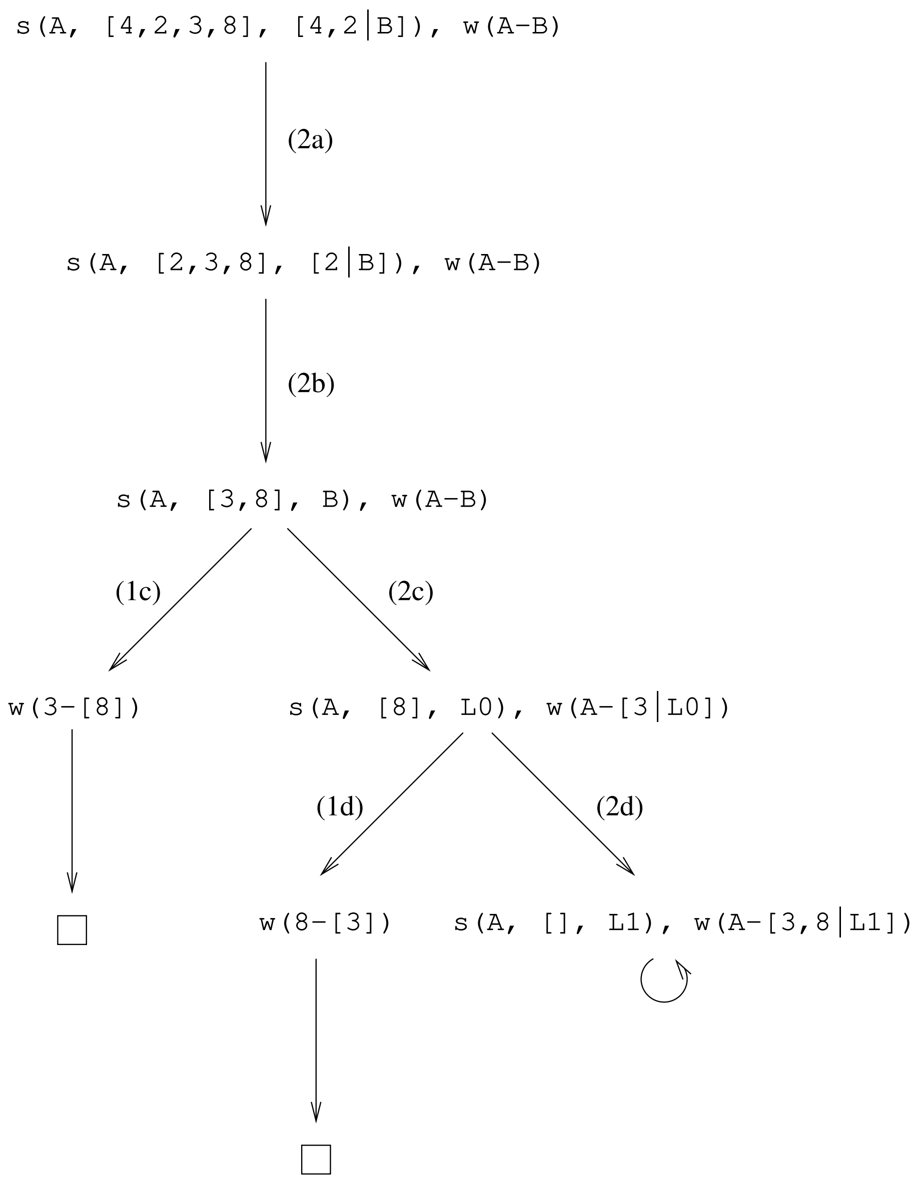
Csak az 1. részfeladatot oldjuk meg. A keresési fa a B.1 ábrán látható. Hogy az ábra jobban követhető legyen, minden redukciós lépéshez egy betűt is rendeltünk, és alább megadjuk az adott lépésben választott klóz fejillesztés utáni alakját.

```
2a  s(A, [4|[2,3,8]], [4|[2|B]]) :- s(A, [2,3,8], [2|B])
2b  s(A, [2|[3,8]], [2|B]) :- s(A, [3,8], B)
1c  s(3, [3|[8]], [8])
2c  s(A, [3|[8]], [3|L0]) :- s(A, [8], L0)
1d  s(8, [8], [])
2d  s(A, [8], [8|L1]) :- s(A, [], L1)
```

GY2. feladat

Írja fel az alábbi kifejezéspárok alapstruktúra-alakját (azaz szintaktikus édesítőszerek nélküli formáját) vagy rajzolja fel a hozzájuk tartozó fastruktúrákat! Állapítsa meg, hogy a két kifejezés egyesíthető-e, és ha igen, milyen behelyettesítéssel!

1. `.(A+C*A,[C])`   `[w+B,3]`
2. `f(2*A+B, [C], B*C)`   `f(U+V, .(V,_), U)`



B.1. ábra: Az 1. RÉSZFELADAT KERESÉSI FÁJA

3. `f(1*Y-U, [V], V*U)` `f(A-3, B, A)`
4. `[f(B*2, A, [B|A]),_]` `[f(Y*Y, [], X)|X]`

Megoldás

Itt az alapstruktúra-alakot fogjuk felírni (annak ellenére, hogy a fastruktúra alak általában áttekinthetőbb).

1. `.(+(A, *(C, A), .(C, []))` `.(+(w, B), .(3, []))`
   A behelyettesítés: `A = w, B = 3*w, C = 3`
2. `f(+(*(2, A), B), .(C, []), *(B, C))` `f(+(U, V), .(V,_), U)`
   A behelyettesítés: `A = B = C = V = 2, U = 2*2`
3. `f(-(*(1, Y), U), .(V, []), *(V, U))` `f(-(A, 3), B, A)`
   A behelyettesítés: `A = 1*3, B = [1], U = Y = 3, V = 1`
4. `.(f(*(B, 2), A, .(B, A)), .(_, []))` `.(f(*(Y, Y), [], X), X)`
   A behelyettesítés: `A = [], B = Y = 2, X = [2]`

GY3. feladat

Bizonyítsuk be ciklus-invariáns segítségével a `hatv` függvény helyességét.

Megoldás

1. $e = 1 \wedge\ a = a_0 \wedge\ h = h_0 \rightarrow\ a_0^{h_0} = ea^h$
2. Tegyük fel, hogy $a_0^{h_0} = ea^h$, ekkor
$$e_1 a_1^{h_1} = ea^{h\&1}(aa)^{(h-(h\&1))/2} = ea^{h\&1}a^{h-(h\&1)} = ea^h = a_0^{h_0}$$
3. A ciklus végén $n = 0$ és ezért $a_0^{h_0} = ea^0 = e$.

GY4. feladat

Írjunk Prolog programot nem-negatív számok legnagyobb közös osztójának kiszámítására, az Euklideszi algoritmus segítségével.
Egy `A` szám `B`-vel való osztásakor képződő `M` maradék kiszámítására az `M is A mod B` aritmetikai eljárást használhatjuk.

Megoldás

```
lnko(A, 0, A).
lnko(A, B, C):-
      B > 0, M is A mod B,
      lnko(B, M, C).
```

A megoldás tesztelése:

```
lnko_teszt :-
      format('~nAz lnko feladat tesztelese:~2n', []),
      A=12, B=32,
      lnko(A, B, L),
      format('    ~d es ~d LNKO-ja ~d.~n', [A,B,L]),
      C=216, D=540,
```

```
        lnko(C, D, M),
        format('    ~d es ~d LNKO-ja ~d.~n', [C,D,M]),
        nl, nl.
```

Kimenete:

```
Az lnko feladat tesztelese:

     12 es 32 LNKO-ja 4.
     216 es 540 LNKO-ja 108.
```

GY5. feladat

Írjunk egy

```
egyszerusitheto(X,Y)
```

Prolog eljárást amely felsorolja a következő tulajdonságokkal bíró `(X,Y)` számpárokat:

- `X` és `Y` tizes számrendszerben kétjegyű természetes számok
- `X` tizes számrendszerbeli alakja AB
- `Y` tizes számrendszerbeli alakja BC
- Az `X/Y` és A/C törtek (végtelen pontossággal) megegyeznek
- A, B és C páronként különböző számjegyek

(azaz az `X/Y` $= AB/BC$ tört egyszerűsíthető a B számjegyek elhagyásával).
(Forrás: 1. Prolog programozási verseny, 1994 november, Ithaca, NY)
Az `X =\= Y` beépített eljárást használhatjuk annak eldöntésére, hogy `X` és `Y` különbözőek-e.

Megoldás

```
egyszerusitheto(X, Y):-
      between(1, 9, A),
      between(1, 9, B),
      between(0, 9, C),
      A =\= B, B =\= C, A =\= C,
      X is 10*A+B, Y is 10*B+C,
%     X/Y =:= A/C.                      % helyette, 0-val való osztás elkerülésére:
      X*C =:= Y*A.

between(N, M, N):-
      N =< M.
between(N, M, I):-
      N < M, N1 is N+1, between(N1, M, I).
```

A megoldás tesztelése:

```
egysz_teszt :-
      format('~nAz egyszerusitheto feladat tesztelese:~2n', []),
      (   egyszerusitheto(X,Y),
          format('    ~d/~d szamjegytorlessel egyszerusitheto.~n', [X,Y]),
          fail
      ;   nl, nl
      ).
```

Kimenete:

```
Az egyszerusitheto feladat tesztelese:

     16/64 szamjegytorlessel egyszerusitheto.
     19/95 szamjegytorlessel egyszerusitheto.
     26/65 szamjegytorlessel egyszerusitheto.
     49/98 szamjegytorlessel egyszerusitheto.
```

GY6. feladat

Az $f(i)$ Fibonacci-szerű számsorozatot a következő rekurzív módon definiáljuk:

$$f(1) = 1, f(2) = 1, f(i) = f(i-1) + 3 * f(i-2)$$

Írjunk egy `tag(I, FI)` Prolog eljárást, amely a sorozat `I`-edik tagját `FI`-ben előállítja. Pl. a `tag(3,F)` hívás eredménye `F=4` lesz.
Kiséreljünk meg olyan definíciót írni a fenti `tag(I, FI)` eljárásra, amelynek futási ideje `I`-vel arányos (lineáris `I`-ben).

Megoldás

```
% nem lineáris idejű egyszerű megoldás:

tag(N, F):-
      N > 0, N =< 2, F = 1.
tag(N, F):-
      N > 2,
      N1 is N-1, N2 is N-2,
      tag(N1, F1), tag(N2, F2),
      F is F1+3*F2.


% tag2:  lineáris idejű megoldás

tag2(I, F):-
      tag2(I, F0, _F1),
      F = F0.


% tag2(I, FI, FI1):  az I-edik tag-szám FI, az I-1-edik FI1.

tag2(1, 1, 0).
tag2(I, F, F1):-
      I > 1,
      I1 is I-1,
      tag2(I1, F1, F2),
      F is F1+3*F2.
```

A megoldás tesztelése:

```
tag_teszt:-
      format('~nA tag feladat tesztelese:~2n', []),
      (   tag(2, F2), tag(5, F5), tag(20, F20),
          format('    tag(2)=~d, tag(5)=~d, tag(20)=~d~n', [F2,F5,F20]),
```

```
            time('tag(20)', tag(20,_)),
            nl, fail
        ;   tag2(2, F2), tag2(5, F5), tag2(20, F20),
            format('    tag2(2)=~d, tag2(5)=~d, tag2(20)=~d~n',
              [F2,F5,F20]),
            time('tag2(20)', tag2(20,_)),
            fail
        ;   nl, nl
        ).
```

Kimenete:

```
A tag feladat tesztelese:

    tag(2)=1, tag(5)=19, tag(20)=4875913
    tag(20): 0.070 seconds.

    tag2(2)=1, tag2(5)=19, tag2(20)=4875913
    tag2(20): 0.000 seconds.
```

GY7. feladat

Adottnak feltételezzük a következő eljárást:

```
szuloje(Gyermek, Szulo).
```

Erre építve definiáljuk a következő eljárásokat (ehhez természetesen segédeljárásokat is definiálhatunk):

(a) `unokatestvere(A, B)` (`A` és `B` szülei testvérek)

(b) `n_ed_unokatestvere(N, A, B)`
(1. unokatestvérek = unokatestvérek
2. unokatestvérek = unokatestvérek gyermekei
stb.)
`N` adott szám, `A` és `B` változók is lehetnek.

