A logikai programozás és a Prolog

Ásványi Tibor

asvanyi@inf.elte.hu

Tartalomjegyzék

Ha adott egy rendszer logikai formulákkal definiált modellje, kiegészítendő vagy eldöntendő kérdéseket fogalmazhatunk meg bizonyítandó állítások formájában. Kiegészítendő kérdés esetén a megfelelő tulajdonságú objektumok, tervek stb. meghatározása számítási folyamatnak tekinthető, ez a logikai programozás (LP) alapgondolata.

Egy logikai program tehát ugyanúgy, mint egy automatikus tételbizonyító rendszer, axiómák halmaza, amelyhez egy következtető gép is tartozik. Attól elsősorban a következetető gép egyszerűsége, a számítási folyamat átláthatósága, irányíthatósága, garantálható terminálása és nagyobb hatékonysága különbözteti meg.

Ebben a jegyzetben áttekintjük a logikai programozás kialakulását, alapfogalmait, mindmáig legfontosabb megvalósítását, a Prolog nyelvet, a Prolog programozás legalapvetőbb módszereit, a logikai programozás irodalmát, néhány kiterjesztését és újabb irányzatokat. Különös hangsúlyt fektetünk a Prolog programozás gyakorlati módszereire, helyes és hatékony programok készítésére.

1. Bevezetés

A logikai programozás (LP) gyökerei Hilbertig nyúlnak vissza, aki a naiv halmazelmélet ellentmondásossága miatt hirdette meg programját a matematika axiomatizálására és *mechanikus tételbizonyítási módszerek* kidolgozására.

A rezolúciós algoritmus [Rob 65] megszületése ezen kutatás egyik mérföldköve. Az algoritmusból kinyerhető válaszok ugyanis a rezolúciót konstruktív bizonyítási eszközzé teszik, és így a bizonyítás adott tulajdonságú objektumok megkeresésének, vagy kiszámításának tekinthető.[1]

A bizonyítás vagy számítás algoritmusa a rezolúcióban és általában mindenféle mechanikus tételbizonyító rendszerben a feladat logikai (deklaratív) leírásából és a rendszer vezérlési módjából áll:

Algoritmus = Logika + Vezérlés.

Néhány sikertelen amerikai kísérlet után, a hetvenes évek elején Robert Kowalski felismerte, hogy erre az algoritmusfogalomra általános célú programozási nyelv is alapozható lenne. Alain Colmerauer és kollégái tervezték meg az első elfogadható hatékonyságú LP nyelv, a *Prolog* részleteit és ugyanők készítették el hozzá Marseille-ben, 1972-ben az első értelmezőprogramot (interpretert) is. Gyakorlati céljuk egy adatbáziskezelő természetes nyelvű felületének megalkotása volt, és ez is lett a Prolog nyelv első alkalmazása.[2]

Az első hatékony Prolog-fordítót David H. D. Warren és munkatársai Edinborough-ban a hetvenes évek második felében fejlesztették ki. Teljesítménye alapján összemérhető volt a kor legjobb Lisp rendszereivel. Megteremtette a Prolog *de facto* szabványát, meghatározta további fejlődésének irányát és közvetve a nyelvnek a kilencvenes évek közepére kidolgozott *ISO* szabványát is. Sajnos, ez még nem terjed ki a nyelv minden részletére (például a modulrendszerre). Ilyen esetekben a manapság talán legelterjedtebb változatot, a svéd

[1] Az elsőrendű predikátumkalkulus és a rezolúciós algoritmus alapszintű ismeretét a továbbiakban feltételezzük [Fek 90, Nil 82, Kow 79]. A tárgyalás fő vonala azonban enélkül is érthető lesz.

[2] A Prolog név a *Programmation en Logique* (francia) kifejezésből származik.

SICStus Prolog rendszert vesszük alapul [Spl 02], mivel ez a megvalósítás eddig is iránymutatónak bizonyult a szabvány kialakítása szempontjából.

A Prolog a mai napig messze a legszélesebb körben használatos LP nyelv. Viszonylagos sikerének titka abban rejlik, hogy tervezői jó gyakorlati érzékkel kötötték meg azokat a kompromisszumokat, amelyeket elméleti oldalról gyakran és hevesen bírálnak, de amelyek lehetővé teszik – megfelelő programozási stílus mellett – a gyors programfejlesztést kevés hibával, hatékony kódot állítva elő. Valóban, talán nincs még egy olyan programozási nyelv, ahol a jó stílus ennyire alapvető lenne.

Ebben a jegyzetben inkább a Prolog nyelv *alapvető* sajátosságainak és programozási módszereinek bemutatására törekedtünk – amennyire ez az adott keretek között lehetséges volt –, mint a témakör általános áttekintésére, mert ezt a témában járatlan olvasó szempontjából előnyösebbnek ítéltük. Reményeink szerint a jegyzet végén elegendő információ található az ismeretek kiszélesítéséhez szükséges irodalom felkutatásához is.

2. Logikai programok

A logikai program egy modellre vonatkozó *állítások* (axiómák) egy halmaza. Az állítások a modell objektumainak tulajdonságait és kapcsolatait, szaknyelven *relációit* írják le. Ha például adottak állítások, amelyek arra vonatkoznak, hogy ki kinek az apja, ezek együtt az *apja* nevű, *kettő* aritású relációt írják le. Ha megmondjuk, hogy kik a férfiak, akkor az erre vonatkozó állítások együtt a *férfi* nevű, *egy* aritású relációt írják le. (Lásd a (2.1) paragrafust.)

Az állítások egy adott relációt meghatározó részhalmazát *predikátumnak* nevezzük. A program futtatása minden esetben egy az állításokból következő tétel konstruktív bizonyítása, azaz – a logikai programozásban szokásos szóhasználattal – a programnak feltett *kérdés* vagy más néven *cél* megválaszolása. Ennek során a predikátumok eljárásokként működnek.

A ma használatos LP nyelvekben minden, a predikátumokat alkotó *állítás* *tény* vagy *szabály* lehet. Az állításokat és a kérdéseket (azaz célokat) együtt *mondatoknak* nevezzük (a kérdések ugyan nem a program részei). Az egyes LP nyelvekben a program mondatai még a programban leírt relációkra vonatkozó *deklarációk* és a program betöltésekor végrehajtandó *direktívák* is lehetnek. A deklarációk a predikátumok, mint eljárások futtatásának módját pontosítják, míg a direktívák magát a futtatási környezetet állítják be. A Prologban a direktívák és a deklarációk is `:-`*cél*`.` alakúak. Minden mondatot pont, és legalább egy elválasztó karakter[3] zár le. A logikai programok *állításait* a szakirodalomban *definit klózoknak* vagy *Horn klózoknak* is nevezik.

2.1. Tények

A legegyszerűbb logikai programok csak tényeket tartalmaznak.

A tények formailag atomi formulák, például[4]:

```
apja('Ábrahám','Izsák').        apja('Ábrahám','Ismáel').
```

[3] elválasztó, azaz újsor, helyköz, vagy tabulátor karakter

[4] Az alábbi példákban a változóneveket nagybetűvel vagy aláhúzásjellel kezdődő azonosítókkal, a neveket (névkonstansokat) pedig kisbetűvel kezdődő azonosítókkal vagy aposztrófok közé zárt karakterláncokkal jelöltük.

```
apja('Ábrahám','Ismeretlen').
apja('Izsák','Jákób').          apja('Izsák','Ézsaú').

anyja('Sára','Izsák').          anyja('Hágár','Ismáel').
anyja('Rebeka','Jákób').        anyja('Rebeka','Ézsaú').

férfi('Ábrahám').    férfi('Izsák').    férfi('Ismáel').
férfi('Jákób').      férfi('Ézsaú').

'nő'('Sára').          'nő'('Hágár').       'nő'('Rebeka').
'nő'('Ismeretlen').
```

Ezek a program objektumaira vonatkozó egyszerű tulajdonságokat, illetve az objektumok között egyszerű kapcsolatokat fejeznek ki.

Megjegyzés: A példaprogramokat – itt, az általános részben és később is – SICStus Prolog 4.0 nyelven írtuk le. Feltesszük, hogy a forrásfájl első sora a következő:
`%% -*- Mode: Prolog; coding: utf-8 -*-`
Ebből a Prolog környezet tudni fogja, hogy ez egy utf-8 kódolású forrásszöveg. Sajnos a változónevekben és azokban a névkonstansokban, amiket nem teszünk aposztrófok közé, csak a latin-1 karakterkészletben is szereplő karakterek alkalmazhatók. Ez nekünk, magyaroknak, elsősorban az ő,ű,Ő,Ű karakterek használatára nézve jelent megszorítást, mivel pontosan ezek a karaktereink nem szerepelnek a latin-1 jelkészletben. (Ezért kellett pl. a 'nő' relációnevet aposztrófok közé tennünk.)

A legegyszerűbb kérdésekre példák[5]:

```
| ?- apja('Ábrahám','Izsák').
yes
| ?- anyja('Sára','Jákób').
no
```

Ha azt kérdezzük, hogy: *Van-e olyan X, akinek Izsák az apja?* – akkor a fenti programnak megfelelően két megoldást kaphatunk (ahol `X` a kérdés ismeretlenje, konkrétan egy egzisztenciálisan kvantált változó):

```
| ?- apja('Izsák',X).
X = 'Jákób' ? ;    X = 'Ézsaú' ? ;
no
```

A fenti eredmények az `apja('Izsák',X)` cél és a megfelelő tények egyesítéséből származnak.[6]

[5] A `| ?-` a Prolog rendszer (és fejlesztői környezet) parancskérő jele. Ide gépeljük be – a mindig ponttal és `<Enter>`-rel lezárt – utasításainkat (például programbetöltés), és a betöltött programok predikátumaira (eljárásaira) vonatkozó kérdéseinket. A `yes`, illetve a `no` a Prolog rendszer válaszai aszerint, hogy sikerült-e bizonyítania az állítást. Itt a bizonyítás persze csak egylépéses, hiszen vagy megtaláljuk a kérdésnek megfelelő tényt, vagy nem. (Ne keverjük össze az *utasításokat* – amelyek formailag Prolog célok – a *direktívákkal*, amelyek formailag – de nem tartalmilag – a deklarációkkal egyeznek meg (2).)

[6] A kérdőjelek arra vonatkoznak, hogy kérünk-e újabb megoldást. A `;` részünkről azt jelenti, hogy igen. Ha nem kérünk több megoldást, ezt egy `<Enter>`-rel jelezhetjük, és ezt a Prolog fejlesztői környezet a `yes` válasszal nyugtázza. A fenti példában a `no` válasz azt mondja, hogy nincs több megoldás.

Az eddig tárgyalt elemi célok és-kapcsolataival *összetett célokat*, vagy más néven *célsorozatokat* képezhetünk. Az összetett célok elemi kérdéseit pedig *részcélok*nak is nevezzük. Például:

```
| ?- apja('Ábrahám',X), 'nő'(X).
X = 'Ismeretlen' ? ;
no
```

Ilyenkor a célsorozat megoldásai a részcélok közös megoldásai. Úgy is fogalmazhatunk, hogy a később megoldott részcél a korábban megoldott részcél megoldásaira további megszorítást (és így szűrőt is) tartalmaz, már amennyiben van közös változójuk.

A célok változói mindig *egzisztenciálisan* kvantáltak. A program futtatása a kérdés konstruktív megválaszolása, azaz változói lehetséges értékeinek meghatározása.

2.2. Szabályok

Az egynél hosszabb célsorozatok valójában új relációkat határoznak meg. A fenti `apja('Ábrahám',X), 'nő'(X)` célsorozat például a bibliai Ábrahám lányaira vonatkozik. Az új relációnak formálisan is nevet adva a logikai programokat alkotó állítások másik osztályának egy példányához, egy *szabályhoz* juthatunk:

```
'Ábrahám lánya'(X) :- apja('Ábrahám',X), 'nő'(X).
```

Ennek olvasata: `X` *Ábrahám lánya,* **ha** *Ábrahám* `X` *apja* **és** `X` *nő.* A szabályokban szereplő változók, mint a fenti példából is látható, *univerzálisan* kvantáltak. A szabályok általános alakja:

A :- B_1, B_2, ... , B_n. (ahol $n > 0$.)
(A, B_1, ..., B_n atomi formulák.)

Egy szabály következmény része (fent A) a szabály *feje*, feltétel része (fent B_1,...,B_n) pedig a szabály *törzse*. A tényeknek csak fejük van. Úgy is tekinthetjük, hogy törzsük üres, vagy logikailag igaz (`true`).

A tények is tartalmazhatnak (univerzálisan kvantált) változókat. Azt például, hogy mindenki szereti Sárát, így is kifejezhetjük:

```
szereti(_Bárki,'Sára').
```

Így nem kell az univerzum minden elemére felsorolnunk a vonatkozó állítást. Van azonban egy lényeges különbség. Ha most megkérdezzük, hogy ki szereti Sárát:

```
| ?- szereti(Ki,'Sára').
true ? ;
no
```

A `true` válasz azt jelenti, hogy a megtalált megoldás nem példányosította a `Ki` változót. Ezt úgy értelmezhetjük, hogy a válaszban nem behelyettesített változó tetszőlegesen példányosítható. Az már természetes, hogy az általános megoldás mellett további megoldást nem találtunk.

A *relációkat* és az ezeket definiáló *predikátumokat* nevükkel és aritásukkal jellemezhetjük, *név/aritás* formában. (Az *aritás* egy predikátum argumentumainak (paraméterhelyeinek) száma. Az *argumentum* az a szövegpozíció, ahová a paramétert írjuk.)

Egy relációt gyakran nem egyetlen állítás határoz meg. Ilyenkor a reláció az állítások által definiált relációk uniója. Minden változó hatóköre pontosan az őt tartalmazó mondat. Például a `'szülője'/2` reláció az `anyja/2` és az `apja/2` relációk uniója:

```
'szülője'(X,Y) :- anyja(X,Y).
'szülője'(X,Y) :- apja(X,Y).

fia(X,Y) :- 'szülője'(Y,X), férfi(X).

lánya(X,Y) :- 'szülője'(Y,X), 'nő'(X).

'nagyszülője'(X,Y) :- 'szülője'(X,Z), 'szülője'(Z,Y).
```

A `fia/2` reláció viszont a `'szülője'/2` és a `férfi/1` relációk metszete, és hasonlóan adódik a `lánya/2` reláció is. Az utolsó szabály eltér az előzőektől abban, hogy a jobb oldalán új változó is előfordul. Így kétféle olvasata is van:

- Minden `X,Y,Z`-re `'nagyszülője'(X,Y)`, *ha*
  `'szülője'(X,Z)` *és* `'szülője'(Z,Y)`.
- Minden `X,Y`-ra `'nagyszülője'(X,Y)`, *ha*
  van olyan Z, hogy `'szülője'(X,Z)` *és* `'szülője'(Z,Y)`.

A szabályok fenti olvasatait mindegyik esetben *deklaratív* olvasatnak nevezzük. Ezzel szemben áll a *procedurális* olvasat, ami a logikai programok futtatásához, vagyis a konstruktív bizonyítási eljáráshoz kapcsolódik. Minden esetben felteszünk egy kérdést. Ez a bizonyítandó cél(sorozat). Ezután a részcélok bizonyítása, és így kiejtése a feladat. A bizonyítást akkor fejeztük be sikeresen, ha az összes részcélt kiejtettük, azaz bizonyítottuk, és így csak az *üres célsorozat maradt.*

A bizonyítás történhet például *felülről lefelé*, vagy *alulról felfelé*, de egyéb bizonyítási módok is találhatók például [Kow 79]-ben.

1. Az *alulról felfelé* való bizonyítás azt jelenti, hogy a szabályok feltételeivel sorban tényeket egyesítünk, sorban kiejtve ezeket. Így a szabályokból újabb tényeket kapunk, mígnem ezek sorban egyesíthetők lesznek a célban szereplő atomi formulákkal.[7] Amennyiben ezt elértük, a cél következik a programból. Ilyenkor azonban a bizonyítás nehezen irányítható a cél felé, ezért a gyakorlatban a *felülről lefelé* való, azaz a céltól a tények felé haladó következtetések általában jobban kezelhetőek. A Prologban is ezt valósították meg.

2. A *felülről lefelé* való bizonyítás nagy vonalakban azt jelenti, hogy a cél *atomi formuláival* valamilyen *állításfejeket* (tényeket illetve szabályfejeket) egyesítünk.[7] (Az egyesítésben részt vevő mondatoknak a rezolúcióhoz

[7] Az egyesítéshez mindig a legáltalánosabb egyesítő helyettesítést alkalmazzuk.

hasonlóan itt is változóidegeneknek kell lenniük, azaz nem lehet közös változójuk. Ezt az egyesítésben résztvevő állítás, azaz tény vagy szabály változóinak átnevezésével biztosíthatjuk.) Tény egyesítésével kiejtjük a megfelelő részcélt a cél(sorozat)ból, szabályfej egyesítésével pedig a szabály törzsével helyettesítjük azt, az egyesítő helyettesítést mindkét esetben a kapott célsorozatra alkalmazva: Ez egy *célredukciós lépés*.

Ha ilyen célredukciós lépések sorozatával az összes részcélt kiejtettük, akkor bebizonyítottuk a célt. A bizonyítás közben végzett változóhelyettesítéseknek az eredeti cél változóira vonatkozó eredményei adják a konstruktív bizonyítás által meghatározott, a feltételeknek eleget tevő objektumokat (termeket).

Tegyük fel például, hogy adott a $Q_1, Q_2, \ldots, Q_n$ célsorozat. Ha most adott az ettől változóidegen A tény, és $Q_1\theta = A\theta$, ahol a θ az A és a Q_1 atomi formulák legáltalánosabb egyesítő helyettesítése, akkor elég $(Q_2, \ldots, Q_n)\theta$ bizonyítása. Ez a bizonyítandó célra vonatkozó *célredukciós lépés* egyik esete. Ha pedig adott a $Q_1, Q_2, \ldots, Q_n$ célsorozattól változóidegen A`:-`$B_1, \ldots, B_m$ szabály, továbbá θ az A és Q_1 atomi formulák legáltalánosabb egyesítő helyettesítése, akkor elég $(B_1, \ldots, B_m, Q_2, \ldots, Q_n)\theta$ bizonyítása. Ez a bizonyítandó célra vonatkozó *célredukciós lépés* másik esete. Mindkét esetben, ha a célredukciós lépés eredménye a C célsorozat, amelyet a φ helyettesítéssel bizonyíthatunk, azaz $C\varphi$ igaznak bizonyul, akkor ezzel $(Q_1, Q_2, \ldots, Q_n)\theta\varphi$ is bizonyítást nyert, azaz a $\theta\varphi$ helyettesítés az eredeti kérdés megoldása.

Mindkét említett bizonyítási mód helyes és teljes, azaz pontosan azokat a célokat lehet velük bebizonyítani, amelyek a programból következnek.

A fentebb vázolt hatékonysági megfontolás és a Prolog nyelvhez való közelítés okán a továbbiakban a célokat mindig *felülről lefelé* fogjuk bizonyítani. Ehhez a bizonyítási módhoz kapcsolódik az állítások *procedurális* olvasata is:

- Az A tény azt jelenti, hogy az A részcél közvetlenül megoldható.
- Az A `:-` B_1, B_2, ... , B_n szabály azt jelenti, hogy az A részcél megoldható a B_1, B_2, ... , B_n célok megoldásával.

A fentiek szerint a `'nagyszülõje'/2` predikátum egyetlen állításának procedurális olvasata a következő: *A* `'nagyszülõje'(X,Y)` *cél megoldásához oldjuk meg a* `'szülõje'(X,Z)` *és a* `'szülõje'(Z,Y)` *célokat.*

Egy részcéllal esetleg több szabályfej illetve tény is egyesíthető. A lehetséges levezetések ezért faszerűen elágazhatnak. Az így létrejövő faszerkezet gyökere az eredeti cél, csúcsai a levezetés során előállított célsorozatok, élei a célredukciós lépések, és levelei az üres célsorozatok, a *megoldáslevelek*, illetve az olyan ágak, ahol a célredukcióra kiválasztott részcél nem redukálható, a *fail-levelek*. Minden megoldáslevélhez az eredeti kérdés egy-egy megoldása tartozik. Az így meghatározott fát *levezetési* vagy *bizonyítási fának*, illetve *keresési fának* vagy *keresési térnek* is nevezik.

Az alábbi példában a fa szintjeit a sorok behúzása jelzi. Minden lépésben a célsorozat első részcélját választjuk ki, és ezt próbáljuk ténnyel való egyesítés esetén kiejteni, illetve szabályfejjel való egyesítés esetén a szabálytörzzsel

helyettesíteni. Ez a *legbal* részcélkiválasztási módszer. A keresési fában a célredukciós lépések során alkalmazott változóhelyettesítések közül mindig csak az aktuális célsorozat változóira vonatkozó helyettesítéseket jelöljük, mégpedig *változó*<-*helyettesítő_term* formában. Ha egy részcél és egy szabályfej vagy tény egyesítése során két változót kell egyesítenünk, az egyszerűség kedvéért mindig a részcél változóját helyettesítjük a szabályfej vagy tény változója helyébe.

Világos, hogy egyes lépésekben másik részcélt is választhatnánk a célredukcióhoz, és akkor az alábbitól különböző keresési fát kapnánk.

Kérdés, hogy nem kapnánk-e más megoldást. Szerencsére igaz az, hogy a különböző részcélkiválasztási módszerek ugyan különböző keresési fákhoz vezetnek, de a fák ugyanazokat a megoldásokat tartalmazzák. Sajnos a különböző keresési fák általában különböző méretűek, mint azt bárki maga is ellenőrizheti, ha az alábbi példában más részcélkiválasztási módszert választ.

```
?- 'nagyszülője'('Ábrahám',X).
     'szülője'('Ábrahám',Z),'szülője'(Z,X).
        anyja('Ábrahám',Z),'szülője'(Z,X).  % meghiúsul
        apja('Ábrahám',Z),'szülője'(Z,X).
          { Z <- 'Izsák' }
            'szülője'('Izsák',X).
              anyja('Izsák',X).           % meghiúsul
              apja('Izsák',X).
                { X <- 'Jákób' }        % 1. megoldás
                { X <- 'Ézsaú' }        % 2. megoldás
          { Z <- 'Ismáel' }
            'szülője'('Ismáel',X).
              anyja('Ismáel',X).          % meghiúsul
              apja('Ismáel',X).           % meghiúsul
          { Z <- 'Ismeretlen' }
            'szülője'('Ismeretlen',X).
              anyja('Ismeretlen',X).        % meghiúsul
              apja('Ismeretlen',X).         % meghiúsul
```

A fenti `'nagyszülője'('Ábrahám',X)` kérdésre a *legbal* részcélkiválasztó módszer a lehető legkisebb keresési fát állítja elő. Könnyen látható azonban, hogy például a `'nagyszülője'(X,'Jákób')` kérdésre egy olyan módszer szolgáltatná a legkisebb keresési fát, amely mindig a *legjobboldali* részcélt választaná. Tapasztalatainkat összegezve azt mondhatjuk, hogy mivel a `'nagyszülője'(X,Y)` típusú kérdésekre az első célredukciós lépés után mindig egy `'szülője'(X,Z), 'szülője'(Z,Y)` típusú célsorozatot kapunk, a továbbiakban mindig azt a részcélt érdemes választani, amelyben kevesebb a behelyettesítetlen változó. (A behelyettesítetlen változó olyan változó, amelyet még nem helyettesítettünk állandó vagy összetett termmel.) Ez a módszer más célsorozatoknál is figyelemreméltó, mert a kevesebb behelyettesítetlen változót tartalmazó részcél általában kevésbé elágazó keresési fához vezet. Ez természetesen csak heurisztika.

Általában az is előfordulhat, hogy ugyanarra a kérdésre az egyik keresési fa véges, míg a másiknak vannak végtelen ágai. Ilyenkor a megoldás keresése elkalandozhat egy végtelen ágon, ami végtelen kereséshez vezethet. Ez azt jelenti, hogy részcélkiválasztási módszerünket úgy célszerű megválasztani, hogy a

keresési fa lehetőleg véges legyen. Ha azonban – mint jelen esetben – nincsenek rekurzív szabályaink, a keresési fa véges.

2.3. Rekurzív szabályok

Tegyük fel, hogy azt szeretnénk leírni, X mikor őse Y-nak. Világos, hogy akkor őse, ha szülője, vagy valamelyik ősének a szülője. Látható, hogy a relációnak két esete van, és az egyik eset rekurzív. Ezt ennek megfelelően egy nem rekurzív és egy rekurzív szabállyal fejezhetjük ki:

```
'őse'(X,Y) :- 'szülője'(X,Y).
'őse'(X,Y) :- 'szülője'(X,Z), 'őse'(Z,Y).
```

Ha most úgy képzeljük, hogy a `'szülője'/2` reláció egy irányított kört nem tartalmazó gráfot ír le, amelynek éleit éppen a `'szülője'/2` reláció adja meg, csúcsait pedig az élek végpontjai, akkor az `'őse'/2` reláció ezen a gráfon a nem üres, irányított utakat adja meg. Ezek az utak a *szülője* kapcsolat természeténél fogva körmentesek és felülről korlátos hosszúságúak.

Tekintsük most az `'őse'('Ábrahám',X)` kérdést. Könnyen ellenőrizhető, hogy a keresőfa előállításához célszerű a legbal részcélkiválasztási módszert alkalmazni, mert így a `'szülője'(X,Y)` célokban `X` már mindig ismert lesz, tehát a keresőfa nem lesz túl széles, azaz az elágazások száma az egyes csúcsokban nem lesz túl nagy, és a keresőfa véges is lesz, hiszen minden rekurzív hívásban egy, az `'Ábrahám'` pontból induló irányított út egy élét dogozzuk fel, és sem végtelen irányított utak, sem irányított körök nincsenek a gráfban.

Ha most például az `'őse'(X,'Izsák')` kérdést tekintjük, és a túl széles keresőfák ellen védekezésül az adódó célsorozatokból a jobb oldali részcélt választjuk, mivel abban a második paraméter már meghatározott, akkor végtelen keresési fát kapunk. Ha a célredukcióban következetesen a második szabályt részesítjük előnyben az elsővel szemben, rögtön a végtelen ágra kerülünk. Ha következetesen az első szabályt részesítjük előnyben, először megtaláljuk a megoldásokat, de utána, ha tovább keresünk, a végtelen ágra kerülünk. Mivel általánosságban egy számítás nem tudhatja, hogy mikor találja meg az utolsó megoldást, csak azt, hogy megoldást talált, illetve azt, hogy bejárta-e a keresési teret, ez nagyon kellemetlen tud lenni. Ha erre a kérdésre is a *legbal* részcélkiválasztási módszert alkalmazzuk, akkor az első `'szülője'(X,Y)` hívás lerögzít valahol a gráfon egy kezdőélet, és a további útkeresés az `'Izsák'` csúcshoz innét folytatódik. Ily módon a keresési fa szélesebb lesz, mint az előbb, de véges marad.

Megfogalmazhatnánk, mint heurisztikát, azt, hogy egy célsorozatból mindig a nem rekurzív predikátummal megoldható részcélt válasszuk. Ez gyakran segítene, de ilyenkor egyrészt felmerülhet, hogy több heurisztika esetén hogyan rangsoroljunk, másrészt mi a helyzet, ha több rekurzív relációra is van részcél a célsorozatban, és mindegyik végtelen keresési fához vezethet? Például a fenti `'őse'/2` relációt a következőképp is átfogalmazhatnánk:

```
'őse00'(X,Y) :- 'szülője'(X,Y).
'őse00'(X,Y) :- 'őse00'(X,Z), 'őse00'(Z,Y).
```

Ebből azt a tanulságot vonhatjuk le, hogy a várható kérdésekre jól illeszkedő formalizálás fontosabb, mint a túlfinomult, és ezért a programozó számára követhetetlen működésű részcélkiválasztó módszer. A módszer egyszerűsége haté-

konysági szempontból is fontos lehet, és különösen előtérbe kerül, ha LP nyelvet szeretnénk tervezni.

3. Bevezetés a Prolog nyelvbe

A Prolog nyelv tervezői a fentiek miatt a *legbal* részcélkiválasztási módszer mellett döntöttek (ld. 10. oldal). A választható állításokat pedig a Prolog gép a felírás sorrendjében próbálja ki, egy visszalépéses kereséssel járva be a keresési fát, amelynek ágait természetesen nem építi fel előre, és visszalépéskor le is bontja, így tartva korlátok között a futó program memóriaigényét.