Megoldás

```
unokatestvere(A, B):-
      szuloje(A, ASz), szuloje(B, BSz),
      testvere(ASz, BSz).

testvere(A, B):-
      szuloje(A, Sz), szuloje(B, Sz), A \== B.

n_ed_unokatestvere(1, A, B):-
      unokatestvere(A, B).
n_ed_unokatestvere(N, A, B):-
      N > 1,
      szuloje(A, ASz), szuloje(B, BSz),
      N1 is N-1,
      n_ed_unokatestvere(N1, ASz, BSz).

% próba-adatok:

szuloje(a,b).
```

```
szuloje(a,c).
szuloje(d,e).
szuloje(d,f).
szuloje(g,e).
szuloje(g,f).
szuloje(e,h).
szuloje(b,h).
szuloje(i,a).
szuloje(j,g).
```

A megoldás tesztelése:

```
unoka_teszt:-
      format('~nAz unokatestver feladat tesztelese:~2n', []),
      (   unokatestvere(X, Y),
          format('    ~w.~n', [unokatestvere(X,Y)]), fail
      ;   n_ed_unokatestvere(2, X, Y),
          format('    ~w.~n', [n_ed_unokatestvere(2,X,Y)]), fail
      ;   nl, nl
      ).
```

Kimenete:

```
A unokatestver feladat tesztelese:

    unokatestvere(a,d).
    unokatestvere(a,g).
    unokatestvere(d,a).
    unokatestvere(g,a).
    n_ed_unokatestvere(2,i,j).
    n_ed_unokatestvere(2,j,i).
```

GY8. feladat

Tegyük fel, hogy egy súlyozott élű irányítatlan gráfot a Prolog adatbázisában tárolt következő alakú tényállításokkal adunk meg:

```
el(Honnan, Hova, Koltseg)
```

Egy ilyen állítás azt fejezi ki, hogy a gráf `Honnan` csúcsából vezet él a `Hova` csúcsba, és ennek az élnek `Koltseg` a költsége, ahol `Koltseg` egy szám. A `Honnan` és `Hova` csúcsok sorrendje nem lényeges (irányitatlan a gráf), de egy adott csúcspár csak egyszer szerepel az '`el`' relációban.
Írjunk egy `kormentes_ut(Honnan, Hova, Koltseg)` Prolog eljárást, amely azt fejezi ki, hogy a `Honnan` csúcsból el lehet a gráfban jutni a `Hova` csúcsba egy önmagát nem érintő útvonalon, amelynek összköltsége `Koltseg`.

Megoldás

```
kormentes_ut(A, B, Kolts):-
      ut(A, B, [A], 0, Kolts).

% ut(A, C, Kizart, K0, K): A-bol C-be eljuthatunk K-K0 költségen,
% úgy hogy közben (a kezdőpont kivételével) nem érintjük a
% Kizart lista elemeként szereplő pontokat (Kizart es K0 bemenő
% parameterek).
```

```
ut(A, C, Kizart, K0, Kolts):-
      szakasz(A, B, K),
      nem_kizart(Kizart, B),
      K1 is K0+K,
      (   B = C, Kolts = K1
      ;   ut(B, C, [B|Kizart], K1, Kolts)
      ).


% nem_kizart(Kizart, A): A nem eleme a Kizart listának.

nem_kizart([], _).
nem_kizart([K|Kizart], A):-
      A \== K, nem_kizart(Kizart, A).


% A-bol B-be eljuthatunk egyetlen él mentén K költséggel.
szakasz(A, B, K):-
      el(A, B, K).
szakasz(A, B, K):-
      el(B, A, K).
```

A megoldás tesztelése:

```
% tesztadatok

el(a,b,1).
el(a,c,2).
el(g,a,5).
el(g,j,2).
el(j,c,8).

ut_teszt:-
      format('~nA kormentes_ut feladat tesztelese:~2n', []),
      (   kormentes_ut(a, Y, K),
          format('    Letezik kormentes_ut ~w-bol ~w-ba ~d koltsegen~n',
            [a,Y,K]),
          fail
      ;   nl, nl
      ).
```

Kimenete:

```
A kormentes_ut feladat tesztelese:

    Letezik kormentes_ut a-bol b-ba 1 koltsegen
    Letezik kormentes_ut a-bol c-ba 2 koltsegen
    Letezik kormentes_ut a-bol j-ba 10 koltsegen
    Letezik kormentes_ut a-bol g-ba 12 koltsegen
    Letezik kormentes_ut a-bol g-ba 5 koltsegen
    Letezik kormentes_ut a-bol j-ba 7 koltsegen
    Letezik kormentes_ut a-bol c-ba 15 koltsegen
```

GY9. feladat

Írjunk egy

```
nszerese(N, L, NL)
```

Prolog eljárást, amely az `L` lista `N`-szeres egymás után füzését egyesíti `NL`-lel. (Pl.:

```
?- nszerese(3, [a,b], [a,b,a,b,a,b])
```

sikerül).
Kiséreljünk meg hatékony, jobb-rekuzív megoldást adni.

Megoldás

```
% 1. megoldás

nszerese(0, _, []).
nszerese(N, L, NL) :-
      N > 0, N1 is N-1,
      append(L, N1L, NL),              % itt N1L még ismeretlen, azaz üres
                                       % változó, de az append 2. arg.
                                       % pozícióján ez megengedhető
      nszerese(N1, L, N1L).

% 2. megoldás

nszerese2(N, L, NL):-
      nszerese2(N, L, [], NL).

% nszerese2(N, L, L0, NL): L0 elé füzve L N-szeresét kapjuk NL-t
nszerese2(0, _, L, L).
nszerese2(N, L, L0, NL):-
      N > 0, N1 is N-1,
      append(L, L0, L1),
      nszerese2(N1, L, L1, NL).

% 3. megoldás (nem jobbrekurziv)

nszerese3(0, _, []).
nszerese3(N, L, NL) :-
      N > 0, N1 is N-1,
      nszerese3(N1, L, N1L),
      append(L, N1L, NL).

% 4. megoldás (trükkös)

nszerese4(N, L, NL):-
      append(L, L0, L1),
      nszerese4kl(N, L1-L0, NL).

% nszerese4kl(N, DL, NL): A DL kulönbséglista N-szerese az NL lista.
% (copy_term(A, B) A egy másolatát egyesíti B-vel).
nszerese4kl(0, _, []).
nszerese4kl(N, DL, NL):-
      N > 0, N1 is N-1,
```

```
        copy_term(DL, NL-NL1),
        nszerese4kl(N1, DL, NL1).
```

A megoldás tesztelése:

```
egyforma(X, X, X, X).

nszerese_teszt :-
      format('~nAz nszerese feladat tesztelese:~2n', []),
      L = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
      21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,
      41,42,43,44,45,46,47,48,49,50],
      format('    50 elemu listak 1000-szerezese:~n', []),
      time('nszerese ', nszerese(1000, L, NL1)),
      time('nszerese2', nszerese2(1000, L, NL2)),
      time('nszerese3', nszerese3(1000, L, NL3)),
      time('nszerese4', nszerese4(1000, L, NL4)),
      time('egyformak', egyforma(NL1, NL2, NL3, NL4)),
      length(L, LH), length(NL1, NLH),
      format('~n    ~d elemu lista 1000-szerese ~d elemu ~3n',[LH,NLH]).
```

Kimenete:

```
Az nszerese feladat tesztelese:

    50 elemu listak 1000-szerezese:
    nszerese : 0.050 seconds.
    nszerese2: 0.050 seconds.
    nszerese3: 0.050 seconds.
    nszerese4: 0.190 seconds.
    egyformak: 0.200 seconds.

    50 elemu lista 1000-szerese 50000 elemu
```

GY10. feladat

Írjunk egy olyan `atrendez(L, L1)` Prolog eljárást, amely adott `L` (egész számokból álló) lista esetén `L1`-ben előállítja `L` átrendezett formáját a következő értelemben véve: `L1`-ben az összes páros szám megelőzi a páratlanokat, a páratlan és a páros részben a számok sorrendje megegyezik az eredetivel. Pl.:

```
atrendez([1,2,3,4,5,6], L1).
```

eredménye:

```
L1 = [2,4,6,1,3,5]
```

Megoldás

```
% 1. megoldás:

% Az L1 lista az L számlista permutáltja, ugy hogy a
% páros elemek megelőzik a páratlanokat.
atrendez(L, L1):-
      adott_parossaguak(L, 0, P),
      adott_parossaguak(L, 1, T),
      append(P, T, L1).
```

```
% adott_parossaguak(L, I, PL): L azon elemeinek listája, amelyek
% I-vel azonos párosságuak a PL lista.  (I 0 vagy 1 lehet.)
adott_parossaguak([], _, []).
adott_parossaguak([X|L], I, [X|L1]):-
      I =:= X mod 2, !,
      adott_parossaguak(L, I, L1).
adott_parossaguak([_X|L], I, L1):-
      adott_parossaguak(L, I, L1).


% 2., hatékonyabb megoldás:
% Az L1 lista az L számlista permutáltja, ugy hogy a
% páros elemek megelőzik a páratlanokat.
atrendez2(L, L1):-
      osztalyoz(L, L1, L2, L2, []).

% osztalyoz(L, P0, P, T0, T): L páros elemeinek listája a P0-P
% különbséglista, a páratlanoké a T0-T különbséglista.
osztalyoz([], P, P, T, T).
osztalyoz([X|L], P0, P, T0, T):-
      I is X mod 2,
      besorol(I, X, P0, P1, T0, T1),
      osztalyoz(L, P1, P, T1, T).

% besorol(I, X, P0, P, T0, T): I az X paritása, eszerint X-et
% besorolja a P0-P (páros) es T0-T (páratlan) különbséglisták
% egyikébe.  A megfelelő különbséglista az egyetlen X elemet
% tartalmazza, a másik üres lesz.
besorol(0, X, [X|P], P, T, T).
besorol(1, X, P, P, [X|T], T).
```

A megoldás tesztelése:

```
atrendezes_teszt :-
      format('~nAz atrendezes feladat tesztelese:~2n', []),
      L = [1,2,3,4,5,6],
      (   atrendez(L, L1),
          format('    ~w atrendezettje (1.  valtozat):  ~w.~n', [L,L1]),
          fail
      ;   atrendez2(L, L1),
          format('    ~w atrendezettje (2.  valtozat):  ~w.~n', [L,L1]),
          fail
      ;   nl, nl
      ).
```