Ha már a Prolog gépet így határoztuk meg, maga a Prolog nyelv inkább tekinthető LP nyelvnek, mint egy tételbizonyító programhoz használt formulaleíró nyelvnek. Az ugyanis, hogy egy logikai formális nyelv melyik a kettő közül, a szerző véleménye szerint a hozzátartozó következtető gép kifinomultságán múlik. Ez a gép a Prolog esetében a lehető legegyszerűbb, és így működése könnyen követhető, a következtetés egyszerű eszközökkel irányítható, annak végessége könnyen biztosítható, hatékonysága előre kiszámítható. Ezek a feltételek pedig alapvetőek egy programozási nyelv esetében. Mint látni fogjuk, a Prolog részletes kidolgozása szerint is egy magas szintű programozási nyelv.

A Prolog programok helyességét mindig a megoldandó célokra vonatkoztatjuk. A parciális helyesség egyszerűen a formalizálás helyességét jelenti. A megállási probléma legegyszerűbb megoldása, ha bebizonyítjuk a kívánt típusú célsorozatokra a keresési fák végességét. Ha nincs rekurzív szabályunk, a közvetett rekurziót is ide értve, akkor a fa végessége automatikusan teljesül. Ha vannak rekurzív szabályok, akkor hasznos például, ha találunk egy nemnegatív egész értékű függvényt, ami a keresési fa csúcsain értelmezett, a fában lefelé haladva nem növekszik, és minden rekurzív hívásnál szigorúan csökken.

Az `'õse'/2` reláció első formalizálása esetén például a feldolgozatlan út (11. oldal) lehetséges legnagyobb hossza egy megfelelő függvény. Ez ugyanis tetszőleges `'õse'(X,Y)` típusú kérdés esetén a `'szülője'(X,Z)` típusú részcélok feldolgozásával mindig eggyel csökken. Mivel nem lehet negatív, a keresési fa véges. A második formalizálás (ld. az `'õse00'/2` predikátumot (11. oldal)) esetében végtelen a keresési fa, nem is találhatunk megfelelő függvényt.

A programozás rendesen a *fejlesztői környezetben* történik, egy belső vagy külső szövegszerkesztő felhasználásával. Az is lehet, hogy a Prolog rendszer (például SICStus) és a szövegszerkesztő (például emacs) egy mindkettő által támogatott szabványos felületen keresztül érintkezik, és a Prolog környezet a szövegszerkesztőn belül indul el, annak lehetőségeit (automatikus tördelés, színezés; fordítás, betöltés, forráskód szintű nyomkövetés stb.) kihasználva. A Prolog rendszer rendesen egy parancsablakban indul el (bár grafikus felülete is lehet). Ide gépeljük be célsorozatainkat, azaz programbetöltő, fordító és egyéb utasításainkat, valamint a betöltött programokra vonatkozó kérdéseinket. Alapértelmezés szerint (a grafikus, fájl és egyéb különleges felületektől eltekintve) itt kapjuk meg a rendszer válaszait is. Futtatható fájl készítése [Spl 02] esetén a programnak legalább egy, a betöltéskor végrehajtandó célt (direktívát (2)) kell tartalmaznia; ez(ek) vezérli(k) működését. Egy program tetszőlegesen sok fájlból állhat. A fejlesztői környezetbe a programok többféleképpen is betölthetők. Az értelmező módban való betöltéshez egyszerűen megadjuk a Prolog parancskérő

jelénél (`?-`) a betöltendő forrásállományok listáját (nevük rendesen *.pl alakú), például `?- [a,b,c].`, ha a program az `a.pl`, `b.pl` és `c.pl` állományokból áll. A betöltött program tetszőleges predikátuma tesztelhető, azaz feltehetünk rá vontakozó kérdéseket.
Tegyük fel, hogy eddigi predikátumaink az `ny.pl` fájlban vannak:

```
| ?- [ny].
{consulting /home/at/pp/ny/ny.pl...}
{consulted /home/at/pp/ny/ny.pl in module user,
 70 msec 4864 bytes}
yes
| ?- 'őse'(Ős,'Jákób').
Ős = 'Rebeka' ? ;    Ős = 'Izsák' ? ;
Ős = 'Sára' ? ;    Ős = 'Ábrahám' ? ;
no
```

A betöltött program tetszőleges predikátuma *közvetlenül* tesztelhető tehát, mivel a Prologban szimbolikusan tetszőleges adatszerkezetet leírhatunk.

4. A logikai programok objektumai

A logikai programok objektumait *termekkel* írjuk le. Nincs szabványos LP termfogalom, de ez minden LP nyelvben hasonló, ezért – az ismétléseket elkerülendő – az ISO Prolog szabványnak megfelelően tárgyaljuk.

Egy Prolog term lehet *egyszerű term*, azaz *változó* vagy *állandó* (*konstans*); továbbá *összetett term*, azaz *struktúra*.

Egy *állandó* lehet *név* vagy *szám*, ami lehet *egész* vagy *lebegőpontos szám*. A számokat a szokásos módon írhatjuk.

A *nevek* kisbetűvel kezdődő azonosítók, vagy aposztrófok közé zárt karaktersorozatok is lehetnek. Nevek még a `+-*/\^<>=~:.?@#&$` karakterekből álló speciális jelsorozatok. (Például: `=<  @>=  ?-  *$` stb.) Végül van még négy különleges név, ezek a `;` (or,else,elsif), a `!` (cut), a `[]` (nil) és a `{}` (empty).

A *változónevek* nagybetűvel, vagy aláhúzásjellel kezdődő azonosítók. A *változók* a matematikai egyenletek *ismeretleneinek* felelnek meg, azaz egy-egy ismeretlen objektumot jelölnek. A számítások célja a kérdések változói, azaz ismeretlenei lehetséges értékeinek meghatározása. Ilyen módon nem lehetséges és nincs is értelme, hogy értékadó utasítást írjunk. Ha egy logikai változót behelyettesítünk, ez a továbbiakban a helyettesítő termtől megkülönböztethetetlen, hacsak visza nem lépünk a helyettesítés elé, amikor is a változó elveszti értékét. Ez a procedurális nyelvekhez szokott programozónak idegen lehet, de a logikai programozás gyakorlása során természetessé válik. Azzal, hogy értékadás helyett változóhelyettesítéssel dolgozunk, talán a legveszélyesebb programozási hibaforrást küszöböltük ki.

Széles körben elterjedt jelölési szokás szerint azokat a változókat, amelyek helyettesítési értéke érdektelen, aláhúzásjellel kezdődő nevekkel jelöljük:

```
apa(Valaki) :- apja(Valaki,_Gyermek).
```

Ha programunk egy mondatában egy változónak egyetlen előfordulása van, ér-

téke érdektelen, ugyanis nincs hova továbbítanunk.[8] Az egyetlen aláhúzásjelből álló *névtelen változó* minden előfordulása – még egyetlen mondaton belül is – más-más logikai változót jelöl, ez tehát csak érdektelen változókat jelölhet. *Minden más változó hatóköre az őt tartalmazó mondat.* Például:

```
apa_és_fiú(Valaki) :-
    apja(Valaki,_), 'szülője'(_,Valaki).
```

Az *összetett termek* strukturált információ ábrázolására szolgálnak. Alakjuk $f(t_1, ..., t_n)$, ahol f tetszőleges *név*, $t_1, ..., t_n$ tetszőleges termek, és az f/n *függvényszimbólum* vagy *funktor* a $t_1, ..., t_n$ termeket egyetlen összetett termbe fogja össze.[9] Bármelyik t_i term lehet állandó, változó, vagy összetett term is. A változómentes termeket *alaptermeknek* nevezzük. (Az alaptermek ezért állandók vagy alaptermekből felépített összetett termek.)

Összetett termek segítségével tehát rekurzív adatszerkezeteket is leírhatunk. Egy üres fát reprezentálhatunk például az `o` állandóval, egy nem üres fát pedig

`fa(` *gyökér*, f_1`, ... ,` f_n `)`

alakban, ahol minden f_i egy-egy részfát jelöl.

A `fa(4,fa(2,fa(1,o,o),fa(3,o,o)),fa(5,o,o))` term ekkor egy kettő mélységű bináris rendezőfát ábrázol.

5. Listakezelés rekurzív LP módszerekkel

A *listákat* ezután unáris fáknak tekinthetjük. Egy *valódi lista*[10] lehet *üres*, vagy *nem üres*. A *nem üres listákat* szabványosan a `'.'/2` függvényszimbólum segítségével építjük fel, a `[]` *üres listából* kiindulva (`[]` olvasata `nil`). Egy `.(X,Xs)` term pontosan akkor *valódi lista*, ha `Xs` is valódi lista.

Az `[1,2,3]` listát így `.(1,.(2,.(3,[])))` alakban írhatnánk le. Tekintettel a listák jelentőségére, a programok olvashatóságát növelendő egy első ránézésre talán kicsit bonyolult, de nagyon hasznos jelölésrendszert vezettek be.

A nem üres listák `.(X,Xs)` konstruktorát `[X|Xs]` alakban, a listák első néhány elemét pedig vesszővel elválasztva írhatjuk.
Az `[X1,X2,...,Xn|[]]` jelölésből `[]` elhagyható. Eszerint

```
| ?- .(X,Xs)==[X|Xs], [X1|[X2|Xs]]==[X1,X2|Xs],
     [X1,X2,X3|[]]==[X1,X2,X3],
      .(1,.(2,.(3,[])))==[1,2,3].
true
```

(Az `X==Y` Prolog hívás pontosan akkor sikeres, hogy az `X` és `Y` termek *azonosak* (8.2).) Az `[X|Xs]` valódi lista első eleme vagy más néven feje, az `X`, tetszőleges term, akár behelyettesítetlen változó vagy lista is lehet. Az `[X|Xs]` valódi lista

[8] Például a SICStus Prolog fordító feltételezi, hogy a nagybetűvel kezdődő azonosítók lényeges változókat jelölnek, ezért ezekre, ha egy állításban csak egy előfordulást talál, figyelmeztetést küld, segítve ezzel a gépelési hibák kiküszöbölését. A másik oldalon, ha egy a SICStus Prolog parancskérő jelénél (lásd a 13. oldalon) begépelt kérdésben egy változó neve aláhúzásjellel kezdődik, a rendszer cél megoldása után ennek értékét nem írja automatikusan ki.

[9] A függvényszimbólumokat tehát – a predikátumokhoz vagy relációkhoz hasonlóan – nevükkel és aritásukkal jellemezzük *név/aritás* formában.

[10] *Valódi lista*: szemben az alább bevezetendő *részleges listákkal.*

maradéka `Xs`, maga is valódi lista kell legyen. Például `[a|[_]]` lista, de `[[_]|a]` nem az, mert az `a` név nem lista.

Egy term pontosan akkor *részleges lista* (parciális lista), ha behelyettesítetlen változó, vagy `[X|Xs]` alakú, ahol `Xs` részleges lista. Eszerint egy term pontosan akkor *részleges lista*, ha ő maga nem valódi lista, de megfelelő változóhelyettesítéssel azzá tehető.

A valódi és a részleges listákat együtt a továbbiakban *listáknak* nevezzük. Mint hamarosan látni fogjuk, a listák mindkét osztálya alapvető jelentőségű.

Végül, az *alaplista* egy változómentes lista, azaz olyan lista, ami egyúttal alapterm is. Az alaplisták tehát a valódi listák egy valódi részhalmazát adják.

Tegyük fel például, hogy `Xs` behelyettesítetlen változó. Ekkor `Xs` és `[1,2|Xs]` *részleges listák*, de `[Xs]` és `[1,2,Xs]` egy, illetve három elemű *valódi listák*. Az előbbi négy lista egyike sem alaplista. Ha viszont `Xs==3`, akkor `Xs` és `[1,2|Xs]` sem listák, de `[Xs]` és `[1,2,Xs]` alaplisták. Az alábbi programmal

```
list([]).
list([_X|Xs]) :- list(Xs).
```

a `list(L)` cél *valódi listával* paraméterezve változóhelyettesítés nélkül sikeres lesz. *Részleges listára* végtelen sok megoldása van, ezek az `L`-ből példányosítható valódi listák legáltalánosabb esetei. Például:

```
| ?- list(L).
L = [] ? ;    L = [_A] ? ;    L = [_A,_B] ? ;
L = [_A,_B,_C] ? ;    L = [_A,_B,_C,_D] ? <Enter>
yes
```

A `list(L)` cél, ha nem listával paraméterezzük, meghiúsul.

A fenti jelölésekkel nézzünk most néhány predikátumot. Ezek segítségével bemutatjuk a listakezelő (és általában a rekurzív adatszerkezeteket kezelő) logikai programokban szokásos alapvető programozási módszereket.

5.1. A rekurzív keresés

Tekintsük először az *eleme* relációt klasszikus leírását.
(A `%` jel után a sor hátralevő része a Prologban megjegyzésnek számít.)

```
% member_(X,Xs) :- X eleme az Xs listának.
member_(X,[X|_Xs]).
member_(X,[_X|Xs]) :- member_(X,Xs).
```

Egy lista elemeit tehát a lista első eleme, és a maradékában lévő elemek adják. A predikátumhívás előfeltételének tekinthetjük, hogy ezen kérdés második paramétere egy valódi lista legyen. Részleges lista esetén ugyanis a vonatkozó cél keresési tere végtelen lesz. Valódi listánál a rekurzióban a lista hosszának csökkenése garantálja a keresési fa végességét. Ezt mutatják a következő futtatási eredmények is. (Próbáljuk meg felrajzolni az alábbi célok keresési fáját!)

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

A megfelelő elemet itt közvetlenül, vagy rekurzió segítségével közvetetten próbáljuk megkeresni. Az itt alkalmazott programozási módszert ezért *rekurzív keresésnek* nevezzük. (Figyeljük meg, miképpen alkalmaztuk a *rekurzív keresést* eddigi rekurzív predikátumainkban.)

5.2. Az eredmény fokozatos közelítése

A következő predikátum a listaösszefűzés klasszikus programja. Használhatjuk egy lista részekre bontására is. Vegyük észre, hogy két állításunk van, aszerint, hogy az első lista üres-e vagy sem. Ez gyakori szervezési elv a predikátumok megfogalmazásakor.

```
% append_(Xs,Ys,XsYs) :-
%     Az Xs és Ys listák összefűzöttje az XsYs lista.
append_([],Ys,Ys).
append_([X|Xs],Ys,[X|Zs]) :- append_(Xs,Ys,Zs).
```

Az üres lista és bármely `Ys` lista összefűzésének eredménye `Ys`. Egy nem üres `[X|Xs]` lista és bármely `Ys` lista összefűzésének eredményében az első elem `X`, míg a maradék az `Xs` és `Ys` összefűzésének eredménye. Mivel a rekurzióban az első és a harmadik lista hossza is csökken, a keresési fa végességéhez elegendő, hogy a hívásban ezen paraméterek valamelyike valódi lista legyen. A második paraméter mindegyik esetben lehet valódi vagy részleges lista is.

Vegyük most az `append_([1,2],[3,4],Zs)` kérdés keresési fáját. (Feltesszük, hogy az LP rendszer az állítások minden `V` változóját minden célredukciós lépésben `V`i-re nevezi át, ahol i a célredukciós lépés sorszáma.)

```
append_([1,2],[3,4],Zs)
  { Zs <- [1|Zs1] }
    append_([2],[3,4],Zs1)
      { Zs1 <- [2|Zs2] }
        append_([],[3,4],Zs2)
          { Zs2 <- [3,4] }            % (megoldás)

% Végeredményben:
Zs=[1|Zs1]=[1|[2|Zs2]]=[1,2|Zs2]=[1,2|[3,4]]=[1,2,3,4]
```

Egyrészt tehát a keresési fa lineáris, mert mindenütt csak *egy* állítás feje illeszkedett a célra. Másrészt az összefűzött listát több lépésben, részleges listákon keresztül közelítettük. Minden lépésben – az aktuális részleges lista végére vonatkozó újabb változóhelyettesítéssel – a kimenő lista egyre nagyobb részét határoztuk meg, mígnem megkaptuk a végeredményként adódó összefűzött listát.

Ezt a programozási módszert, amikor tehát az eredmény adatszerkezetet felülről lefelé építjük fel, először bizonyos részleteit határozatlanul hagyva, majd ezeket a részleteket újabb és újabb változóhelyettesítésekkel egyre pontosítva, az *eredmény fokozatos közelítésének* nevezzük.

(Próbáljuk meg felrajzolni az alábbi kérdések keresési fáját! Figyeljük meg, hogyan érvényesül a számításokban az eredmény fokozatos közelítése! Az első feladatban a végeredmény is részleges lista, a másodikban pedig egy lista kettévágására elágazó keresési fát, és eredményül a lehetséges vágásokat kapjuk.)

```
| ?- append_([1,2],Ys,Zs).
Zs = [1,2|Ys] ? ;
no
| ?- append_(Xs,Ys,[1,2,3]).
Xs = [],  Ys = [1,2,3] ? ;
Xs = [1],  Ys = [2,3] ? ;
Xs = [1,2],  Ys = [3] ? ;
Xs = [1,2,3],  Ys = [] ? ;
no
```

5.3. Az akkumulátor módszer

A rekurzív predikátumokban alkalmazott harmadik számítási eljárás az *akkumulátor módszer*. Ezt a következő példaprogram illusztrálja.

```
% rev_app(Xs,Ys,Zs) :-
%     Az Xs lista fordítottját az Ys listával
%     összefűzve kapjuk a Zs listát.
rev_app([],Ys,Ys).
rev_app([X|Xs],Ys,Zs) :- rev_app(Xs,[X|Ys],Zs).
```

Az üres lista fordítottjának és bármely `Ys` listának az összefűzöttje az `Ys` lista. Egy nem üres `[X|Xs]` lista fordítottjának és bármely `Ys` listának az összefűzöttje az `Xs` lista fordítottjának és az `[X|Ys]` listának az összefűzöttje.

Mivel a rekurzióban az első lista hossza csökken, a keresési fa végességéhez elegendő, hogy a hívás első paramétere valódi lista legyen. Ha az első paraméter részleges lista, a keresési fa nyilván végtelen lesz. Második és a harmadik paraméterként tetszőleges listákat várunk, ezek tehát lehetnek valódi vagy részleges listák, esetleg behelyettesítetlen változók is.
Nézzük most a `rev_app([1,2],[3,4],Zs)` kérdés keresési fáját.

```
rev_app([1,2],[3,4],Zs)
  rev_app([2],[1,3,4],Zs)
    rev_app([],[2,1,3,4],Zs)
      { Zs <- [2,1,3,4] }                % megoldás
```

A második, az *akkumulátor argumentumban* építjük fel az eredmény adatszerkezetet alulról fölfelé. Így az eredmény a rekurzió alján ebben az akkumulátor argumentumban áll elő. Az eredmény visszaadásásához ezért szükségünk van egy harmadik, *alagút argumentumra*. A harmadik paraméter tehát változatlanul végigmegy egészen a rekurzió aljáig, és csak itt egyesítjük az akkumulátor értékével. Az *akkumulátor* és *alagút* argumentumokat együtt *akkumulátor párnak* nevezzük, mert a programokban mindig így, párban fordulnak elő.

A gyakorlati programokban gyakran írunk olyan predikátumokat, amelyekben – egyetlen predikátum argumentumait tekintve – több akkumulátor párt, illetve fokozatos közelítéssel előállított paramétert is találhatunk.

Megfigyelhetjük például, hogy a `?- append_(Xs,Ys,[1,2,3])` célban – míg `Xs` fokozatos közelítéssel áll elő –, addig `Ys` alagútként, a harmadik paraméter pedig valamiféle negatív akkumulátorként viselkedik. (Nem *felépíti*, hanem *lebontja* a bemenő paraméterből az alagúton visszaküldeni kívánt adatszerkezetet.)

5.4. Az általánosítás módszere

Az alábbi predikátum segítségével az *általánosítás* módszerére adunk példát. Ilyenkor a problémát egy általánosabb feladatra való visszavezetéssel oldjuk meg. Ez természetesen nem a logikai programozás sajátsága; az emberi, illetve gépi problémamegoldás minden területén használjuk. Programkészítésnél gyakran hasznos, ha egy rekurzív eljárással az eredeti feladatnál általánosabbat oldunk meg, majd megfelelően paraméterezve hívjuk meg.

```
% reverse(Xs,Ys) :- Az Xs lista fordítottja az Ys.
reverse(Xs,Ys) :- rev_app(Xs,[],Ys).
```

Az `Xs` lista fordítottja az `Ys` lista, ha `Xs` fordítottját az üres listával összefűzve az `Ys` listát kapjuk.

Ez a predikátum is örökli tehát `rev_app/3`-tól azt a tuladonságot, hogy ha a rá vonatkozó kérdés első paramétere valódi lista, akkor ezen cél keresési tere véges lesz; míg ha a rá vonatkozó cél első paramétere részleges lista, akkor ezen cél keresési tere végtelen lesz.

Feltéve, hogy `Xs` behelyettesítetlen változó, a `reverse([1,2,3],Xs)` cél rendben kiszámolja egyetlen, `Xs=[3,2,1]` megoldását, és keresési tere is véges lesz. Természetesen a `reverse(Xs,[1,2,3])` kérdés egyetlen megoldása is ez, de keresési tere végtelen. A Prologban, mivel a `rev_app/3` kódjában a rekurziót leállító tény megelőzi a rekurzív szabályt, első válaszként ez utóbbi kérdésre is megtaláljuk ezt a megoldást, de ha további megoldást is keresünk, akkor állandóan visszalépünk, és egyre hosszabb változólistákat próbálunk a második paraméterrel egyesíteni, ami persze mindig sikertelen lesz. Ezzel szemben:

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

Általában a fák kezelésénél is az itt megismert négy rekurziófüggő programozási módszert használjuk.

6. A Prolog gép

A teljes Prolog nyelv metalogikai és logikán kívüli nyelvi elemeket is tartalmaz. Ebben fejezetben az ezektől mentes *tiszta Prolog* általános végrehajtási mechanizmusát ismertetjük, majd ezt finomítjuk tovább, a mai Prolog megvalósítások irányában.

A tiszta Prolog program felhasználói predikátumok halmaza, amelyre kérdéseket tehetünk fel. Minden predikátum állítások (tények és szabályok) sorozata. Egyetlen szabálytörzs sem tartalmaz olyan célt, amely beépített eljárást (predikátumot) hívna meg.

6.1. Tiszta Prolog programok végrehajtása

Mint már említettük, a Prolog gép egy program végrehajtása során a feltett kérdés keresési fáját járja be, mégpedig preorder módon (3). Közben mindig csak a fa aktuális ágát tartja nyilván. Az aktuális ág minden csúcsát a hozzá tartozó célsorozat, és a célsorozat első elemi részcéljához (predikátumhívásához) tartozó

predikátumnak a még ki nem próbált állításai címkézik, ez utóbbiak felírásuk sorrendjében. Az aktuális ág éleit a megfelelő célredukciós lépéshez felhasznált tény vagy szabály mellett a kapcsolódó egyesítő helyettesítésben behelyettesített célsorozatbeli változók halmaza címkézi. Vegyük észre, hogy nem szükséges az egyesítő helyettesítések feljegyzése, és – mint látni fogjuk – explicit kiszámítása sem, csak végrehajtásuk. A tiszta Prolog programok végrehajtásának alábbi algoritmusában – mint később meg is indokoljuk (6.2) – az *egyesítés* helyett az *illesztés* szót használjuk.

(1) Kezdetben a keresési fa gyökerét az eredeti kérdés (célsorozat) címkézi. Ekkor a keresési fa gyökere az aktuális csúcs. (Mindig a gyökértől legtávolabbi csúcs lesz az aktuális).
(2) Ha az aktuális csúcshoz az üres célsorozat tartozik, megoldást találtunk. Ez az eredeti kérdés változóinak aktuális helyettesítése. Ilyenkor a megoldást kiírjuk, majd megkérdezzük a felhasználót, hogy kér-e további megoldást. Ha igen, akkor (9)-től folytatjuk (visszalépés). Ha nem, akkor *sikeresen befejeztük a keresést.*
(3) Ha az aktuális csúcshoz nem az üres célsorozat tartozik, akkor most az aktuális csúcsot az azt címkéző célsorozat első elemi részcéljához tartozó predikátum állításlistájával is felcímkézzük. Az állítások mindegyikének változóit átnevezzük úgy, hogy a kérdéstől változóidegen legyen.
(4) Ha az aktuális állításlista üres, akkor zsákutcába jutottunk, és (9)-től folytatjuk (visszalépés).
(5) q legyen az aktuális állításlista első eleme. Töröljük q-t e listáról.
(6) Az aktuális célsorozat első predikátumhívását megpróbáljuk illeszteni q fejével (6.2). (Egy tény feje önmaga, törzse pedig üres. Egy *a:-b,...,z.* állítás feje *a*, törzse pedig a *b,...,z* célsorozat.)
(7) Ha az illesztés meghiúsul, (4)-től folytatjuk.
(8) Ha az illesztés sikeres, egy célredukciós lépést végzünk: Az illesztés során behelyettesített változók minden, célsorozatbeli illetve állításbeli előfordulása is megfelelőképpen helyettesítődik. (Ezt rendesen a változók láncolt ábrázolása biztosítja.) A keresési fa éppen nyilvántartott ágát egy új csúccsal bővítjük. Az ide vezető élet az aktuális célsorozatnak az illesztés során behelyettesített változói halmazával és q-val címkézzük. Az új csúcs első címkéjét úgy kapjuk, hogy az aktuális célsorozat első predikátumhívását a q törzsével helyettesítjük, ahol tehát az illesztő helyettesítést már minden résztvevőn elvégeztük. Ezután az új csúcs lesz az aktuális, és (2)-től folytatjuk.
(9) *Visszalépés:* Ha az aktuális csúcs a keresési fa gyökere, a program végrehajtása *meghiúsulással ér véget.*
Különben töröljük a keresési fa aktuális csúcsát, és a fölötte lévő lesz az új aktuális csúcs. Közben a két csúcsot összekötő élet címkéző változóhalmaz elemeinek behelyettesítését töröljük. Ezután (4)-től folytatjuk.

6.2. Az illesztés algoritmusa

Mint láttuk, a Prolog gép működésének kulcsműveletei a célredukció és a visszalépés. A célredukció kulcsa viszont a kiválasztott részcél és állításfej egyesítése a legáltalánosabb egyesítő helyettesítés algoritmusa szerint. Mivel azonos funktornevű és aritású atomi formulákat kell egyesítenünk, a feladat azonos hoszúságú termsorozatok egyesítése. ISO Prologban, és így a modern Prolog változatokban

is, hatékonysági okokból az egyesítő algoritmus alábbi, egyszerűsített változata használatos. A továbbiakban ezt nevezzük *illesztésnek*:

(1) Ha az illesztendő termsorozatok üresek, készen vagyunk, különben illesztjük a két sorozat első elemét (2) szerint. Ezután folytatjuk az illesztést a maradék sorozatokra (1) szerint.
(2) Ha mindkét term állandó, akkor attól függően, hogy különbözők vagy azonosak, az illesztés meghiúsul, vagy változóhelyettesítés nélkül sikeres lesz.
(3) Ha egyik term állandó, a másik összetett, az illesztés meghiúsul.
(4) Ha mindkettő összetett term, akkor azonos funktornév és aritás esetén paramétereik sorozatait (1) szerint illesztjük. Különböző funtornév vagy aritás esetén az illesztés meghiúsul.
(5) Ha mindkét term változó, bármelyiket helyettesíthetjük a másikkal. (Hatékonysági okokból általában az állításfej változóját helyettesítjük a célsorozat változójával.)
(6) Ha csak az egyik term változó, akkor ezt helyettesítjük a másik (*állandó* vagy *összetett*) termmel.

A fenti algoritmus változóhelyettesítései a célredukciós lépésben részt vevő célsorozatban és állításban a helyettesített változó minden előfordulására végrehajtódnak.

A célsorozat helyettesített változóit minden behelyettesítésnél feljegyezzük. Ha az illesztés végül meghiúsul, a célsorozat változóinak behelyettesítéseit e feljegyzés alapján megszüntetjük. Ha az illesztés végül sikeres lesz, a keresési fa éppen nyilvántartott ágának megfelelő élét e feljegyzés alapján felcímkézzük a célsorozat behelyettesített változóinak halmazával (6.1).

A fenti algoritmus (1)-es pontjában a termsorozatok megfelelő termpárjainak illesztése más sorrendben, vagy akár korutinszerűen is végrehajtható.