Kimenete:

```
Az atrendezes feladat tesztelese:

    [1,2,3,4,5,6] atrendezettje (1. valtozat): [2,4,6,1,3,5].
    [1,2,3,4,5,6] atrendezettje (2. valtozat): [2,4,6,1,3,5].
```

GY11. feladat

Tegyük fel, hogy tetszőlegesen nagy egész számok kezelésére van szükségünk, ezért az ilyen számokat a tizes számrendszeri alakjukból képzett számlistával ábrázoljuk, pl. 1256 ábrázolása `[1,2,5,6]`. Írjunk egy `osszead(L1, L2, L3)` eljárást, ahol `L1` és `L2` adott számlisták. Az eljárás állítsa elő azt az `L3` számlistát, amely az `L1` és `L2` által jelölt számok összegét ábrázolja. Pl.:

```
osszead([9,2,5,6], [7,5,8], L)
```

eredménye:

```
L = [1,0,0,1,4].
```

(Vigyázat: nem feltételezhető, hogy a listákkal ábrázolt összeadandó számok a Prolog számtartományán belül vannak)

Megoldás

```
% 1. megoldás:

% S1 és S2 jegyeik listájával adott számok összege S.
osszead(S1, S2, S):-
      length(S1, H1),
      length(S2, H2),
      H is max(H1, H2),
      kieg(H1, H, S1, L1),
      kieg(H2, H, S2, L2),
      egyformakat_osszead(L1, L2, AT, S0),
      atvitelt_hozza(AT, S0, S).

% kieg(H, N, S, L): Az L lista a H hosszúságú S listából
% úgy áll elő, hogy N (>=H) hosszúságúra egészítjük
% ki, 0 elemek eléfűzésével.
kieg(N, N, S, S):-
      !.
kieg(N1, N, S, L):-
      N2 is N1+1,
      kieg(N2, N, [0|S], L).

% egyformakat_osszead(S1, S2, A, S):
% S1 és S2 jegyeik azonos hosszúságú listájával adott
% számok összeadásának eredménye az S (velük azonos
% hosszúságú) lista és az A átvitel.
egyformakat_osszead([], [], 0, []).
egyformakat_osszead([X1|L1], [X2|L2], A, [X|L]):-
      egyformakat_osszead(L1, L2, A0, L),
      X0 is X1+X2+A0,
      X is X0 mod 10,
      A is X0 // 10.

% atvitelt_hozza(A, S1, S2):
% Az S1 listával ábrázolt számhoz az A átvitelt adva kapjuk
% az S2 számot.
atvitelt_hozza(0, S, S).
atvitelt_hozza(1, S, [1|S]).
```

```
% 2. megoldás

% S1 és S2 jegyeik listájával adott számok összege S.
osszead2(S1, S2, S):-
      reverse(S1, R1), reverse(S2, R2),
      ford_osszead(R1, R2, 0, R),
      reverse(R, S).

% ford_osszead(R1, R2, A, R): Az R1 és R2 jegyeik fordított
% listájával adott számok valamint az A átvitel összege
% az R (fordított listával megadott) szám.
ford_osszead(R, [], 0, R):- !.
ford_osszead([], R, 0, R):- !.
ford_osszead(R1, R2, A, [J|R]):-
      szet(R1, J1, M1),
      szet(R2, J2, M2),
      J0 is J1+J2+A,
      J is J0 mod 10,
      A1 is J0 // 10,
      ford_osszead(M1, M2, A1, R).

% szet(R, J, M): Az R jegyeinek fordított listájával megadott
% szám utolsó jegye J, az ennek levágása után maradó
% (fordított listával megadott) szám M.
szet([], 0, []).
szet([X|R], X, R).
```

A megoldás tesztelése:

```
osszeg_teszt :-
      format('~nAz osszeg feladat tesztelese:~2n', []),
      L1 = [9,2,5,6], L2 = [7,5,8],
      (   osszead(L1, L2, L),
          format(' ~w es ~w osszege (1.  valt.):  ~w.~n', [L1,L2,L]),
          fail
      ;   osszead2(L1, L2, L),
          format(' ~w es ~w osszege (2.  valt.):  ~w.~n', [L1,L2,L]),
          fail
      ;   nl, nl
      ).
```

Kimenete:

```
Az osszeg feladat tesztelese:

     [9,2,5,6] es [7,5,8] osszege (1. valt.): [1,0,0,1,4].
     [9,2,5,6] es [7,5,8] osszege (2. valt.): [1,0,0,1,4].
```

GY12. feladat

Tegyük fel, hogy a dátumokat egy ‘`d(Ev,Ho,Nap)`’ 3-argumentumú rekordstruktúrával ábrázoljuk. Írjunk egy

```
masnap(Datum, KovDatum)
```

Prolog eljárást, amely adott `Datum` eseten meghatározza a következő nap dátumát `KovDatum`-ban (`Datum`-ról feltételezhetjük, hogy érvényes dátum). A szökőéveket a Gregorián naptár szerint vegyük figyelembe (a 100-zal osztható de 400-zal nem osztható évek nem szökőévek, a többi néggyel osztható év szökőév). Példák:

```
?- masnap(d(1900, 2, 28), Kov).          Kov = d(1900, 3, 1) ;
?- masnap(d(2000, 2, 28), Kov).          Kov = d(2000, 2, 29) ;
```

Megoldás

```
% masnap(Datum, KovDatum):  KovDatum a Datum után következő nap.
masnap(d(Ev, Ho, Nap), KovDatum):-
      honap_utso_napja(Ho, Ev, Utso),
      masnap(Ev, Ho, Nap, Utso, KovDatum).


% masnap(Ev, Ho, Nap, Utso, KovDatum):  KovDatum az Ev/Ho/Nap
% dátum után következő nap, feltéve hogy Ho utolsó napja Utso
masnap(Ev, Ho, Nap, Utso, d(Ev,Ho,KovNap)):-
      Nap >= 1, Nap < Utso, !,
      KovNap is Nap+1.
masnap(Ev, 12, 31, _, d(KovEv,1,1)):-
      !,
      KovEv is Ev+1.
masnap(Ev, Ho, Nap, Nap, d(Ev,KovHo,1)):-
      KovHo is Ho+1.

% honap_utso_napja(Ho, Ev, Nap):  Ho hónap utolsó napja Ev-ben Nap.
honap_utso_napja(2, Ev, Nap):-
      !, feb_utso_napja(Ev, Nap).
honap_utso_napja(Ho, _, Nap):-
      harmincnapos(Ho), !, Nap = 30.
honap_utso_napja(_, _, 31).

% harmincnapos(Ho):  A Ho sorszámadik hónap 30 napos.
harmincnapos(4).
harmincnapos(6).
harmincnapos(9).
harmincnapos(11).

% feb_utso_napja(Ev, Nap):  Ev-ben február utolsó napja Nap.
feb_utso_napja(Ev, Nap):-
      szokoev(Ev), !, Nap = 29.
feb_utso_napja(_, 28).

% Ev szökőév.
szokoev(Ev):-
      Ev mod 4 =:= 0,
      (    Ev mod 100 =0
      ;    Ev mod 400 =:= 0
      ).
```

A megoldás tesztelése:

```
kovetkezo :-
```

```
        format('~nA kovetkezo feladat tesztelese:~2n', []),
        (   D = d(1900, 2, 28),
            masnap(D, Kov),
            format('    ~w utan kovetkezo nap:  ~w~n', [D, Kov]), fail
        ;   D = d(2000, 2, 28),
            masnap(D, Kov),
            format('    ~w utan kovetkezo nap:  ~w~n', [D, Kov]), fail
        ;   nl,nl
        ).
```

Kimenete:

```
Ak kovetkezo feladat tesztelese:

    d(1900,2,28) utan kovetkezo nap: d(1900,3,1)
    d(2000,2,28) utan kovetkezo nap: d(2000,2,29)
```

GY13. feladat

Írjunk egy `gyakorisaga(Nev)` Prolog eljárást, amely a `Nev` atomban előforduló összes különböző karaktert pontosan egyszer, előfordulási gyakoriságával együtt kiírja. Pl.

```
?- gyakorisaga(abbabbac).
```

eredménye lehet a következő:

```
b gyakorisaga 4 a gyakorisaga 3 c gyakorisaga 1
```

(A kiírás sorrendje tetszőleges lehet).

Megoldás

```
% 1. megoldás:

% Kiírja a Nev-ben elofordulo karaktereket gyakoriságukkal együtt.
gyakorisaga(Nev):-
      atom_codes(Nev, KarL),
      setof(K, member(K, KarL), EgyszL),
      (   member(K, EgyszL),
          elem_gyakorisaga(KarL, K, KN),
          put_code(K), write(' gyakorisaga '), write(KN), nl,
          fail
      ;   nl
      ).

% elem_gyakorisaga(L, X, N):
% az L listában az X elem gyakorisága N.
elem_gyakorisaga([], _K, 0).
elem_gyakorisaga([K|L], K, N):-
      !, elem_gyakorisaga(L, K, N0), N is N0+1.
elem_gyakorisaga([_|L], K, N):-
      elem_gyakorisaga(L, K, N).


% 2. megoldás:
```

```
% Kiírja a Nev-ben elofordulo karaktereket gyakoriságukkal együtt.
gyakorisaga2(Nev):-
      atom_codes(Nev, KarL),
      (   append(L, [K|_], KarL),
          \+ member(K, L),
          elem_gyakorisaga(KarL, K, KN),
          put_code(K), write(' gyakorisaga '), write(KN), nl,
          fail
      ;   nl
      ).


% 3. megoldás:

% Kiírja a Nev-ben elofordulo karaktereket gyakoriságukkal együtt.
gyakorisaga3(Nev):-
      atom_codes(Nev, KarL),
      egyszeres(KarL, EgyszL),
      (   member(K, EgyszL),
          elem_gyakorisaga(KarL, K, KN),
          put_code(K), write(' gyakorisaga '), write(KN), nl,
          fail
      ;   nl
      ).

% egyszeres(L, E): Az E lista az L ismétlődés nélkül vett
% elemeiből áll.
egyszeres([], []).
egyszeres([X|L], E):-
      member(X, L), !, egyszeres(L, E).
egyszeres([X|L], [X|E]):-
      egyszeres(L, E).
```

A megoldás tesztelése:

```
gyakori_teszt :-
      format('~nA gyakori feladat tesztelese:~2n', []),
      Nev = abbabbac,
      (   format(' ~w gyakorisagai (1. valtozat):~n', [Nev]),
          gyakorisaga(Nev),
          fail
      ;   format(' ~w gyakorisagai (2. valtozat):~n', [Nev]),
          gyakorisaga2(Nev),
          fail
      ;   format(' ~w gyakorisagai (3. valtozat):~n', [Nev]),
          gyakorisaga3(Nev),
          fail
      ;   nl, nl
      ).
```

Kimenete:

```
A gyakori feladat tesztelese:

     abbabbac gyakorisaga (1. valtozat):
 a gyakorisaga 3
```

```
b gyakorisaga 4
c gyakorisaga 1

    abbabbac gyakorisaga (2. valtozat):
a gyakorisaga 3
b gyakorisaga 4
c gyakorisaga 1

    abbabbac gyakorisaga (3. valtozat):
b gyakorisaga 4
a gyakorisaga 3
c gyakorisaga 1
```