Vegyük észre, hogy az illesztés fenti algoritmusának (6)-os pontjában változó és összetett term illesztése esetén az eredeti egyesítő algoritmus szerint csak akkor hajthatnánk végre a változóhelyettesítést, ha a változó a struktúrában nem fordul elő. Ez a *változóelőfordulás-ellenőrzés*. Ez az eredeti egyesítő algoritmus szerves része [Fek 90, Kow 79, Der 96, S-S 94]. Tegyük fel például, hogy adott az `eq(X,X).` tény, és feltesszük az `eq(Y,f(Y))` kérdést. Ez a klasszikus célredukció szabályai szerint meghiúsul, a Prolog változat szerint azonban az `{X<-f(Y),Y<-f(Y)}` helyettesítést állítja elő. Ez az eredeti definíció szerint nem egyesítő helyettesítés. A Prolog szabvány szerint meghatározatlan az eredmény, ha az illesztő algoritmus során egy változó az őt tartalmazó struktúrával kerül szembe [Der 96]. A mi illesztő algoritmusunk speciálisabb a szabványosnál [Der 96], amely ebben az esetben akár végtelen ciklusba is kerülhet. Kikerüljük ugyanis az illesztő helyettesítés előállítását. Így ugyan jobban megkötjük a Prolog fordító megvalósítójának kezét, de egyrészt elkerüljük a végtelen ciklusokat, másrészt Colmerauer [Col 82] szellemében azt mondhatjuk, hogy egy `{Y<-f(Y)}` helyettesítés egy ciklikus `Y=f(Y)` termet hoz létre. A SICStus Prolog például eszközöket is ad az ilyen típusú termek kezelésére [Spl 02].

A megközelítés hátránya, hogy a ciklikus termekkel dolgozó programok általában nem jól hordozhatók a különböző Prolog változatok között. Másrészről viszont vegyük észre, hogy a *változóelőfordulás-ellenőrzés* elhagyása egyike azoknak a zseniális kompromisszumoknak, amelyek a Prologot gyakorlatilag használható programozási nyelvvé teszi. Enélkül ugyanis egy változó és egy összetett

term egyesítésének műveletigénye a struktúra méretével lenne arányos, míg így – a Prolog termeket láncoltan ábrázolva – az illesztés az összetett termtől független, állandó költséggel, valójában egyetlen pointerállítással megoldható.

6.3. NSTO programok

Ezeknél nincs szükség változóelőfordulás-ellenőrzésre. A továbbiakban tehát megvizsgáljuk, hogyan írhatunk olyan programokat, amelyekben a predikátumhívások során az eredeti egyesítő algoritmus sem végezne sikeres *változóelőfordulás-ellenőrzést*. Az ilyen predikátumhívásokat *NSTO*[11] céloknak nevezzük, különben *STO* célokról beszélünk. Ha egy programban minden cél NSTO, akkor maga a program is NSTO. Különben a program STO.

Ezzel kapcsolatosan egy elégséges feltételt fogalmazunk meg. (További, bonyolultabb feltételek találhatók még [D-M 93]-ban.)

Ha egy program minden állítása eleget tesz az alábbi feltételek *valamelyikének*, és a beépített eljárásokra vonatkozó hívások is NSTO célok, akkor a program maga is NSTO.

1. Az állításfejben és a vele kapcsolatos kérdésekben is paraméterként csak egyszerű termek (állandók és változók) használatosak.

2. Az állításfej nem tartalmaz kettőzött változót, azaz olyan változót, amelynek egynél több előfordulása lenne.

3. Az állításra vonatkozó célok nem tartalmaznak kettőzött változót.[12]

A fenti feltételek felhívják figyelmünket a programban lehetséges kritikus pontokra. Ezeken a pontokon az alábbi példa szerint járhatunk el. Tegyük fel, hogy a `member_/2` predikátumot a szokásos módon írtuk le:

```
% member_(X,Xs) :- X eleme az Xs listának (NSTO célokhoz).
member_(X,[X|_Xs]).
member_(X,[_X|Xs]) :- member_(X,Xs).
```

Látható, hogy ennek első állításával lehet probléma az NSTO tulajdonság szempontjából, hiszen kettőzött változót tartalmaz. Gyakorlati problémák esetén ilyenkor majdnem mindig teljesül az NSTO tulajdonság harmadik elégséges feltétele, esetünkben az, hogy a `member_(Y,Ys)` célban az `Y` és `Ys` együtt sem tartalmaz kettőzött változót.

Ha a fenti feltételek alapján sem lehetünk biztosak abban, hogy az NSTO tulajdonság teljesül, és más, saját ötletünk sincs ennek bizonyítására (15.5. feladat), vagy ha éppen az bizonyítható, hogy egy cél STO, célszerű a kérdésben a fenti `member_/2` predikátum helyett az alábbi `eleme/2` predikátumot meghívni:

[11] not subject to occurs check

[12] Eddigi programjainkban a programot elindító kérdést minden esetben a Prolog fejlesztői környezetéből, interaktívan tettük föl. Ez alapján úgy tűnhet, mintha most a programok használatát önkényesen korlátozni akarnánk. A gyakorlati programozásban azonban csak a tesztelési szakaszban használjuk így programjainkat, ily módon a procedurális nyelvekben szokásos tesztágyak megírását elkerülve. A végső program – mint a fejlesztői környezettől független alkalmazás – tartalmazza az indító direktívát (2) is, környezetével pedig a szokásos felületi eszközök segítségével tartja a kapcsolatot. Így az állításokra vonatkozó lehetséges hívások behatárolhatók.

```
% eleme(X,Xs) :- X eleme az Xs listának (STO változat).
eleme(X,[Z|_Xs]) :- unify_with_occurs_check(X,Z).
eleme(X,[_X|Xs]) :- eleme(X,Xs).
```

Itt a `unify_with_occurs_check(X,Z)` cél a Prolog megfelelő, szabványos beépített eljárását hívja meg, amely az `X` és `Z` Prolog termeket változóelőfordulás-ellenőrzéssel egyesíti, feltéve, hogy van egyesítő helyettesítésük, különben meghiúsul:

```
| ?- member_(X,[f(Y,Y),f(X)]).
X = f(Y,Y) ? ;
X = f(f(f(f(f(f(f(f(f(f(...)))))))))) ? ;
no
| ?- eleme(X,[f(Y,Y),f(X)]).
X = f(Y,Y) ? ;
no
| ?- member_(f(X,X),[f(Y,g(Z)),f(Y,g(Y))]).
X = g(Z),   Y = g(Z) ? ;
X = g(g(g(g(g(g(g(g(g(g(...)))))))))),
Y = g(g(g(g(g(g(g(g(g(g(...)))))))))) ? ;
no
| ?- eleme(f(X,X),[f(Y,g(Z)),f(Y,g(Y))]).
X = g(Z),   Y = g(Z) ? ;
no
```

Általában, ha egy programban használunk összetett termeket, akkor azokra az állításokra, amelyek fejében kettőzött változók szerepelnek, és a rájuk vontakozó Prolog célok NSTO tulajdonságát nem tudjuk bizonyítani, a következő átalakítást végezzük el: míg az állításfejben az NSTO tulajdonság szempontjából kritikus kettőzött változót találunk (jelölje most `X`), egyik előfordulását nevezzük át tetszőleges, az adott állításban még nem használt változónévre (jelölje most `Z`), majd az állítástörzs elejére szúrjunk be egy új `unify_with_occurs_check(X,Z)` hívást.

Ilyen módon programunk minden predikátumhívása NSTO lesz, kivéve a fenti átalakítással előállított `unify_with_occurs_check(X,Z)` hívásokat. Programunk így az elsőrendű logika szabályainak megfelelően működik, és hordozhatósága sem sérül. A Prolog beépített eljárásai esetében a hívások NSTO tulajdonsága általában automatikusan teljesül. Az ebben a munkában tárgyalt beépített eljárások közül kivételek az `(=)/2` és `(\=)/2` (7,7), `arg/3` (8.3), `read/2` (10.2), `retract/1` és `retractall/1` (10.3), `findall/3` (11), valamint a `catch/3` (12) predikátumok. Az ezekre vonatkozó hívások STO/NSTO tulajdonsága a fentiekhez hasonlóan ellenőrizhető és kezelhető.

Az itt ismertetett módszer lényege, hogy a klasszikus – és költséges – változóelőfordulás-ellenőrzéses egyesítést a programoknak csak azon – elvétve előforduló – pontjain alkalmazzuk, ahol az valóban szükséges.

6.4. Első argumentum indexelés

Az első argumentum indexelés – ha programunk kódolásánál figyelembe vesszük – nagyban növelheti a program hatékonyságát.

A választási pontok létrehozása és kezelése ugyanis viszonylag költséges. Predikátumhívásaink azonban gyakran *determinisztikusak*, ami alatt a Prologban azt értjük, hogy pontosan egy állításfej illeszkedik rájuk. (Egy hívás *indeterminisztikus*, ha több állításfejjel is illeszthető. Ilyenkor ugyanis a program futása elvileg többféleképpen is folytatódhatna, bár a Prolog-fordító az indeterminisztikusságot az ismert visszalépéses vezérléssel oldja fel.) Ha a Prolog gép felismeri egy predikátumhívás determinisztikusságát, nem hoz létre választási pontot. Az első argumentum indexelés pedig éppen a determinisztikusság felismerését támogatja. Lássuk, hogyan:

A (6.1) pontban ismertetett programvégrehajtó algoritmus (3). lépésében a keresési fa aktuális csúcsába felvettük az aktuális célsorozat első részcéljához tartozó predikátum összes állításának listáját. Ezzel – feltéve, hogy a lista egynél több elemet tartalmaz – egy választási pontot hoztunk létre, ahol tehát a keresési fa elágazik a predikátumhíváshoz illeszkedő állításfejek szerint.

A fenti listára nyilván elég lett volna azokat az állításokat felvenni, amelyek feje illeszthető a kérdéses részcéllal. Ezeket előre meghatározni azonban túl sok spekulatív munkával[13] jár, ezért a legtöbb Prolog megvalósítás egy kompromisszumos megoldást alkalmaz, amit *első argumentum indexelésnek* nevezünk. Ennek lényege:
(1) Abban az esetben, ha a hívás első paramétere állandó, csak azok az állítások kerülnek a listára, amelyek fejének első formális paramétere ugyanaz az állandó, vagy változó.
(2) Ha a hívás első paramétere összetett term, akkor csak azok az állítások kerülnek a listára, amelyek fejének első formális paramétere változó, vagy ezen formális paraméter fő funktora megegyezik a hívás első paraméterének fő funktorával.
(3) Ha a hívás első paramétere változó, akkor a hivatkozott predikátum minden állítása a listára kerül.

A lehetséges listákat a Prolog-fordító rendesen már fordítási időben létrehozza, és egy táblázatot készít belőlük, ahonnan a megfelelő lista kiválasztása futás közben már elhanyagolható műveletigénnyel lehetséges.[14]

A fentiek szerint a `list/1, append_/3, rev_app/3` predikátumok (5) futása során – ha a megfelelő célok első paramétereit valódi listákként adjuk meg – minden eljáráshívás determiminisztikus lesz, és egyetlen (rekurzív) hívás sem hoz létre választási pontot. Ha ugyanis a cél első paramétere üres lista, mindegyik esetben az első, míg ha nem üres lista, a második állításra indexelünk. A `member_/2` predikátumot az első argumentum indexelés nem érinti, mivel mindkét állítása fejrészének első paramétere változó.

6.5. Utolsó hívás optimalizáció

Ha a (6.1) részben ismertetett Prolog gép működésének (5)-ös pontjában az aktuális állításlista egyelemű volt, akkor első elemének törlése után üres állításlista maradt, ami azt jelenti, hogy egy későbbi esetleges visszalépés ezen a

[13] Spekulatív munka: amit nagy valószínűséggel részben megtakaríthatnánk.

[14] A változó-átnevezések ugyanúgy oldhatók meg, mint a procedurális nyelveknél. Másrészt, a többszörös argumentum-indexelés éppen a táblázat méretének gyors növekedése miatt nem szokásos.

csúcson a (4)-es pont szerint újabb visszalépést eredményez. Ezért a Prolog gép algoritmusának (8)-as pontjában ilyenkor megtehetjük, hogy a keresőfa éppen ekkor nyilvántartott ágának új csúccsal való bővítése helyett az aktuális csúcsot írjuk felül, az ide vezető élet címkéző változóhalmazhoz pedig hozzávesszük azokat a változókat is, amelyek az eredeti változatban az új csúcshoz vezető élet címkéznék. Ilyenkor tehát a keresőfa aktuális ága nem lesz hosszabb.

Ha tehát egy predikátumhívást megvalósító állítástörzs utolsó részcéljának meghívása pillanatában a tartalmazó predikátum meghívása óta nincs élő választási pont, akkor ez az utolsó hívás újrahasznosíthatja a szülője (a tartalmazó predikátum) meghívása által igényelt memóriát. Ezt *utolsó hívás optimalizációnak* nevezzük. Ha itt a szülő predikátum önmagát hívja, akkor ez a végrekurzió lényegében véve ciklusként működhet, azaz *végrekurzió-optimalizáció* alkalmazható.

Mivel a `list/1, member_/2, append_/3` és `rev_app/3` predikátumok (5) futása során a rekurzív hívás pillanatában nincs élő választási pont, mindegyik esetben végrekurzió-optimalizációt végezhetünk. Amikor pedig a `reverse/2` predikátum hívja meg a `rev_app/3` eljárást, utolsó hívás optimalizáció alkalmazható.

7. Vezérlésmódosítás a Prologban

Az eddig tárgyalt tiszta Prolog nyelvben nem tudjuk megfogalmazni a tagadást. Azt például, hogy két term illeszthető (`A=B`), könnyen leírhatjuk, például az `X=X.` tény segítségével. Ennek tagadását azonban (`A\=B`) még nem tudjuk megfogalmazni.[15] Hasonlóképpen, a `member_/2` predikátumot definiáltuk a (5.1) részben, de a `nonmember_/2` predikátumhoz eddigi eszközeink elégtelennek bizonyulnak. Ebből már következik, hogy például két rendezetlen lista unióját vagy metszetét sem tudjuk megfogalmazni, hiszen itt meg kéne mondani, mi legyen, ha az egyik lista egy eleme a másik listán nem található.

Állításaink – tényeink és szabályaink – általában nem alkalmasak arra, hogy negatív információt következtessünk ki programunkból. Legfeljebb azt tudjuk eldönteni, hogy egy állítás következik-e a programból. (Például ha az állítás – mint cél – keresési tere véges.) Ha viszont egy ilyen eldönthető kérdésről feltehető, hogy bizonyíthatósága egybeesik igazságával, bizonyíthatatlanságából egyben negáltjára következtethetünk. (Ilyenek például a (5) listakezelő predikátumaira vonatkozó kérdések.) Ha tehát rendelkezni tudnánk arról, hogy mi történjen, ha egy célsorozat sikeres (illetve ha meghiúsul), valamelyest korlátozott, de gyakorlati eszközünk lenne a tagadás kezelésére.

7.1. Feltételes célok

A strukturált programozási stílus szempontjából talán a legfontosabb vezérlésmódosító eszközök a *feltételes célok*. Alapformájuk:

`(` *ha* `->` *akkor*
`;` *különben*

[15] Az `(=)/2` és a `(\=)/2` valójában a szabványos Prolog nyelv [Der 96] beépített predikátumai, ezért nem definiálhatjuk őket felül. A Prolog célokban mindkettő neve közbevetett (infix) módon írható (9).

)

Ahol a *ha*, az *akkor* és a *különben* is tetszőleges Prolog célsorozatok lehetnek. Amennyiben a *ha* célsorozat sikeres, a feltételes cél megoldásait az *akkor* célsorozat megoldásai adják. Egyébként a feltételes cél megoldásait a *különben* célsorozat megoldásai szolgáltatják.

A fenti megfogalmazásból következik, hogy a *ha* célsorozatba sikeres végrehajtása esetén sem lehetséges a visszalépés. Ilyenkor ugyanis a *ha* célsorozat esetleges választási pontjait a Prolog rendszer közvetlenül annak első sikeres végrehajtása után levágja. A *különben* ág helyére – külön zárójelpár nélkül – feltételes cél is írható, hasonlóan az Algol, Pascal, vagy a C nyelvek többirányú elágazásaihoz. A

`(` *ha* `->` *akkor* `)`

feltételes cél is megengedett. Mivel ennek nincs *különben* ága, ott megoldások sincsenek. Így ez a fajta feltételes cél egyenértékű a

`(` *ha* `->` *akkor* `; fail )`

szerkezettel, ahol a `fail/0` beépített eljárás, amelynek hívása mindig meghiúsul. Ha azt akarjuk, hogy a *ha* meghiúsulása esetén a feltételes cél sikeres legyen, akkor a

`(` *ha* `->` *akkor* `; true )`

szerkezetet használjuk, ahol a `true/0` beépített eljárás, amelynek hívása mindig sikeres.

A jelen alfejezet (7) bevezetőjében említett problémák néhány lehetséges megoldása ezután a következő.

```
% X\=Y :- X nem illeszthető Y-nal.
X\=Y :- ( X=Y -> fail ; true ).

% nincs_benne(Xs,Y) :- Az Xs listán nem található Y.
nincs_benne([],_Y).
nincs_benne([X|Xs],Y) :-
    ( X = Y -> fail
    ; nincs_benne(Xs,Y)
    ).

% nonmember_(X,Xs) :- nincs_benne(Xs,X).
nonmember_(X,Xs) :- ( member_(X,Xs) -> fail ; true ).

% metszet(Xs,Ys,Zs) :-
%     Az Xs lista Ys listán nem szereplő
%     elemei elhagyásával adódik a Zs lista.
metszet([],_Ys,[]).
metszet([X|Xs],Ys,Zs) :-
    ( member_(X,Ys) ->
          Zs = [X|Ms], metszet(Xs,Ys,Ms)
    ; metszet(Xs,Ys,Zs)
    ).

% unió(Xs,Ys,Zs) :-
%      Az Xs lista Ys listán nem szereplő elemeit
%      sorban Ys elé kapcsolva kapjuk a Zs listát.
```

```
unió([],Ys,Ys).
unió([X|Xs],Ys,Zs) :-
    ( member_(X,Ys) -> unió(Xs,Ys,Zs)
    ; Zs = [X|Us], unió(Xs,Ys,Us)
    ).
```

Vegyük észre, hogy a `nincs_benne/2`, a `metszet/3` és az `unió/3` rekurzív predikátumok argumentumsorrendjét az első argumentum indexelés kihasználása vezérelte (6.4). Mindegyik rekurzív hívásra végrekurzió-optimalizáció végezhető (6.5). Az ezeket néhol megelőző `Zs=[X|Vs]` alakú célok, az egyenlőség paramétereinek *illesztését* célozzák.

A szabványos Prolog nyelv ugyanis *nem* tartalmaz értékadó utasítást, hiszen a logikai változók az állítások *ismeretlenjei* (4), [Der 96, Kee 90, S-S 94], ami azt jelenti, hogy a program írásakor ismeretlen, a program által kiszámítandó objektumot jelölnek. Ha tehát egyszer behelyettesítjük a logikai változót, azaz meghatározzuk a programban kódolt relációk által definált, de eddig ismeretlen értékét, nincs értelme azt felülírni. Tipikus kezdő Prolog programozói hiba az `Xs=[Y|Xs]` cél és hasonlók használata. Ez a Prolog cél biztosan nem fogja végrehajtani a programozó által óhajtott értékadást. Ha például az `Xs` alapterm, az `Xs` és az `[Y|Xs]` illesztése nyilván meghiúsul. Ha az `Xs` behelyettesítetlen változó, akkor a cél STO, ami a Prologban általában nem kívánatos stb.

A `member_(X,Xs)` cél (5.1) futtatása – visszalépések során – e kérdés minden lehetséges megoldását felsorolja. Első megoldása után általában választási pontot hagy maga után. Ha azonban csak ellenőrzésre szeretnénk használni, ez gyakran zavaró lehet. Befejezésül két különböző megoldást adunk erre a problémára. Az első azt példázza, hogy a feltételes szerkezet (azaz feltételes cél) a felesleges választási pontoktól is megszabadít bennünket, ahogy fenti példáinkban is. A második azt mutatja, hogy a meghiúsulást nem kell expliciten programoznunk. Ha ez zavar bennünket – mint ez hagyományos programozási nyelveken nevelkedett programozókkal gyakran megesik – hatékonysági okokból jobb, ha a felesleges állításokat az alábbi példának megfelelően megjegyzésbe tesszük.

```
member1(X,Xs) :- ( member_(X,Xs) -> true ).

% member_check(_X,[]) :- fail.
member_check(X,[Y|Ys]) :-
    ( X = Y -> true
    ; member_check(X,Ys)
    ).
```

7.2. Tagadás

Tegyük fel, hogy szeretnénk definiálni a `'nőtlen_hallgató'/1` Prolog predikátumot, feltéve, hogy adottak a `hallgató/1` és a `'nős'/1` predikátumok. Az előző részben mondottak alapján nyilvánvalóan adódik az alábbi megoldás.

```
'nőtlen_hallgató'(X) :- hallgató(X), 'nőtlen'(X).

'nőtlen'(X) :- ( 'nős'(X) -> fail ; true ).
```

```
hallgató('Péter').   hallgató('János').
hallgató('Jakab').
'nős'('Péter').        'nős'('József').
```

Megfigyelhetjük, hogy a `(\=)/2` a `nonmember_/2` és a `'nőtlen'/1` közös sémája a következő:

```
nem(P) :- ( P -> fail ; true ).
```

Kérdés, hogy a tagadásnak ez az általánosabb formája értelmezhető-e Prolog predikátumként.

Világos, hogy a Prolog célok formailag összetett termek vagy esetleg nevek. Ez lehetővé teszi, hogy egy Prolog célt futási időben határozzunk meg – hisz ez a kiszámító részprogram számára közönséges Prolog term –, majd mint *metacélt* hívjuk meg. A metacél a forráskódban általában változóként, de egy Prolog célsorozat elemeként jelenik meg. Az `X=p(Y),X` célsorozat például jelentésében egyenértékű a `p(Y)` céllal. A Prolog szabályok törzsében megjelenő metacélok egyben a szabályfej formális paraméterei is lehetnek, mint a fenti `nem(P)` fejű szabály esetén is. Ha egy szabály törzsében megjelenő metacél egyben a szabályfej formális paramétere, a szabályt tartalmazó predikátumot *metapredikátumnak* nevezzük, mint például a `nem/1` predikátumot. A megfelelő paraméterhelyet – mint a `nem/1` predikátum egyetlen argumentumát – *meta-argumentumnak* nevezzük. A meta-argumentumokat a metapredikátum hívásakor célokkal paraméterezhetjük.

A `'nőtlen_hallgató'/1` predikátumot így is megadhatnánk:

```
'nőtlen_hallgató'(X) :- hallgató(X), nem('nős'(X)).
```

A `nem/1` metapredikátum egyetlen formális paramétere `P`, amely a szabálytörzsben metacélként hívódik meg: ha `P` sikeres, esetleges választási pontjait a feltételes szerkezet levágja, majd a feltételes cél *akkor* ágán a `fail` cél hajtódik végre és a `nem(P)` hívás meghiúsul. Ha `P` meghiúsul, a feltételes cél *különben* ágán a `true` cél hajtódik végre és a `nem(P)` hívás sikeres lesz. Összegezve, `nem(P)` pontosan akkor sikeres, ha a `P` cél meghiúsul (és így `P`-nek egyetlen megoldása sem adódik).

Ez a *negáció, mint meghiúsulás* elve. Ez *nem* logikai tagadás, de viszonylag hatékonyan valósítható meg[16], és némi programozói önfegyelemmel alkalmazva általában kiváltja azt.
Figyeljük meg először a `nem(P)` cél négy tulajdonságát:
(I) Sosem hagy maga után választási pontot.
(II) Függetlenül a `P` céltól, sosem példányosítja annak változóit.
(III) `nem(P)` pontosan akkor sikeres, ha `P` keresési fája véges, és nem tartalmaz megoldást.
(IV) `nem(P)` pontosan akkor hiúsul meg, ha `P` keresési fája tartalmaz megoldást, de az első megoldás ág előtt nincs végtelen ág.
(Vegyük észre, hogy ezek a megfigyelések akkor is érvényesek, ha a tagadás implicit, azaz a konkrét esetben feltételes céllal programozzuk, mint fentebb a `(\=)/2`, a `nonmember_/2` és a `'nőtlen'/1` predikátumoknál.)

[16]Meg kell jegyeznünk, hogy bár a `nem('nős'(X))` és a `'nőtlen'(X)` ugyanazt jelentik, a `nem('nős'(X))` a mai Prolog rendszerekben általában kevésbé hatékony, mert a metacélok legtöbbször nem optimalizálhatók a program fordítása során.

(II)-ből világos, hogy egy negált Prolog cél sosem ad megoldás(oka)t, azaz nem lehet (rész)eredmények kiszámítására, hanem csak szűrésére, azaz ellenőrzésére használni. Ezzel összhangban, nem mindegy, hogy egy negált cél hol helyezkedik el a célsorozatban, hiszen az elvárt működéshez általában szükséges, hogy változói példányosítva legyenek. Például

```
| ?- 'nőtlen_hallgató'(X).
X = 'János' ? ;    X = 'Jakab' ? ;    no
```

függetlenül attól, hogy melyik változatot használjuk. Ha azonban a szabálytörzs két részcélját fölcseréljük (megint mindegy, hogy melyik változatban), akkor a

```
| ?- 'nőtlen_hallgató'(X).
no
```

eredményt kapjuk. Ilyenkor ugyanis a `'nőtlen'(X)`, illetve a `nem('nős'(X))` cél meghiúsul, hiszen a `'nős'(X)` – lévén `X` még behelyettesítetlen változó – sikeres lesz.

Általában a Prolog tagadás logikai értelemben vett helyességének *elégséges* feltétele, hogy a negált cél a meghívás pillanatában ne tartalmazzon változót, és keresési fája véges legyen.[17]

A `nem/1` predikátumban fent leírt tagadást a Prologban a `(\+)/1` beépített eljárás hatékonyabban valósítja meg. Ez előtag műveleti jelként is használható, mint a `nőtlen_hallgató/1` predikátum alábbi verziójában (lásd még (9)):

```
'nőtlen_hallgató'(X) :- hallgató(X), \+ 'nős'(X).
```

8. A Prolog metalogikai predikátumai

A gyakorlati programozásban gyakran van szükségünk olyan információkra, számításokra, amelyek eddigi eszközeinkkel nem, vagy nem elég hatékonyan fejezhetők ki. Ilyenek az aritmetikai számítások, ismeretlen adatok típusának meghatározása, összehasonlításuk, ismeretlen szerkezetű termek kezelése stb.

8.1. Aritmetika

A Prologban a szokásos aritmetikai műveleti jelek, mint a `+,-,*,/,**` stb. csupán közbevetett (infix) illetve előtag (prefix) műveleti *jelként* írható függvényszimbólumok. Ennek az az oka, hogy a nyelv támogatja a szimbolikus számításokat, mint például a `-x^2+3*x-4` polinom deriválása vagy integrálása. Ebből a szemléletből fakad, hogy például a `2+3` kifejezés itt csak a `+(2,3)` termet jelenti, és a Prolog rendszer általában nem, kizárólag csak *aritmetikai argumentumokban* értékeli ki. Ezek
– az `is/2` beépített eljárás jobb oldali argumentuma, és
– az `(=:=)/2, (=\=)/2, (<)/2, (>)/2, (=<)/2, (>=)/2` aritmetikai összehasonlító predikátumok argumentumai.

A fenti eljárások nevei a hívásokban közbevetett (infix) jelöléssel írhatók. A `Term is Exp` cél tehát először kiértékeli az `Exp` aritmetikai kifejezést, majd

[17] A tagadással kapcsolatban a *zárt világ feltételezéssel* élünk. Ez röviden azt jelenti, hogy feltételezésünk szerint elegendő információ áll rendelkezésünkre, azaz az aktuális modellben minden állítás, ami nem következik a programunkból, hamis.

az eredményt megpróbálja *illeszteni* a `Term` Prolog termmel. *Aritmetikai kifejezés* egy egész (`integer`) vagy lebegőpontos (`float`) szám, illetve egy összetett (`compound`) term, amelynek funktora aritmetikai függvényszimbólum (ezek a szokásos aritmetikai műveleti jelek, logaritmikus, trigonometrikus stb. függvényjelek [Spl 02, Der 96]), paraméterei pedig aritmetikai kifejezések. (Ez azt jelenti, a Prolog célok *aritmetikai argumentumait* paraméterező kifejezések változóit a célok meghívásáig megfelelően be kell helyettesíteni.) Például (`//` az egészosztás, `floor` az egészrész):

```
| ?- X is -2**3, Y is floor(cos(0))+3*(8-7//2).
X = -8.0,    Y = 16
| ?- X = -2**3, Y = floor(cos(0))+3*(8-7//2).
X = -2**3,    Y = floor(cos(0))+3*(8-7//2)
| ?- 5 is 2+3, 5.0 =:= 2+3.
yes
| ?- 2+3 is 2+3.
no
| ?- 5.0 is 2+3.
no
```

Tipikus kezdő Prolog programozói hiba az `X is X+1` cél és hasonlók használata. Ez a Prolog cél biztosan nem fogja végrehajtani a programozó által óhajtott értékadást. Ha például a fenti utasítás meghívásakor az `X` értéke egy szám, az `X+1` értékének és az `X` értékének illesztése meghiúsul. Ha pedig az `X` behelyettesítetlen változó, akkor az `X+1` kiértékelése az `instantiation_error` kivételt váltja ki (12).