GY14. feladat

Írjunk egy `titkosit(Szoveg, Eltolas, Titkositott)` Prolog eljárást. Ennek hívásakor `Szoveg` egy adott név (atom) lesz, `Eltolas` pedig egy 1 és 25 közé eső egész szám. Az eljárás feladata a `Szoveg`-et karakterenként átalakítani és eredményként egy új nevet létrehozni a `Titkositott` argumentumban. A titkosítás abból áll, hogy a `Szoveg`-ben szereplő minden kisbetű helyett az ABC-ben `Eltolas` hellyel utána következő kisbetűt kell tenni (az eltolás ciklikus, azaz az ABC-ben a z betű után ismét az a betű jön). A nem kisbetű karaktereket a `Szoveg`-ben változatlanul kell hagyni. Példa:

```
titkosit('Zizi zuz?', 6, T).
```

eredménye:

```
T = 'Zofo faf?'
```

Megoldás

```
titkosit(Szo, Eltolas, TSzo):-
      atom_codes(Szo, KarL),
      karlistat_titkosit(KarL, Eltolas, TKarL),
      atom_codes(TSzo, TKarL).

% karlistat_titkosit(KL, Elt, TKL): a KL karakterkód-lista Elt
% eltolással titkosított formája TKL.
karlistat_titkosit([], _, []).
karlistat_titkosit([Kar|KarL], Eltolas, [TKar|TKarL]):-
      kart_titkosit(Kar, Eltolas, TKar),
      karlistat_titkosit(KarL, Eltolas, TKarL).

% A Kar karakterkód Eltolas eltolással titkosított formája TKar.
kart_titkosit(Kar, Eltolas, TKar):-
      kisbetu(Kar), !, eltol(Kar, Eltolas, TKar).
kart_titkosit(Kar, _, Kar).

% Kar egy kisbetű kódja.
kisbetu(Kar):-
      Kar >= 0'a, Kar =< 0'z.

% Kar egy kisbetű kodja, ennek Eltolas ciklikus eltolással
% titkosított formája TKar.
eltol(Kar, Eltolas, TKar):-
```

```
        Kar1 is Kar+Eltolas,
        (  kisbetu(Kar1) -> TKar = Kar1
        ;  TKar is Kar1 - (0'z-0'a+1)
        ).
```

A megoldás tesztelése:

```
% A Szo szót Elt-tal titkosítja, és az eredményt kiírja.
teszt_titkosit(Szo, Elt):-
      titkosit(Szo, Elt, TSzo),
      format(' ~w titkos alakja (eltolas = ~d):  ~w~n',
            [Szo,Elt,TSzo]).

titkos_teszt:-
      format('~nA titkos feladat tesztelese:~2n', []),
      teszt_titkosit(titkos_szo, 13),
      teszt_titkosit(gvgxbf_fmb, 13),
      teszt_titkosit('Zizi zuz?', 6),
      nl, nl.
```

Kimenete:

```
A titkos feladat tesztelese:

     titkos_szo titkos alakja (eltolas = 13): gvgxbf_fmb
     gvgxbf_fmb titkos alakja (eltolas = 13): titkos_szo
     Zizi zuz? titkos alakja (eltolas = 6): Zofo faf?
```

GY15. feladat

Írjunk egy `egyszerusit(Kif, UjKif)` Prolog eljárást, amely a `Kif` tetszőleges Prolog kifejezésben előforduló `A+B` alakú részkifejezéseket, ahol `A` és `B` egyaránt számok, az `A` és `B` számok összegére cseréli, a többi részkifejezést változatlanul hagyva. Pl.

```
?- egyszerusit(
      f(a+1,[Y,1+2,Y],3+4),
      Ujkif).
```

eredménye:

```
Ujkif = f(a+1, [Y,3,Y], 7)
```

Kiséreljünk meg olyan megoldást adni, amely a cserét a kifejezésfában alulról felfelé végzi, és ezáltal a kifejezés egyszeri bejárásával azt tovább nem egyszerűsíthető alakra hozza, azaz pl.

```
?- egyszerusit(f(1+2+3+4), Ujkif).
```

eredménye:

```
Ujkif = f(10)
```

Megoldás

```
% 1.  megoldás:
```

```
egyszerusit(Kif, Kif):-
      var(Kif), !.
egyszerusit(Kif, Kif):-
      atomic(Kif), !.
egyszerusit(A+B, Ertek):-
      number(A), number(B), !,
      Ertek is A+B.
egyszerusit(Kif, UjKif):-
      Kif =..  [F|ArgL],
      listat_egyszerusit(ArgL,UjArgL),
      UjKif =..  [F|UjArgL].

% listat_egyszerusit(L1, L2):  L2-t úgy kapjuk, hogy L1 minden
% elemére alkamazzuk az egyszerusit eljárást.

listat_egyszerusit([], []).
listat_egyszerusit([A|AL], [B|BL]):-
      egyszerusit(A, B),
      listat_egyszerusit(AL, BL).


% 2.  megoldás:

% Bonyolultabb megoldás, alulról felfelé cserél, az eredmény
% tovább nem egyszerüsíthető.

egyszerusit2(Kif, Kif):-
      var(Kif), !.
egyszerusit2(Kif, Kif):-
      atomic(Kif), !.
egyszerusit2(A+B, Ertek):-
      !,
      egyszerusit2(A, AErt),
      egyszerusit2(B, BErt),
      uj_ertek(AErt, BErt, Ertek).
egyszerusit2(Kif, UjKif):-
      Kif =..  [F|ArgL],
      listat_egyszerusit2(ArgL,UjArgL),
      UjKif =..  [F|UjArgL].


% listat_egyszerusit2(L1, L2):  L2-t úgy kapjuk, hogy L1 minden
% elemére alkamazzuk az egyszerusit2 eljárást.

listat_egyszerusit2([], []).
listat_egyszerusit2([A|AL], [B|BL]):-
      egyszerusit2(A, B),
      listat_egyszerusit2(AL, BL).

% uj_ertek(A, B, E): E az A+B kifejezés egyszerüsített alakja.

uj_ertek(A, B, E):-
      number(A), number(B), !, E is A+B.
uj_ertek(A, B, A+B).
```

A megoldás tesztelése:

```
egysz_kiir(Kif, Ujkif):-
      format(' ~p egyszerusitett alakja:~n ~p~n',
      [Kif, Ujkif]).

egyszeru_teszt :-
      format('~nAz egyszeru feladat tesztelese:~2n', []),
      Kif1 = f(a+1,[Y,1+2,Y],3+4),
      Kif2 = 1+2+3+4,
      write(' egyszerusit/2:  '), nl,
      egyszerusit(Kif1, Ujkif1),
      egysz_kiir(Kif1, Ujkif1),
      egyszerusit(Kif2, Ujkif2),
      egysz_kiir(Kif2, Ujkif2),nl,
      write(' egyszerusit2/2:  '), nl,
      egyszerusit2(Kif1, Ujkif21),
      egysz_kiir(Kif1, Ujkif21),
      egyszerusit2(Kif2, Ujkif22),
      egysz_kiir(Kif2, Ujkif22), nl, nl.
```

Kimenete:

```
A egyszeru feladat tesztelese:

  egyszerusit/2:
    f(a+1,[_408846,1+2,_408846],3+4) egyszerusitett alakja:
        f(a+1,[_408846,3,_408846],7)
    1+2+3+4 egyszerusitett alakja:
        3+3+4

  egyszerusit2/2:
    f(a+1,[_408846,1+2,_408846],3+4) egyszerusitett alakja:
        f(a+1,[_408846,3,_408846],7)
    1+2+3+4 egyszerusitett alakja:
        10
```

GY16. feladat

Írjunk egy olyan `melysege(Kif, N)` Prolog eljárást, amely tetszőleges adott `Kif` Prolog kifejezés esetén `N`-ben visszadja annak mélységét. Egy kifejezés mélységén a neki megfelelő fastruktúra magasságát értjük, másképpen: a nem összetett kifejezések mélysége 0, egy összetett kifejezés mélysége a legnagyobb mélységű argumentumának mélységénél eggyel nagyobb. Például:

```
:- melysege((a+b)*(f(1)+_V), N)          N = 3
:- melysege([1+2,3+4,5+6], N)            N = 4
```

Megoldás

```
% melysege(K, M): K melysege M.
melysege(V, 0):-
      var(V), !.
melysege(A, 0):-
      atomic(A), !.
melysege(Kif, M):-
```

```
      Kif =..  [_|ArgL],
      arglista_melysege(ArgL, M0),
      M is M0+1.

% arglista_melysege(L, M): az L lista elemei melysegének
% maximuma M.
arglista_melysege([], 0).
arglista_melysege([Arg|ArgL], M):-
      arglista_melysege(ArgL, M0), melysege(Arg, M1),
      max(M0, M1, M).

% A és B számok maximuma C.
max(A, B, C):-
      A>B, !, C=A.
max(_A, B, B).
```

A megoldás tesztelése

```
teszt_melysege(Kif):-
      melysege(Kif, M),
      format('    ~w melysege:  ~d~n',[Kif,M]).

mely_teszt :-
      format('~nA mely feladat tesztelese:~2n', []),
      teszt_melysege((a+b)*(f(1)+_D)),
      teszt_melysege([1,2+3,4]),
      teszt_melysege([1,4,2+3]),
      nl, nl.
```

Kimenete:

```
A mely feladat tesztelese:

    (a+b)*(f(1)+_410972) melysege: 3
    [1,2+3,4] melysege: 3
    [1,4,2+3] melysege: 4
```

GY17. feladat

Tegyük fel, hogy a repülőtársaságok járataira vonatkozóan a Prolog adatbázisában a következő alakú tényállításokkal rendelkezünk:

```
jarat(Honnan, Hova)
```

Egy ilyen állítás azt fejezi ki, hogy van repülőjárat `Honnan` városból `Hova` városba és vissza. Írjon egy

```
elerheto(N, Honnan, CélLista)
```

Prolog eljárást, amely adott `N` és `Honnan` esetén `CélLista`-ban előállítja a `Honnan` városból *legfeljebb* `N` járattal elérhető városok ismétlődés nélküli listáját.

Megoldás

```
% CelLista a Honnan N lépésben elérhető városok listája.
elerheto(N, Honnan, CelLista):-
      setof(Varos, eljut(N, Honnan, Varos), CelLista).
```

```
% eljut(N, Honnan, Hova):  N-nél kevesebb lépésben eljuthatunk
% Honnan Hova.  Tegyük fel, hogy a repülőtársaságok járataira
eljut(N, Hova, Hova):- N >=0.
eljut(N, Honnan, Hova):-
      N>0,
      ( jarat(Honnan, Kozbulso)
      ; jarat(Kozbulso, Honnan)
      ),
      N1 is N-1,
      eljut(N1, Kozbulso, Hova).
```

A megoldás tesztelése

```
jarat(budapest, varso).
jarat(varso, stockholm).
jarat(stockholm,oslo).
jarat(london, oslo).
jarat(madrid,parizs).
jarat(madrid, lisszabon).


jarat_teszt :-
format('~nA jarat feladat tesztelese:~2n', []),
(   member(N, [1,2,3,4]),
          (Honnan=budapest; Honnan=parizs),
          elerheto(N, Honnan, Cel),
          format('    ~d lepesben elerheto:  ~w-~w.~n',[N,Honnan,Cel]),
          fail
;   nl, nl
).
```

Kimenete:

```
A jarat feladat tesztelese:

     1 lepesben elerheto: budapest-[budapest,varso].
     1 lepesben elerheto: parizs-[madrid,parizs].
     2 lepesben elerheto: budapest-[budapest,stockholm,varso].
     2 lepesben elerheto: parizs-[lisszabon,madrid,parizs].
     3 lepesben elerheto: budapest-[budapest,oslo,stockholm,varso].
     3 lepesben elerheto: parizs-[lisszabon,madrid,parizs].
     4 lepesben elerheto: budapest-[budapest,london,oslo,stockholm,varso].
     4 lepesben elerheto: parizs-[lisszabon,madrid,parizs].
```

GY18. feladat

Tegyük fel, hogy a repülőtársaságok járataira vonatkozóan a Prolog adatbázisában a következő alakú tényállításokkal rendelkezünk: `jarat(Honnan, Ind, Hova, Erk)`. Egy ilyen állítás azt fejezi ki, hogy indul repülőjárat `Honnan` városból `Ind` időpontban, amely `Hova` városba `Erk` időpontban érkezik. Az időpontokat `Ora-Perc` alakú Prolog struktúra-kifejezéssel ábrázoljuk (ahol `Ora` és `Perc` egész számok). Írjunk egy olyan `menetrend(Honnan, Hova)` Prolog eljárást, amely adott `Honnan` és `Hova` városok esetén kiírja a két város közötti közvetlen, vagy egyetlen átszállással járó összes utazás adatait. Az átszállóhelyen legalább 45 percet, de legfeljebb 2 órát kell tölteni (és még ugyanaznap tovább kell utazni). A kiírás tartalmazza az (összes) érkezési és indulási időpontot és az átszállóhely nevét is, ha van ilyen, pl.:

```
  bud   waw
  ind   erk   atszallohely   erk   ind
18-00 19-30   NONSTOP
12-00 16-00   prg          13-10 15-10
```

(Megj: nem szükséges a kiírást táblázatba rendezni, csak a fenti információ legyen benne.)

Megoldás

```
menetrend(Honnan, Hova):-
      format('    Jaratok ~w varosbol ~w varosba~n', [Honnan, Hova]),
      format('       Ind Erk Atszallohely Erk Ind~n', []),
      (   jarat(Honnan, I, Hova, Erk),
          format(' ~p ~p NONSTOP~n', [I, Erk]),
          fail
      ;   atszallo_jarat(Honnan, I, Hova, Erk, atsz(Atsz, AE, AI)),
          format(' ~p ~p~| ~w~t~15+ ~p ~p~n',
            [I, Erk, Atsz, AE, AI]),
          fail
      ;
          nl
      ).

% Honnan-ból Ind-kor indulva egyetlen átszállással el lehet
% jutni Hova-ba, Erk-kori érkezessel.  Atsz az átszállással
% kapcsolatos adatok:  atsz(Atszallohely, Atsz_Erk, Atsz_Ind).
atszallo_jarat(Honnan, Ind, Hova, Erk, Atsz):-
      jarat(Honnan, Ind, Atszallohely, Atsz_Erk),
      jarat(Atszallohely, Atsz_Ind, Hova, Erk),
      atszallhat(Atsz_Erk, Atsz_Ind),
      Atsz = atsz(Atszallohely, Atsz_Erk, Atsz_Ind).

% Erk és Ind időpontok közötti idő 45 perc és 2 óra
% közé esik
atszallhat(Erk, Ind):-
      percre(Erk, E), percre(Ind, I),
      Ido is I-E,
      Ido >= 45,
      Ido =< 120.

% percre(Idopont, Perc):  Éjféltől az Idopont-ig Perc perc
% telt el.
percre(O-P, Perc):-
      Perc is O*60+P.
```

A megoldás tesztelése

```
% tesztadatok
jarat('Budapest', 8-00,'Warsaw', 9-30).
jarat('Budapest', 18-00,'Warsaw', 19-30).
jarat('Budapest', 9-00, 'Prague', 10-10).
jarat('Budapest', 8-00, 'Prague', 9-10).
jarat('Budapest', 9-30, 'Prague', 10-30).
jarat('Prague', 11-10, 'Warsaw', 12-00).
jarat('Prague', 12-10, 'Warsaw', 13-00).
```

```
kiir_teszt :-
      format('~nA kiir feladat tesztelese:~2n', []),
      retractall(portray(_-_)),
      asserta((                           % Az időpontok formázása:
            portray(Ora-Perc):-
            format('~|~t~d~2+-~`0t~d~3+',[Ora,Perc])
      )),
      menetrend('Budapest', 'Warsaw').
```

Kimenete:

```
A kiir feladat tesztelese:

     Jaratok Budapest varosbol Warsaw varosba
        Ind   Erk   Atszallohely   Erk   Ind
       8-00  9-30   NONSTOP
      18-00 19-30   NONSTOP
       9-00 12-00   Prague       10-10 11-10
       9-00 13-00   Prague       10-10 12-10
       8-00 12-00   Prague        9-10 11-10
       9-30 13-00   Prague       10-30 12-10
```

GY19. feladat

Írjunk meg egy `kigyujt(File, Nev)` Prolog eljárást, amelynek feladata, hogy az adott `File`-ban levő sorok közül kiírja a képernyőre azokat, amelyek az adott `Nev` atomot a sorban bárhol tartalmazzák. A sorok végét egyetlen 10-es kódú karakter jelzi (Unix konvenció). A `File` minden sora legfeljebb egyszer íródjék ki.

Megoldás

```
kigyujt(F,Nev):-
      see(F), atom_codes(Nev, KarL),
      repeat,
      (    sorbe(L) ->
            tartalmazza(L, KarL), sorki(L), fail
      ;    !, seen
      ).

% L a következő sor karakterkódjainak listája.  File-végnél
% meghiúsul.
sorbe(L):-
      get_code(C), sorbe(C, L).

% sorbe(C, L): Feltéve, hogy C a kurrens karakter, L a sormaradék
% karakterkódjainak listája (beleértve C-t is).  File-végnél
% meghiúsul.
sorbe(-1,_):- !, fail.
sorbe(10, []).
sorbe(C, [C|L]):-
      get_code(C1), sorbe(C1, L).

% Az L lista folytonos részeként tartalmazza a KarL listát.
tartalmazza(L, KarL):-
```

```
        append(KarL, _, L), !.
    tartalmazza([_|L], KarL):-
        tartalmazza(L, KarL).

    % sorki(L): az L karakterkód-listát kiírja egy sorba.
    sorki([]):- nl.
    sorki([C|L]):-
        put_code(C), sorki(L).
```

A megoldás tesztelése

```
    gyujtes_teszt(File, Szo):-
        seeing(F),
        format('~nA gyujtes feladat tesztelese:~2n', []),
        format(' A ~w file azon sorai amelyek a "~w"~n', [File, Szo]),
        format(' szot tartalmazzak:~2n', []),
        kigyujt(File , Szo),
        nl, nl,
        see(F).
```

Kimenete:

```
A gyujtes feladat tesztelese:

     A gyakfel file azon sorai amelyek a "sorbe"
     szot tartalmazzak:

             (    sorbe(L) ->
       sorbe(L):-
             get_code(C), sorbe(C, L).
       sorbe(-1,_):- !, fail.
       sorbe(10, []).
       sorbe(C, [C|L]):-
             get_code(C1), sorbe(C1, L).
             gyujtes\_teszt(gyakfel, sorbe).
```

Időmérő segédeljárás

Az alábbi `time/2` eljárást a GY3 és GY6 feladatok tesztelésében alkalmaztuk.

```
  time(Text, Goal):-
        statistics(runtime, [T0,_]),
        call(Goal), !,
        statistics(runtime, [T1,_]),
        T is T1 - T0,
        format('    ~w: ~3d seconds. ~n', [Text, T]).
```

C függelék

A logikai programozás előzményei

A logikai programozás kialakulására nagy hatással voltak az 1960-as években végzett kutatások a matematikai logikának a programfejlesztés folyamatában való alkalmazásáról, a következő területeken:

- matematikai logika mint programspecifikációs nyelv
- programhelyesség-bizonyítás (programverifikáció)
- automatikus programgenerálás

C.1. Programspecifikácó és programgenerálás

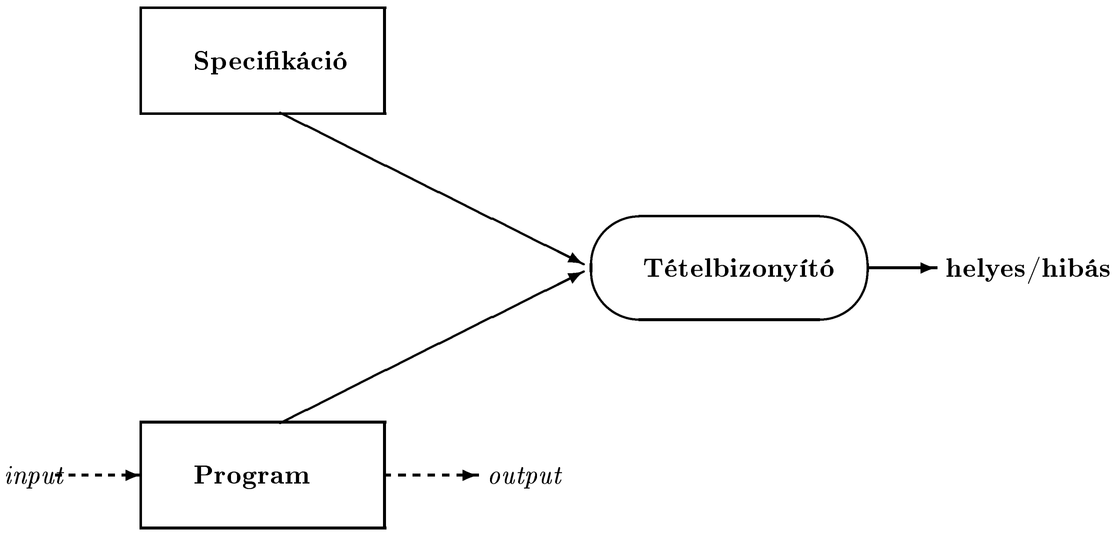
A programspecifikáció nem más, mint a program be- és kimenete közötti összefüggés leírása. Ez többnyire egy függvény, de lehet reláció is.
Például, két szám legnagyobb közös osztójának (lnko) kiszámítását végző program informális specifikációja a következő lehet:

> A és B lnko-ja C, ha A és B közös osztója C, valamint A és B közös osztói között nincs C-nél nagyobb.