Fel kell még hívnunk a figyelmet arra, hogy az aritmetikai összehasonlító predikátumok csak valódi aritmetikai kifejezések számértékeit hajlandók összehasonlítani. Például az `Y=:=X+1` utasítás *nem* használható az `Y is X+1` helyett az `X+1` értékének kiszámítására:

```
| ?- Y is 6+1.
Y = 7
| ?- Y =:= 6+1.
{INSTANTIATION ERROR: _157=:=6+1 - arg 1}
```

A most következő két predikátum egyenlő hosszúságú valódi számlistákkal ábrázolt vektorok összegét és skaláris szorzatát számolja ki.

```
% add(V1,V2,V) :-
%     A V vektor a V1 és V2 vektorok összege.
add([],[],[]).
add([X|Xs],[Y|Ys],[Z|Zs]) :-
    Z is X+Y, add(Xs,Ys,Zs).

% mult(V1,V2,S) :-
%     S a V1 és V2 vektorok skaláris szorzata.
mult([],[],0).
mult([X|Xs],[Y|Ys],S) :-
    mult(Xs,Ys,S0), S is S0+X*Y.
```

A fenti predikátumok determinisztikusak az első argumentum indexelés szerint. `add/2` -re így végrekurzió-optimalizáció is alkalmazható. `mult/2` is átalakítható

ilyenné, ha bevezetünk egy akkumulátort, amiben a részösszegeket gyűjtjük, vagyis általánosítjuk a feladatot úgy, hogy a két vektor skaláris szorzatát egy kezdőértékhez kell hozzáadni:

```
mult(V1,V2,S) :- mult(V1,V2,0,S).

% mult(V1,V2,A,S) :- S = A+V1*V2.
mult([],[],A,A).
mult([X|Xs],[Y|Ys],A0,S) :-
     A1 is A0+X*Y, mult(Xs,Ys,A1,S).
```

8.2. Termek típusa és összehasonlítása

Egy Prolog term (ld. 4) lehet *változó* (`var`), *állandó* (`atomic`), vagy *összetett*, azaz *struktúra* (`compound`). Az állandók és a struktúrák együttes típusneve `nonvar`.

Egy *állandó* lehet *név* (`atom`), vagy *szám* (`number`), ami lehet *egész* (`integer`) vagy *lebegőpontos* (`float`).

Zárójelben megadtuk a típusok eredeti Prolog-beli neveit is, mert ezek egyben a megfelelő (egy aritású) típusellenőrző Prolog predikátumok nevei is. Például a `'nagyszülője'/2` predikátumot (2.2) hatékonyabbá tehetjük a következő módon (az Olvasóra bízzuk a változtatás magyarázatát):

```
'nagyszülője'(X,Y) :-
    ( var(X), nonvar(Y) -> 'szülője'(Z,Y), 'szülője'(X,Z)
    ; 'szülője'(X,Z), 'szülője'(Z,Y)
    ).
```

A Prologban két tetszőleges term összehasonlítható. Két term azonosságát vizsgálják az `(==)/2` és a `(\==)/2` beépített predikátumok. Eszerint az

```
X \== Y, X=Y, X==Y.
```

célsorozat sikeres, hiszen a két változó eredetileg különbözik, de az illesztésük során azonosakká válnak.

A **termek szabványos rendezésében** a *változók* **megelőzik** a *lebegőpontos számokat*, ezek az *egészeket*, ezek a *neveket*, ezek pedig az *összetett termeket*. Az `A @< B` cél pontosan akkor sikeres, ha a termek szabványos rendezése szerint `A` megelőzi `B`-t, azaz `A` kisebb mint `B`.

A valós számok egymás között számértékük szerint rendezettek, és az egész számokról is ezt mondhatjuk, de a valósak kisebbek, mint az egészek:

```
| ?- 5 < 5.1, 5.1 @< 5, 9.9e99 @< -999999999.
yes
```

A *nevek* az őket alkotó karakterláncok szerint ábécésorrendbe rendezettek, ahol az egyes karaktereket ASCII kódjaik szerint hasonlítjuk össze. A *struktúrák* elsősorban aritásuk, másodsorban nevük szerint rendezettek; végül paramétereiket kell sorban összehasonlítani a lexikografikus rendezettség szabályai szerint, a paraméterpároknál újra a szabványos term rendezést alkalmazva.

A változók rendezése minden esetben meghatározott, de rendszerfüggő. A változók és a változót tartalmazó struktúrák rendezettsége a változóhelyettesítések során változhat.

A szabványos rendezettség szerint termjeinket a `(@>)/2`, `(@<)/2`, `(@>=)/2`, `(@=<)/2`, beépített predikátumok segítségével hasonlítjuk össze. Például:

```
% rendbeszúr(Ys,X,Zs) :-
%     Ys (szabványos term rendezés szerint)
%     monoton növekvő valódi Prolog listába
%     X rendezett beszúrásával adódik Zs.
rendbeszúr([],X,[X]).
rendbeszúr([Y|Ys],X,Zs) :-
    ( X @=< Y -> Zs = [X,Y|Ys]
    ; Zs = [Y|Us], rendbeszúr(Ys,X,Us)
    ).
```

8.3. Termműveletek

Az eddigi példákban feltettük, hogy a programok által feldolgozandó termek szerkezetét jól ismerjük. Ez azonban a gyakorlatban nincs mindig így (például ha egy fájlból jövő bemenetet kell feldolgozni). Ilyenkor a termeket a termműveletek segítségével szedhetjük részekre, majd esetleg újra összerakjuk őket.

A struktúrák kezeléséhez a legfontosabbak a `functor/3` és az `arg/3` beépített eljárások.

A `functor(Term,Funktornév,Aritás)` hívás segítségével meghatározhatjuk vagy ellenőrizhetjük a *nonvar* `Term` funktornevét és/vagy aritását. (Az állandók aritása nulla.) Ha a híváskor a `Term` változó, a `Funktornév` név, az `Aritás` pedig pozitív egész, a `Term`-ben létrejön a megfelelő összetett term, paramétereiban új változókkal.

Az `arg(I,Struktúra,Arg)` cél illeszti `Arg`-gal a `Struktúra` összetett term `I`-edik paraméterét, ahol `1 =< I =< Aritás`.

Így tehát *ki is nyerhetjük* vagy *ki is tölthetjük* egy összetett term paramétereit (általában miután egy `functor(Term,Funktornév,Aritás)` hívás segítségével meghatároztuk a term aritását, vagy éppen létrehoztuk a struktúrát):

```
% cserél(T0,X,Y,T) :-
%     T a T0 másolata, kivéve, hogy X T0-beli
%     előfordulásainak T-ben Y felel meg.
% (Elégséges előfeltétel: T0 és Y alaptermek; vagy
%  T behelyettesítetlen változó és nem fordul elő sem T0-ban sem Y-ban.)
cserél(T0,X,Y,T) :-
    ( T0 == X -> T = Y
    ; compound(T0) ->
        functor(T0,F,N), functor(T,F,N),
        arg_csere(N,T0,X,Y,T)
    ; T = T0
    ).

arg_csere(N,T0,X,Y,T) :-
    ( N > 0 ->
        arg(N,T0,A), cserél(A,X,Y,B), arg(N,T,B),
        N1 is N-1, arg_csere(N1,T0,X,Y,T)
    ; true
```

```
    ).

% Teszt:
| ?- cserél(a(X,nil,b(nil,2,B,nil)),nil,[],T).
T = a(X,[],b([],2,B,[]))
```

Az állandók szétszedésére és összerakására szolgálnak az `atom_codes(Név,Karakterkódlista)` *(atom=név)* és a `number_codes(Szám,Karakterkódlista)` predikátumok. (További termkezelők [Der 96, Spl 02]-ben.) Ezek segítségével tehát bármilyen szövegkezelő műveletet listakezelésre vezethetünk vissza. Például:

```
% aA(A,B) :- B az A név másolata, kivéve, hogy
%            nagybetűssé tettük (á kódja 0'á stb.).
aA(A,B) :-
    atom_codes(A,Cs), cC(Cs,Ds), atom_codes(B,Ds).

cC([],[]).
cC([C|Cs],[D|Ds]) :- bB(C,D), cC(Cs,Ds).

bB(C,D) :-
    ( 0'a=<C, C=<0'z -> D is C-0'a+0'A
    ; éÉ(C,D) -> true
    ; D = C
    ).

éÉ(0'á,0'Á). éÉ(0'é,0'É). éÉ(0'í,0'Í).
éÉ(0'ó,0'Ő). éÉ(0'ö,0'Ö). éÉ(0'ő,0'Õ).
éÉ(0'ú,0'Ú). éÉ(0'ü,0'Ü). éÉ(0'ű,0'Ű).

% Teszt:
| ?- aA('ó míly gyönyörű ő',Ismeretlen).
Ismeretlen = 'Ő MÍLY GYÖNYÖRŰ Õ'
```

9. Prolog műveleti jelek

Említettük, hogy bizonyos (például az összehasonlító) predikátumok nevei közbevetett (infix) módon írhatók, és némely aritmetikai függvényszimbólumot is műveleti jelként jelölhetünk, noha mindezek – környezetfüggetlenül tekintve – csupán összetett termek függvényszimbólumai. Ez pedig a Prolog *műveleti jel fogalmán* alapul. Eszerint bizonyos *neveket* műveleti jelként definiálhatunk.

1200 prioritási szint van. Az első szinten lévő műveleti jelek kötnek a legerősebben, míg az 1200-adik szinten lévők a leggyengébben. Egy műveleti jel a Prologban *nem csinál semmit*, hanem – megfelelően használva – a program olvashatóságát javító *jelölési lehetőség*. Például `a+b*c == +(a,*(b,c))`, a 400 prioritású `*` műveleti jel ugyanis erősebben köt, mint az 500 prioritású `+`. Egy term prioritása nulla, ha változó vagy állandó, zárójelbe tettük, listaként vagy a szokásos függvényjelöléssel – $f(t_1, \ldots, t_n)$ formában – írtuk le. Egy zárójelbe nem tett term prioritása fő funktorának prioritásával egyezik meg, amennyiben azt műveleti jelként írtuk le.

Egy *műveleti jel* lehet *közbevetett* (infix), *előtag* (prefix), vagy *utótag* (posztfix), aszerint, hogy két paramétere közé, egyetlen paramétere elé vagy mögé írjuk. Ugyanaz a név legfeljebb két műveleti jelhez tartozhat. Ilyenkor az egyik közbevetett, a másik előtag vagy utótag. Egy közbevetett műveleti jel lehet `yfx` (balról jobbra zárójeleződő, például `+`, `-`), `xfy` (jobbról balra zárójeleződő, például `->`, `;`), vagy `xfx` (mindenképpen zárójelezendő, például `=`, `\=`) módú. Itt `f` a függvényszimbólumot, azaz a műveleti jelet, `x` és `y` a paramétereit, vagyis az operandusait jelöli; y = yes: a megengedő oldal, ahol tehát a műveleti jellel kisebb vagy egyenlő prioritású kifejezés is lehet; míg x = áthúzás: ennek az oldalnak a prioritása kisebb kell legyen. Például `a+b+c==(a+b)+c`, `a+b` prioritása ugyanis 500, ami a 2. `+` jel bal oldalán megengedett, de `b+c` prioritása is 500, ami az 1. `+` jel jobb oldalán nem megengedett. Az előtag műveleti jelek `fx` vagy `fy`, az utótag műveleti jelek `xf` vagy `yf` módúak lehetnek a fenti logika szerint.

A műveleti jeleket az `op(Prioritás,Mód,Név)` utasítással hozhatjuk létre, illetve definiálhatjuk felül a fentiek szerint. (A `Név` itt egyetlen név, vagy valódi névlista is lehet.) Ha a `Prioritás`-nak `0`-át adunk meg, ez az adott nevű és módú műveleti jel törlését eredményezi. Ezt az utasítást általában *direktívaként* (2) írjuk le, és így műveleti jeleinket a program betöltésekor hozza létre. Például:

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

A fentiekben a Prolog ISO-szabványos, előre adott műveleti jeleit vezettük be. Mivel a vessző közbevetett műveleti jel (ilyenkor aposztrófok nélkül írható) és paraméter- illetve listaelem-elválasztó is, ezért a paraméterlisták és a szabványos Prolog listák elemeinek prioritása kisebb kell legyen, mint `1000`, hogy a listaelem-elválasztó vesszők és a közbevetett műveleti jel vesszők össze ne keveredjenek egymással. (Ha tehát egy paraméter vagy listaelem prioritása elérné az ezret, zárójelbe kell tenni.)