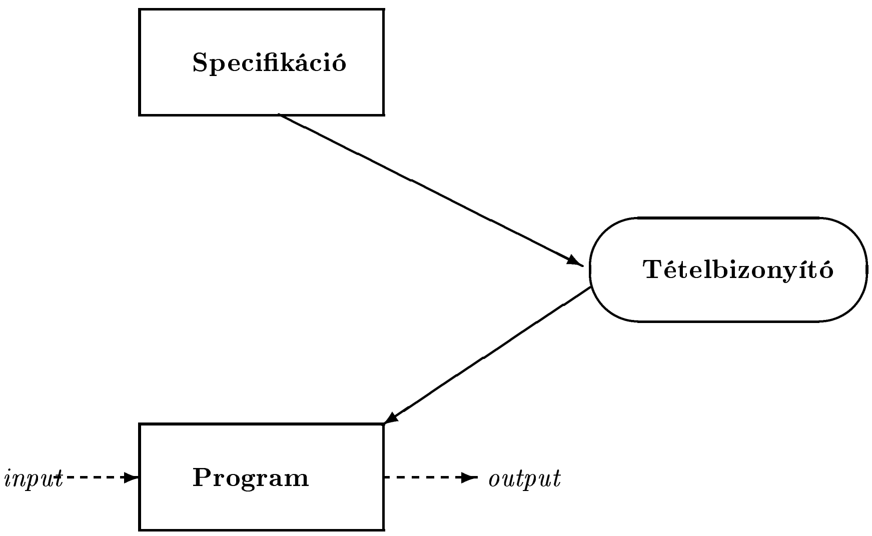
Ugyanez kicsit formálisabban (`ko` = közös osztó):

```
lnko(⟨A, B⟩, C) ⇔
      ko(⟨A, B⟩, C) ∧
      ¬ (∃ D)(ko(⟨A, B⟩, D) ∧ D > C).
```

Egy program akkor helyes, ha bármely (megengedett) bemenő érték-rendszer a kapott kimenő értékkel együtt kielégiti a programspecifikációt. Már a 1960-as években történtek kisérletek a programhelyesség gépi bizonyítására. Az ilyen programhelyesség-bizonyító (programverifikációs) rendszerek leegyszerűsített sémáját az alábbi ábra mutatja:



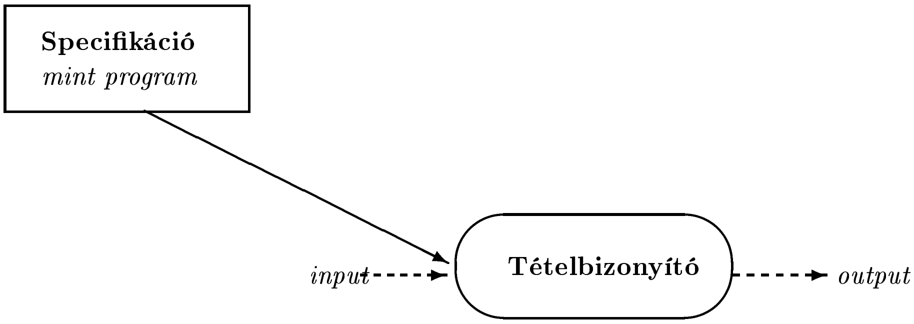
A programspecifikáció egy ambiciózusabb felhasználását kisérlik meg az automatikus programgenerálási rendszerek. Ezek sémája a következő:



Itt tehát a tételbizonyító **generálja** a specifikációnak megfelelő programkódot.
A programverifikációs és automatikus programgenerálási módszerek azonban csak rendkívül korlátozott módon voltak használhatóak, elsősorban az automatikus tételbizonyítási eszközök rossz hatásfoka miatt. Mégis, ezeknek a módszereknek az alapján jött létre a logikai programozás irányzata.

C.2. A logikai programozás sémája

A logikai programozás sémáját mutatja a következő ábra:



Példa

A legnagyobb közös osztó definiciója (és a megfelelő axiómák) alapján igaz

$$\forall \mathtt{X}, \mathtt{Y}\ \exists \mathtt{Z}\ lnko(\langle \mathtt{X}, \mathtt{Y}\rangle, \mathtt{Z})$$

Legyen adott egy konkrét bemenet, pl. $\langle 216{,}90\rangle$, akkor erre

$$\exists \mathtt{Z}\ lnko(\langle \mathtt{216}, \mathtt{90}\rangle, \mathtt{Z})$$

Egy **konstruktív** tételbizonyító ezt úgy bizonyítja, hogy meghatároz egy olyan `Z`-t, amelyre

$$lnko(\langle \mathtt{216}, \mathtt{90}\rangle, \mathtt{Z})$$

fennáll, azaz `Z = 18`, ami a keresett programkimenet.

C.3. A logikai programozástól a Prolog programnyelvig

A logikai programozás éppúgy tételbizonyító eszközökre épít, mint a programverifikációs és programgenerálási módszerek. Így tehát ez utóbbiakhoz hasonlóan komoly problémák merülhetnek fel abban, hogy

- a gépi tételbizonyítás folyamata lassú és nagy helyigényű;
- a feladat bonyolultságának növekedésével hatványozottan nő a lehetséges bizonyitási utak száma (kombinatorikus robbanás);
- a bizonyítás folyamata „fekete doboz”, a felhasználó által át nem látható, így nehéz a gépnek „segíteni” a bizonyításban.

Ezeket a problemákat a Prolog nyelv kifejlesztésekor úgy oldották meg, hogy mind a használható logikai nyelvet, mind pedig a tételbizonyítási módszert rendkívül leegyszerűsítették. Így a Prolog csak ún. Horn-klóz alakú (lásd alább) logikai formulákat enged meg és a tételbizonyítási algoritmust az ún. SL-rezolúcióra szűkíti le (SL = **L**inear resolution with **S**election function, lásd alább). Ezen túlmenően az SL-rezolúción belül is egy nagyon egyszerű kiválasztási függvényt alkalmaz.
Az így kialakuló nyelv fontos tulajdonsága, hogy leírható a tételbizonyítástól teljesen függetlenül, mint egy mintaillesztéses eljárásszervezésen és visszalépéses keresésen alapuló nyelv. Mindezek által a „tételbizonyítási”, azaz programfutási folyamatot a programozó át tudja látni, vezérelni tudja, és így el tudja kerülni a kombinatorikus robbanás veszélyét. A nyelv további, logikailag nem tiszta eszközöket is ad a végrehajtási folyamatba való beavatkozásra (cut, vágó).

Horn klózok

A Horn klózok olyan

$$a \leftarrow b_1 \wedge \ldots \wedge b_n$$

alakú implikációk, amelyekben a és b_i egyszerű, $r(t_1, \ldots, t_k)$ alakú relációkifejezések (literálok), $n \geq 0$. Az implikáció baloldala a klózfej, jobboldala a klóztörzs. A klózban az összes változót univerzális kvantorral lekötöttnek tekintjük.
Egy klózt **szabály**nak hívunk, ha $n > 0$, azaz ha az implikáció jobboldala nem üres. Példa (nsz = nagyszülője, sz = szülője):

$$nsz(U, N) \leftarrow sz(U, Sz) \wedge sz(Sz, N)$$

Ennek kvantorokkal ellátott alakja:

$$\forall UNSz(nsz(U, N) \leftarrow sz(U, Sz) \wedge sz(Sz, N))$$

ami ugyanaz mint:

$$\forall UN(nsz(U, N) \leftarrow \exists Sz(sz(U, Sz) \wedge sz(Sz, N)))$$

Kiolvasva: U-nak nsz-je N, ha van olyan Sz, hogy U-nak sz-je Sz, és Sz-nek sz-je N.
Ha egy Horn-klózban az implikáció jobboldala üres ($n = 0$), azt azonosan igaznak tekintjük (tényállítás). Ilyenkor az $\leftarrow$ jelet is elhagyjuk, pl.:

$$sz(istvan, geza)$$

$$sz(imre, istvan)$$

$$ose(0, Sz, Sz)$$

(A változókat nagybetűvel, a konstansokat kisbetűvel írjuk. Az utolsó tényállítás értelmezése: minden Sz személy 0-ik generációs őse Sz, azaz saját maga)
Végül megengedjük, hogy a Horn-klóz baloldala üres, azaz azonosan hamis legyen (ún. célsorozat), pl.:

$$\leftarrow nsz(imre, X)$$

ennek jelentése: $\forall X \neg nsz(imre, X)$, azaz $\neg \exists X nsz(imre, X)$.
A célsorozat tehát egy negatív állítás, a keresett adat meglétét tagadja (példánkban: $imre$nak nincs nsz-je).
A Prolog végrehajtási mechanizmusának alapját képező tételbizonyítási módszer ebből a negatív állításból kiindulva, viszonylag egyszerű következtetési lépések alkalmazásával ellentmondásra próbál jutni. Mindezt konstruktív módon teszi, azaz megkeresi azokat a változó-behelyettesítéseket, amelyek ellentmondanak a negatív célsorozatnak (példánkban: megkeresi azt az X értéket amely $imre$nak nsz-je).

SL-rezolúció

Legyen adott tényállításoknak és szabályoknak egy halmaza (a program) és egy célsorozat.
A rezolúció folyamata során a (negatív) célsorozatból kiindulva következtetési lépések sorát végezzük el. Minden egyes következtetési lépés eredménye egy újabb célsorozat. Az így előálló célsorozatokat rezolvenseknek hívjuk.
A kezdő **rezolvens** tehát az eredetileg adott célsorozat lesz.
Egy rezolúciós következtetési lépés (kissé leegyszerűsítve) a következő tevékenységekből áll:

- kiválasztunk a rezolvensből egy literált (hogy melyiket, azt az ún. kiválasztási függvény határozza meg)
- keresünk egy olyan programklózt, amelynek a feje és a kiválasztott literál „egyesíthető”, azaz változó-behelyettesítésekkel azonos alakra hozható

- Az egyesítéshez szükséges változó-behelyettesítéseket elvégezzük és a kiválasztott literál helyébe a programklóz törzsét írjuk
- az így kapott új célsorozat a rezolúciós lépés eredménye, az új rezolvens.

A Prolog nyelvben a kiválasztási függvény mindig az első literált választja, és az egyesíthető klózokat is a felírás sorrendjében veszi sorra.
A rezolúciós lépéseket mindaddig ismételjük, amíg vagy

- a rezolvens literáljai elfogynak (egy ún. üres célsorozathoz jutunk) ami a sikeres bizonyítás jele, vagy
- további rezolúciós lépés nem végezhető el, azaz nem találunk a kiválasztott literállal egyesíthető fejű klózt (zsákutca)

Az üres célsorozat azonosan hamis értékű, tehát az indirekt bizonyítás sikerességét jelzi. Az eredeti célsorozatban szereplő változóknak, a bizonyítás közben adott behelyettesítése jelenti a feladat megoldását.
Ha a bizonyítás folyamata zsákutcába kerül, visszamegyünk az előző rezolvenshez és megpróbálunk egy másik rezolúciós lépést elvégezni (azaz egy másik klózt keresni) — ez a visszalépéses keresés.
Tekintsük a következő példát SL-rezolúcióra

Az adott program legyen:

$$nsz(U, N) \leftarrow sz(U, Sz) \wedge sz(Sz, N)$$

$$sz(istvan, geza)$$

$$sz(imre, istvan)$$

Az adott célsorozat legyen:

$$\leftarrow nsz(imre, X)$$

Ekkor a következő rezolúciós lépések végezhetők el (a szükséges változó-behelyettesítéseket | jel után adjuk meg):

$$\leftarrow nsz(imre, X)$$

$$\leftarrow sz(imre, Sz) \wedge sz(Sz, X) \mid U = imre, N = X$$

$$\leftarrow sz(istvan, X) \mid Sz = istvan$$

$$\leftarrow \Box \mid X = geza$$

Ez a bizonyítás tehát sikeres, eredménye az $X = geza$ behelyettesítés.

Egy másik bizonyítás a $\leftarrow nsz(Y, geza)$-ból kiindulva

$$\leftarrow nsz(Y, geza)$$

$$\leftarrow sz(Y, Sz) \wedge sz(Sz, geza) \mid U = Y, N = geza$$

$$\leftarrow sz(geza, geza) \mid Y = istvan, Sz = geza$$

zsákutca, vissza az előző rezolvenshez

$$\leftarrow sz(istvan, geza) \mid Y = imre, Sz = istvan$$

$$\leftarrow \Box$$

Ez a bizonyítás is sikeres, eredménye a $Y = imre$ behelyettesítés (a zsákutcába vezető úton történt behelyettesítések természetesen nem számitanak).

D függelék

A logikai programozás történetéről

Az alábbi áttekintés Szeredi Tamás munkája 1999 júniusából.

D.1. Bevezetés

Ebben a dolgozatban a logikai programozás történetét szeretném áttekinteni, a 70-es évektől napjainkig. Ezelőtt azonban érdemes röviden ismertetni, hogy mi is a logikai programozás, milyen jellegű problémák esetén használható, és hogy miben különbözik más programozási módszerektől. Ezeket a kérdéseket tárgyalja az első fejezet. Mivel a logikai programozás főbb korszakai durván egy-egy évtizedhez kapcsolódnak, ezért a második és harmadik fejezetek a 70-es ill. 80-as évek főbb irányzatait tekintik át. Végül az utolsó fejezet a logikai programozás mai állását, jelentőségét vizsgálja meg.

D.2. Mi a logikai programozás

Az 1950-es évek végén kezdtek először komolyabban felmerülni nem-numerikus, azaz nem konkrét algoritmushoz kötődő feladatok a számítástechnikában. Akkortájt a programozók csak az adott gép assembly nyelvén, vagy ahhoz viszonylag közeli nyelveken programozhattak, míg ezekhez a problémákhoz valamilyen magasabb-szintű nyelvre, egy újfajta szemléletre, hozzáállásra lett volna szükség. Ekkor merült fel, hogy a programozáshoz valamilyen matematikai leíró nyelvet lehet esetleg használni. Ennek első példája az 1960-ban létrejött LISP nyelv (List Programming), amely a matematikai függvény fogalmára épül (az ilyen nyelveket hívják funkcionális nyelveknek). Ez volt egyben az első deklaratív nyelv is.
Míg az imperatív nyelvek alapvetően „felszólító” jellegűek („add össze x-et és y-t, és rakjad az eredményt z-be”), addig a deklaratív nyelvek függvénykapcsolatokat, relációkat írnak fel („x, y és z között az a kapcsolat, hogy az első kettő összege a harmadik”). Fontos jellemzője a deklaratív nyelveknek, hogy nem a változtatható változó, hanem a matematikai változó fogalmára építenek: egy deklaratív nyelvben például az $x = x + 1$ utasításnak nincs értelme, hiszen matematikailag ez egy mindig hamis állítás.
A 60-as évek végére kezdtek olyan feladatok előjönni, amelyekre a függvény-központú szemlélet nem illett jól. Ilyenek voltak például a helyzetmegoldást, cselekvési terv készítését igénylő alkalmazások; ennek egy nagyon egyszerű példáját írom le a következőkben.

> Van egy szoba, amiben van egy majom. A fal mellett van egy szék, és a plafon közepéről lelóg egy banán. A majom a banánt nem éri el, csak ha a széket odatolja, és rááll.

Erre, és ilyen jellegű feladatok megoldására kerestek eszközt. Ezeket ugyan meg lehet oldani funkcionális nyelven is, de csak viszonylag nehézkesen. Ekkor merült fel a logika, mint egy másik matematikai eszköz használatának ötlete; a matematikai logika és az arra épülő tételbizonyítási algoritmusok megfelelő eszköznek tűntek. Ebben a szemléletben a programozónak valamilyen szabályok szerint le kell írnia a világot (a világ megfelelő részének modelljét), és ezen belül a problémák megoldása már egy automatikus tételbizonyító segítségével történhet.
Ha csak egy konkrét feladatot kell megoldani, akkor ezt nem olyan nehéz imperatívan megtenni. Például a majomba „beprogramozhatjuk”, hogy „ha meg akarok szerezni egy banánt, de nem érem el, és van egy szék a szobában, akkor toljam oda, álljak fel rá, és vegyem le a banánt”.

Ha viszont egy általános feladatosztályt kell megoldani, akkor jól jön a logikai szemlélet: mi nem a konkrét algoritmust írjuk le, csak a világot amiben a probléma játszódik, illetve az abban a világban megengedett tevékenységeket. Egy ilyen környezetben sokkal általánosabban fogalmazhatunk meg kérdéseket, a konkrét megoldást egy automatikus tételbizonyító keresi meg (azaz próbálja meg levezetni az általunk megadott világképből).
A logikai szemléletben tehát a majom „programja” egész máshogy néz ki:

statikus világ:	majom lehetséges cselekedetei:
„van egy szoba”	„egyik helyről a másikra megy”
„a szobában a falnál van egy szék”	„egyik helyről a másikra eltolja a széket”
„a szoba közepén lóg egy banán”	„felmászik a székre”
...	„lemászik a székről”
	„magához vesz egy tárgyat, ha eléri”
	...

Ha ezek után azt mondjuk, hogy „a majom megszerzi a banánt”, akkor a fenti állításokból a tételbizonyító levezeti, hogy ez egy igaz állítás-e, és ha igaz, akkor egyben előállítja a szükséges lépéssorozatot. Előnye ennek, hogy ugyanebben a világban külön munka nélkül sokféle más kérdést is feltehetünk, és hogy sokkal átláthatóbb, könnyebben fejleszthető és karbantartható az így megoldott probléma.
Amerikában, amikor elkezdtek ilyen feladatokat megoldani, a logikai leírás és megoldás alkalmazásakor beleütköztek a hatékonyság problémájába. Ugyanis bármilyen komolyabb feladat esetén az előbbinél sokkal bonyolultabb szabályrendszerre van szükség. Mivel a levezethető szabályok száma exponenciálisan nő a kiindulási szabályok számának növekedésével (ezt hívják kombinatorikus robbanásnak), az algoritmus hamar használhatatlanul lassúvá válik. Ennek következtében az amerikaiak holtágnak hitték, és elvetették ezt az irányzatot (bár ebben az is közrejátszott, hogy úgy gondolták, hogy az egyes feladatokat testreszabottan LISP segítségével meg tudják oldani).
Európában ezzel szemben továbbvitték a logikai programozás gondolatát. Az volt az alapgondolat, hogy a tételbizonyítást nem csak eszközként lehet felhasználni egyes alkalmazásokban, hanem — ahogy a funkcionális nyelvek a függvény-fogalomra építenek — a logikára is lehet egy teljes programozási nyelvet építeni. Egy ilyen nyelvben a tételbizonyító a program végrehajtási mechanizmusaként van jelen, azaz központi szerepet kap. Fontos követelmény, hogy a tételbizonyító konstruktív legyen, azaz egy állítás igazságának bizonyításával egyidőben konkrét megoldást is keressen. A tételbizonyítót sikerült olyan mértékben leegyszerűsíteni, hogy a bizonyítási folyamat átláthatóvá vált, és azt a programozó is tudta befolyásolni. Az így kapott hatékonyság ára az volt, hogy a kapott programozási nyelv már közel sem volt teljesen általános (logikai szempontból), de még így is elég „erős” maradt ahhoz, hogy a gyakorlatban hasznosítható legyen.

D.3. A logikai programozás első évtizede, a Prolog születése

A logikai programozás alapgondolata, alapelvei Robert Kowalski-tól származnak. Ezeket az alapelveket először Alain Colmerauer francia kutató vitte át a gyakorlatba, amikor 1972–73-ban a marseilles-i egyetemen munkatársaival megtervezte és megvalósította a Prolog (Programming in Logic) programozási nyelv első változatát.
A Prolog tulajdonképpen kompromisszumot jelentett a logikai programozás elvei és az akkori számítástechnika gyakorlata között. Ahhoz, hogy az elveket a gyakorlatban hasznosítani lehessen, le kellett egyszerűsíteni a használt logika nyelvét: csak az „A ha B és C és ... és D” alakú szabályokat, az ún. Horn klózokat lehetett megadni. Másrészről engedni kellett a deklarativitásból is: nem-deklaratív elemeket is bevezettek, hogy a programozó bele tudjon avatkozni a tételbizonyítási folyamatba. Ilyen elem a „vágó”, amely megmondja a tételbizonyítónak, hogy a keresési tér egy bizonyos részét ne járja be.
Az ún. Marseille Prolog Fortran-ban íródott, és viszonylag lassú volt — ez azért is volt, mert interpretálta, és nem fordította a nyelvet. Ennek ellenére a 70-es években ezt a Prologot több helyen is alkalmazták.
A Prolog első alkalmazása is természetesen Marseille-ben készült, és a természetes nyelvek fordításával volt kapcsolatos.
A második nagy Prolog központ Edinburgh lett; itt már régebb óta volt egy erős logikai iskola (itt dolgozott Kowalski is), és itt kezdett el a logikai programozás témájával foglalkozni David Warren, aki később jelentős mértékben hozzájárult a Prolog fejlődéséhez. 1974-ben Warren több hetet töltött Marseille-ben, hogy megismerje a Prologot, és itt készített egy Prolog alkalmazást, a „Warplan” nevű cselekvéstervező programot.

Edinburgh-ba való visszatérése után elkezdte ott is népszerűsíteni ezt a nyelvet: előadásokat tartott, és írt egy rövid angol nyelvű dokumentációt is.
Magyarországon is voltak hivei a logika alkalmazásának a programozásban, így például a NIM IGÜSZI-ben (a Nehézipari Minisztérium számítóközpontjában) is működött egy csoport Németi István vezetésével, amely a logikának a programozásban való alkalmazásával foglalkozott. Németi többször járt Edinburgh-ban, és hozott a Prologról is információkat[1]. Ezek alapján 1975-ben Szeredi Péter elkészített egy saját Prolog megvalósítást [11].
A Prolog számos ember érdeklődését felkeltette Magyarországon, és ezek hamar sok alkamazási ötlettel jöttek elő. A legelső alkalmazás néhány hét alatt készült el: ez egy egyszintű műhelyt előregyártott panelekből megtervező program volt.
Érdekes módon Magyarországon nagyon nagy volt a kereslet a Prolog iránt, amiben az is közrejátszott, hogy a Prologot egy alkalmazó intézményben fejlesztették ki. Nyilván annak is volt szerepe, hogy Magyarországon nem volt egy olyan erős LISP-kultúra, mint Amerikában. Egy 1982-es áttekintő cikk szerint [8] az első magyar Prologot a 70-es évek végén 16 nagy intézménynél (cégnél, egyetemen, kutatóintézetben), 13-féle különböző architektúrájú számítógépen installálták fel. Számtalan applikáció készült ebben az időszakban Prologban, különféle témakörben. Ezek között voltak gyógyszerkutatással kapcsolatos alkalmazások (pl. „Gyógyszerek kölcsönhatásának előjelzése"), szakértői rendszerek (pl. „Egy interaktív információs renszer a levegőszennyezés szabályozásához"), CAD applikációk (építészetben, gépészmérnöki és villamosmérnöki tevékenységekben). Készültek továbbá különböző szoftverekhez kapcsolódó applikációk (programhelyesség-ellenőrzés, szoftverspecifikáció és -tervezés), szimulációs programok, és sok egyéb területhez tartozó alkalmazások.
A 70-es évek második felében a megvalósítók és alkalmazók közötti gyümölcsöző együttműködés révén sokat fejlődött a magyar Prolog. Újfajta beépített eljárásokkal bővült, és nyomkövetést segítő eszközt is készítettek hozzá.
Egy anekdota: a Prolog egyik nagyon népszerű demó programját Szeredi János készítette el; ez a program egy egyszerű magyar nyelvű kérdés-válasz rendszer volt. A program képes volt egyszerű állításokat és szabályokat értelmezni és eltárolni az emberekről és saját magáról, és ezekkel kapcsolatos kérdésekre tudott válaszolni. Különlegessége az volt, hogy nem fogadott el magára (a programra) nézve negatív következményű állításokat. Ha például már beírta az ember azt, hogy „Aki kedves, az hülye", akkor nem fogadta el a „Te kedves vagy" állítást, és még meg is indokolta, hogy miért. (A történet szerint az első sikeres próbálkozás a program átverésére a „Te hüje vagy" állítás beírása volt.)
Amíg Magyarországon elsősorban a Prolog alkalmazásokkal foglalkoztak, Edinburgh-ban fokozatosan kialakult a Prolog mai arculata: megváltozott a szintaxis, kibővült a beépített eljárások készlete, az eljárások angol neveket kaptak. Egy jelentős lépés volt, hogy 1977-ben Warren elkészítette az első Prolog fordítóprogramot. Amerikában továbbra is nagyon barátságtalanul viseltettek a logikai programozás gondolatával szemben, így a Prolog nyelv fejlesztésében szinte kizárólag európai kutatók vettek részt. Európán belül is Magyarország volt az egyik legfőbb alkalmazó.
A Warren-féle fordítóprogramban számos új ötlet, új módszer került elő, és ez hatással volt a magyarországi Prolog fejlesztésre is. 1978-ban kezdődött el a második magyar Prolog megvalósítás, az MProlog kifejlesztése, egy Szeredi Péter által vezetett csoportban. Ezek a munkálatok a NIM IGÜSZI-ben kezdődtek, de a legtöbb ember fokozatosan átkerült az SZKI akkor nemrég alakult Elméleti Laboratóriumába.
1980-ban tartották az első nagyszabású logikai programozással foglalkozó konferenciát Debrecenben (bár akkor ezt nem konferenciának, hanem workshop-nak hívták). Azt, hogy Magyarországnak ezen a területen mekkora súlya volt, az is mutatja, hogy a résztvevők között nagyjából egyenlő számban voltak magyarok és külföldiek (kb. 60–60 fő).
A 70-es évek során Magyarországon viszonylag sok állami támogatást lehetett szerezni a logikai programozással kapcsolatos kutatás-fejlesztési feladatokra. A 80-as évek elejére azonban kezdtek ezek a lehetőségek megszűnni. A fő probléma a gépidő drágasága volt a nagy nyugati szervereken, és mivel ekkor az emberi munkaerő még viszonylag olcsó volt, a feladatokat célszerűbb volt kézzel megoldani — a természetes intelligencia olcsóbbnak bizonyult a mesterségesnél.

[1] Magát a Marseille Prologot is elhozta, de a Fortran megvalósítások különbözőségei miatt ezt nem sikerült feltámasztani.

D.4. A logikai programozás második évtizede, a japán 5. generációs projekt

Az 1980-as évek elején Japán nagyméretű projektbe vágott bele a számítástechnikában. Az ún. 5. generációs projekt egy újfajta számítógépes rendszer, az FGCS (Fifth Generation Computing System) megalkotását célozta meg, amelynek felhasználó felé nyújtott felületét az intelligencia jellemzi, hardver szinten pedig a párhuzamosság nyújtotta előnyöket használja ki.
1981 őszén jelentették be ennek a 10 éves projektnek a kezdetét. Ebben nagyon ambiciózus célokat tűztek ki maguk elé (például természetes nyelvű kommunikáció a számítógéppel), és a megvalósítás fő elemeként a logikai programozást választották.
A logikai programozás több szempontból is jól illik az alapgondolathoz, jó kapcsolatot teremt az intelligencia és a párhuzamosság között. Egyrészt már korábban láttuk, hogy az intelligens programok fejlesztésére ez egy kézenfekvő eszközt ad. Másrészről, mivel deklaratív, és így egy változóhoz csak egy érték tartozhat, ezért párhuzamos futtatás esetén nem lép fel a hagyományos probléma az értékadások szinkronizációjának kapcsán.
Ez a projekt egy hatalmas fellendülést hozott a logikai programozás világában. Mind Amerikában, mind Európában felkeltette az emberek érdeklődését a téma iránt — felgyorsultak a kutatások, és elkezdtek megjelenni a kereskedelmi igényű megvalósítások is [12].
David Warren időközben Kaliforniában az SRI kutatóintézetben vállalt állást, ahol 1983-ban egy új megvalósítási modellt készített, amit később WAM-nak (Warren Abstract Machine) neveztek el. 84-ben többedmagával alapított egy céget egy WAM-alapú kereskedelmi Prolog létrehozására. Ezt a céget, és az általa készített Prologot Quintus-nak nevezték el (utalva az 5. generációs projektre). Ez a rendszer a 80-as évek végére az egyik vezető kereskedelmi Prolog megvalósítássá vált.
Térjünk vissza most a magyar oldalra, a 80-as évek elejére. Az MProlog 82-re már egy jól használható rendszerré vált, ennek következtében szinte elsőként tudott megjelenni a nagyobb Prolog-rendszerek piacán. Fejlesztésében a fő nehézséget az embargó okozta, mivel a nyugaton használt gépek Magyarországon nem voltak hozzáférhetőek. Ezért is jött kapóra amikor egy magyar származású kanadai üzletember egy olyan cég megalapítását vetette föl, amely az MProlog továbbfejlesztésével és terjesztésével foglalkozna. Ez a cég létre is jött Logicware néven, 1983-ban. Több előnye is volt ennek az együttműködésnek. A magyarok aktív közreműködésével a legmodernebb architektúrákra sikerült hordozni az MPrologot. Ugyanakkor a kanadai szakemberek nagyon színvonalas dokumentációkat, bevezető anyagokat írtak hozzá. Volt azonban egy nagyon rossz döntése a Logicware vezetésének, amely döntően befolyásolta a cég további sorsát. Annak reményében, hogy ettől a Prolog nyelv a hagyományosabb számítástechnikai környezetbe könnyebben beilleszthetővé válik, a nyelv teljes terminológiáját, a beépített eljárások neveit átalakították, figyelmen kívül hagyva az akkorra már szabványosodó Edinburgh-i Prolog dialektust. Ez, és az erős verseny a Quintus-szal és más Prologokkal ahhoz vezetett, hogy a Logicware az 1980-as évek végére kiszorult a Prolog piacról.
Magyarországon az MPrologot a 90-es évek elejéig tovább használták és fejlesztették, de ma már nem folytatják ezt a munkát.
Japánban az 5. generációs projekt folyamán számos problémába ütköztek, és számos hibát is elkövettek a fejlesztés során. A hardverfejlesztés területén ugyan készítettek olyan hardvert, ami a 77-es Warren-féle fordítóprogramot támogatja, de ez a WAM megjelenése óta már elavultnak számított. Egy másik, ennél általánosabb jellegű probléma az, hogy specializált hardvert nem éri meg készíteni, mert lehetetlen tartani a lépést az általános eszközök fejlődésével; bármilyen hatékonyan is van a hardver a problémához illesztve, nem kell sok idő ahhoz, hogy jobb eredményt lehessen elérni általános célú hardverrel. Hasonló hibát követtek el a szoftverfejlesztésben a párhuzamosság kérdésében is: átvették a londoni Imperial College-ben kifejlesztett, csak az ún. ÉS-párhuzamosságot támogató közelítésmódot, amelynek következtében el kellett hagyniuk a Prolognak egy alapvető részét, a visszalépéses keresést. Ezzel egy fontos előnyét veszítették el a Prolognak a funkcionális nyelvekkel szemben.
Ezek miatt a gondok miatt a japán 5. generációs projekt nem hozta meg a tőle várt eredményeket, ami a 90-es évek elején nagyon negatív hatással volt a logikai programozás népszerűségére. Azt az esetleg indokolatlan mértékű népszerűséget, amit a logikai programozás a 80-as években „hitelbe” kapott, azt a 90-es évek elején kellett visszafizetnie.

D.5. A logikai programozás ma

A 80-as évek végén jelent meg a logikai programozásnak egy új irányzata, a CLP (Constraint Logic Programming). Ennek alapgondolata az, hogy egy szűkebb problémakörre koncentrálva egy sokkal erősebb következtetési mechanizmust lehet adni. Ez a következtető „gép” különböző tudomány-területekről jöhet, mint például a mesterséges intelligencia, vagy az operációkutatás. A CLP egy közös keretet ad arra, hogy a problémánkat deklaratív módon írhassuk le, a megoldására viszont a megfelelő tudomány-területről alkalmazhatunk módszereket.
A 90-es évek második felére, a korábban elszenvedett kudarcok ellenére a logikai programozás kezdte megtalálni a helyét. Ugyan nem egy univerzális csodaszer, mint ahogy az ötödik generációs projektben hitték, de jól használható keresési, optimalizálási, szimbolikus feladatokra. Szélesebb körű elterjedésének legfőbb gátja, hogy más jellegű gondolkodást igényel a programozótól, mint a hagyományos programozási nyelvek. De talán éppen ezért egy újfajta rálátást is ad a problémákra, és ma már a legtöbb egyetemen az informatikus képzésben tananyag a logikai programozás. Egyre terjed az ipari felhasználása is, például a Boeing repülőgépgyár is komolyan használja. Különösen dinamikusan terjed a CLP alkalmazása, erre példa a francia Ilog cég, amely világsikert ért el azzal, hogy a CLP-t a C++ nyelvbe ágyazva bocsájtotta a programfejlesztők rendelkezésére.

Irodalomjegyzék

[1] Zsuzsa Farkas, Péter Köves, and Péter Szeredi. MProlog: an implementation overview. In Evan Tick and Giancarlo Succi, editors, *Implementations of Logic Programming Systems*, pages 103–117. Kluwer Academic Publishers, 1994.

[2] Iván Futó. Prolog with communicating processes: From T-Prolog to CSR-Prolog. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 3–17, Budapest, Hungary, 1993. The MIT Press.

[3] Robert A. Kowalski. Predicate logic as a programming language. In *Information Processing '74*, pages 569–574. IFIP, North Holland, 1974.

[4] Márkusz Zsuzsa. *„Prologban programozni könnyű"*. Novotrade, 1988.

[5] Szeredi Péter. Egy logikai alapú magasszintű programozási nyelv. In *Programozási Rendszerek '75*, pages 191–209. Neumann János Számítógéptudományi Társaság, 1975.

[6] Péter Szeredi. A Prolog nyelv. In Iván Futó, editor, *Mesterséges Intelligencia*, pages 390–418. Aula Kiadó, 1999.

[7] P. Roussel. Prolog: Manuel de reference et d'utilisation,. Technical report, Groupe d'Intelligence Artificielle Marseille-Luminy, 1975.

[8] Edit Sántáné-Tóth and Péter Szeredi. PROLOG applications in Hungary. In K. L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 19–32. Academic Press, 1982.

[9] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, 1983.

[10] Farkas Zsuzsa, Futó Iván, Langer Tamás és Szeredi Péter. *MPROLOG programozási nyelv.* Műszaki Kiadó, Budapest, 1988.

[11] Szeredi Péter. *A short history of Prolog in Hungary — a personal account.* Kézirat, 1993

[12] Peter Van Roy *1983–1993: The wonder years of sequential Prolog implementation* Journal of Logic Programming, volume 19–20, pages 385–441. 1994.

[13] David H. D. Warren. Logic Programming and Compiler Writing Software Practice and experience, Vol 10, 97-125 (1980)

Tárgymutató