Látható, hogy a mondatok felírásához szükséges jelek – mint a `:-` szimbólum és a vessző – is műveleti jelek. Műveleti jel továbbá a negációt jelölő `\+` és az `(If->Then;Else)` szerkezet összekötő jelei is. Környezetfüggetlenül tekintve tehát nem csak az atomi formulák paraméterei, hanem a Prolog célok, sőt, programjaink mondatai is termek. Ugyanaz a term adat, ha egy állításfej vagy egy predikátumhívás egy paramétere, cél, ha egy célsorozat eleme, és mondat, ha a program mondatainak sorában foglal helyet, szabályosan ponttal és helyközzel lezárva. A tagadást például – ha nem lenne bépített predikátum – a

```
\+ Cél :-
    ( Cél -> fail
    ; true
```

```
).
```

állításal definiálhatnánk. Ebben a `Cél` első előfordulása Prolog adatot, konkrétan formális paramétert, míg második előfordulása Prolog célt jelöl. (Vegyük észre, hogy a fenti esetben a feltételes szerkezetet nem lenne szükséges zárójelbe tenni, a program jó olvashatósága miatt mégis minden esetben ragaszkodunk a feltételes szerkezetek szokásos zárójelezéséhez és behúzásához.) A szabványos, funkcionális termjelöléssel élve a fenti állítást az alábbi formában is leírhatnánk (bár ezt legfeljebb a próba kedvéért javasoljuk):

```
:-(\+(Cél),;(->(Cél,fail),true)).
```

Az, hogy a Prologban – formai értelemben – nincs éles határvonal program és adat között, jelentősen megkönnyíti a tanuló programok, értelmező- és fordítóprogramok és általában a nyelvi feldolgozó programok készítését, főként ha a tárgy és a célnyelv is megfelel a Prolog termfogalmának [SzB 01]. (Ha ez valamelyik esetben nem így van, *logikai nyelvtanok* segítségével könnyíthetjük meg munkánkat [D-M 93, Spl 02, S-S 94, SzB 01].)

10. A Prolog logikán kívüli predikátumai

Ezek valójában *csak* műveleti jelentéssel bíró eljárások. Deklaratív olvasatuk nincs. Kívül esnek a logikai programozási modellen, a gyakorlati programozás szempontjából mégis a Prolog nyelv nélkülözhetetlen elemei.

Ide tartoznak a mellékhatásos predikátumok, úgymint a környezet kezelő (interface) predikátumok különböző fajtái és a programmódosító predikátumok, amelyek tanuló programok készítését, általában a program keresési fájának alternatív ágai között az információ átadását támogatják [Kee 90, S-S 94]. Az ezen predikátumok által okozott mellékhatás gyakran éppen az adott predikátumhívás lényege, és sosem léptethető vissza.

A környezetkezelő predikátumok fajtái általában a programbetöltő, adatbeolvasó és kiíró, hálózati kapcsolatokat és más nyelvű programmokkal az adatforgalmat kezelő predikátumok [Spl 02, SzB 01]. Ez utóbbiak a különböző megvalósításokban tipikusan a C/C++, Java, Basic, Pascal, Tcl/Tk stb. programnyelvekhez, valamilyen adatbáziskezelőhöz (ennek segítségével intelligens adatbáziskezelő felületet készíthetünk) valamilyen GUI (grafikus felhasználói felületi) eszközhöz (például Tcl/Tk), illetve néhány Prolog-kiterjesztés kiegészítő (például funkcionális, objektumorientált, constraint kezelő, logikai nyelvtan stb.) összetevőjéhez biztosítják a kapcsolatot, és egyéb, a programfejlesztést támogató eszközökkel együtt a rendszerkönyvtárakban foglalnak helyet.

10.1. Programbetöltés

A programbetöltés adott forrásfájl vagy tárgykód a Prolog fejlesztői környezetbe való betöltését jelenti.

Egyetlen szabványos predikátuma az `ensure_loaded/1`, amelynek paramétere egyetlen fájlnév, vagy fájlnevek listája is lehet. Minden fájlra nézve ellenőrzi annak korát, és csak akkor tölti be, ha szükséges. Némely rendszerben csak *direktívaként* (2) [Der 96], általában azonban korlátozás nélkül hívható, és a betöltött kódot a memóriába fordítja.

A legtöbb rendszerben ezenkívül még rendelkezésünkre áll az értelmező módban való betöltés (`consult/1` vagy egyszerűen a betöltendő fájlok listája (3)) és a fordításos betöltés (`compile/1`) lehetősége, lefordított tárgykód előállítása, betöltése, speciális modul(fájl)kezelők, (`use_module/2` ... (13.2)) stb. [Spl 02, SzB 01]. Az értelmező módban betöltött predikátumok listázhatók (`listing/1`), nyomkövethetők (`trace/0`, `notrace/0`, stb.), míg a fordításos betöltés nagyságrenddel gyorsabb, optimalizált kódot eredményez[Spl 02, SzB 01] .

10.2. Bemenet és kimenet

Csak a soros elérésű szövegfájlok kezelésének alapvető predikátumait tekintjük át [Der 96, Spl 02]. A szövegállományokat `read`, `write`, vagy `append` módban, mégpedig az `open(Fájl,Mód,Csatorna)` utasítással nyithatjuk meg. Ezután műveleteink a csatornára vonatkoznak, egészen annak lezárásáig (`close/1`). A szabványos bemeneti és kimeneti csatornákat nem kell és nem is lehet megnyitni és lezárni. Ezekre `user` néven hivatkozhatunk. Az I/O utasítások karakter vagy term szintűek.

A `peek_code(F,C)` igaz, ha az `F` csatorna aktuális karakterének kódja `C` (megpróbálja illeszteni `C`-vel). A `get_code(F,C)` cél hasonló, de hatására az aktuális karakter a következő lesz. Az `at_end_of_stream(F)` igaz, ha az `F` csatornát már végigolvastuk (`F\=user`). A `put_code(F,C)` utasítás kiírja a `C` kódú karaktert, míg a `nl(F)` sort emel. Például:

```
% appf(F1,F2) :- F1 szövegfájl végére beszúrja F2 -t.
appf(F1,F2) :-
    open(F1,append,A), open(F2,read,R), af(A,R).

af(A,R) :-
    get_code(R,C),
    ( C == -1 -> close(A), close(R)    % R vége
    ; put_code(A,C), af(A,R)
    ).
```

A `read(F,T)` hívás egy teljes termet olvas be, és megpróbálja illeszteni `T`-vel. Mivel tetszőleges term tovább folytatható egy közbevetett műveleti jellel és egy kapcsolódó termmel, a beolvasandó termet az állításokhoz hasonlóan mindig egy pont és legalább egy elválasztó karakter zárja le:

```
| ?- read(user,X), read(user,Y).
|: 12.    a+b*c.
X = 12,    Y = a+b*c
```

A `write(F,T)` utasítás kiírja a `T` *tetszőleges* Prolog termet (akár teljes listát is), lezáró pont nélkül. A neveket (atomokat) azonban sosem teszi aposztrófok közé, hiszen ezek általában kiírandó üzeneteket ábrázolnak:

```
| ?- write(user,'A Pi értéke = '), read(user,Pi).
A Pi értéke = 3.14159265358979323846.
Pi = 3.141592653589793
```

Ha egy másik Prolog program által visszaolvasható kimenetet szeretnénk, a `writeq(F,T)` utasítást használjuk:

```
| ?- writeq(user,apja('Izsák','Jákób')),
     write(user,'.'), nl(user).
apja('Izsák','Jákób').
```

10.3. Önmódosító programok

Az eddigiekben kimondatlanul csak a statikus Prolog predikátumokkal foglalkoztunk. Egy predikátum ugyanis alapértelmezés szerint statikus, és kódja a programbetöltés után már nem változik. Dinamikusként is deklarálhatjuk, például a `:-dynamic(p/2).` deklaráció hatására a `p/2` predikátum dinamikus, azaz futási időben módosítható lesz; így programunk új állításokat is meg tud tanulni.

A deklarációnak mindig meg kell előznie a hivatkozott predikátum első állítását. Egy deklarált – és így létező – predikátumhoz nem szükségszerűen tartozik állítás. Ha egy létező predikátumhoz nem tartozik állítás, természetesen minden rá vonatkozó hívás meghiúsul, de ha a predikátum nem létezik, ez az `existence_error` kivételt (12) váltja ki.

A dinamikus predikátumokhoz újabb állításokat is adhatunk. Az `asserta(K)` utasítás hatására az állítás feje által meghatározott predikátum első állítása lesz a `K` állítás, míg az `assertz(K)` hatására a megfelelő predikátum utolsó állítása lesz.
Tegyük fel például, hogy az `e.pl` fájl tartalma a következő:

```
:- dynamic(elérhető/2).
elérhető(X,X).
```

és az alábbi módon dolgozzuk fel (feltesszük, hogy az `él/2` predikátum egy körmentes irányított gráfot határoz meg):

```
| ?- [e].
{consulting /home/at/pp/pny/lp/e.pl...}
{consulted /home/at/pp/pny/lp/e.pl in module user,
 10 msec 240 bytes}
yes
| ?- assertz((elérhető(X,Y):-él(X,Z),elérhető(Z,Y))).
yes
| ?- listing(elérhető/2).     % elérhető/2 listázása
elérhető(A, A).
elérhető(A, B) :-
        él(A, C),
        elérhető(C, B).
```

A dinamikus állítások törlése a `retract(ÁllításSéma)` utasítással lehetséges, ahol az `ÁllításSéma` *fej* része név vagy összetett term kell legyen, ami a `retract` utasítás által hivatkozott predikátumot azonosítja. A `retract(ÁllításSéma)` cél törli a hivatkozott predikátum első olyan állítását, amely illeszthető az ÁllításSéma termmel. Visszaléptetéssel akár az összes illeszkedő állítás törölhető:

```
| ?-  retract((elérhető(X,Y):-Törzs)).
Y = X,    Törzs = true ? ;
Törzs = él(X,_A),elérhető(_A,Y) ? ;
no
```

Ennek példájára – ha nem volna beépített predikátum – a következőképpen írhatnánk le a `retractall(P)` utasítást, amely az összes, a `P` predikátumhívásra illeszkedő *fej* részű állítást törli:

```
retractall(P) :- retract((P:-_)), fail.
retractall(_P).
```

Ha egy nem létező predikátumra meghívjuk a fenti programmódosító utasítások valamelyikét (`asserta(Állítás)`, `assertz(Állítás)`, `retract(ÁllításSéma)`, `retractall(ÁllításFejSéma)`), akkor ez a nem létező predikátum dinamikus predikátumként létrejön:

```
% betölt(F) :- Beolvassuk az F fájl mondatait, és
% dinamikus predikátumokat hozunk létre belőlük.
% Csak az állításokat vesszük figyelembe.
betölt(F) :- open(F,read,Cs), töltő_ciklus(Cs).

töltő_ciklus(Cs) :-
    read(Cs,Mondat),
    ( Mondat == end_of_file -> close(Cs)    % Cs vége.
    ; Mondat = :-(_) -> töltő_ciklus(Cs)
    ; assertz(Mondat), töltő_ciklus(Cs)
    ).
```

Leggyakrabban a program futása során kiszámított, kikövetkeztetett információk, lemmák és negatív lemmák megőrzésére, újra való kiszámításuk elkerülésére használunk dinamikus predikátumokat.

Globális változóként való használatuk általában ellenjavallt. Ez egyrészt procedurális programozási stílust eredményezhet, másrészt nagyon költséges, mert például egy predikátum minden módosítása az első argumentum indexeléshez használt táblázatának újraépítését igényli, minden *assert* a beszúrandó állítást – függetlenül a benne tárolt adatszerkezetek méretétől – le kell másolja stb.

11. Prolog célok megoldásainak összegyűjtése

Egy Prolog célnak több megoldása is lehet. Mivel azonban ezek a keresési fa különböző ágain állnak elő, a visszalépéses keresés miatt egyszerre sosem állnak rendelkezésünkre. A célok megoldásait összegyűjtő beépített predikátumok közül [Der 96, Kee 90, Spl 02, S-S 94, SzB 01] talán a legfontosabb és legegyszerűbb a `findall(Mit,Honnan,Hová)` utasítás. Itt a `Honnan` egy tetszőleges cél, illetve zárójelezett célsorozat. Ennek a megoldásait a `Mit` term szerint összeállítva a `Hová` listába gyűjtjük össze.

Ha például – a (2.2) részben ismertetett módon – adott a `szülője/2` reláció, kiszámíthatjuk bárki szülei, illetve gyermekei listáját:

```
szülei(X,Ys) :- findall(Y,'szülője'(Y,X),Ys).
gyermekei(X,Ys) :- findall(Y,'szülője'(X,Y),Ys).

% Teszt:
| ?- szülei('Izsák',L).
L = ['Sára','Ábrahám']
```

```
| ?- gyermekei('Izsák',L).
L = ['Jákób','Ézsaú']
```

A `findall(Mit,Honnan,Hová)` cél a `Honnan` cél(sorozat) megoldásait abban a sorrendben gyűjti össze, amelyben azokat megkapjuk. Ha leírjuk a (fél)testvér relációt,

```
tesvér(X,Y) :- 'szülője'(Z,X), 'szülője'(Z,Y), X\==Y.
```

akkor a féltestvérek csak egy, míg a teljes testvérek két szülőn keresztül testvérek. Ha ezek után egy *findall* utasítás segítségével leírjuk a

```
% testvérei(X,Ys) :- X testvérei Ys elemei.
testvérei(X,Ys) :- findall(Y,tesvér(X,Y),Ys).
```

relációt, akkor a teljes testvérek megkettőzve szerepelnek az `Ys` listán:

```
| ?- testvérei('Izsák',L).
L = ['Ismáel','Ismeretlen']
| ?- testvérei('Jákób',L).
L = ['Ézsaú','Ézsaú']
```

Ha ez nem kívánatos, a legtöbb Prolog-megvalósításban rendelkezésünkre áll a `sort(Xs,Ys)` predikátum, amely az `Xs` listát rendezi szigorúan monoton növekvően a `(@<)/2` reláció (8.2) szerint az `Ys` listába, az esetleges elemkettőzéseket értelemszerűen megszüntetve. Ennek segítségével könnyen kitalálhatjuk a

```
% 'gyűjt'(Mit,Honnan,Mibe) :-
%     Honnan megoldásainak halmaza Mit szerint Mibe.
'gyűjt'(Mit,Honnan,Mibe) :-
    findall(Mit,Honnan,Hová), sort(Hová,Mibe).
```

predikátumot. Ezt használva a `testvérei/2`-ben a `findall/3` helyett a megfelelő megoldást kapjuk:

```
% testvérei(X,Ys) :- X testvéreinek halmaza Ys.
testvérei(X,Ys) :- 'gyűjt'(Y,tesvér(X,Y),Ys).
```

Ez tehát már nem ad megkettőzött testvéreket:

```
| ?- testvérei('Jákób',L).
L = ['Ézsaú']
```

12. Kivételkezelés a Prologban

A hiba- és kivételkezelés (exception handling) minden modern programozási nyelvnek, így az ISO Prolognak is fontos részét képezi.

A tiszta Prologban egy predikátumhívás – feltéve, hogy keresési tere véges – meghiúsul vagy sikeresen lefut, esetleg választási pontot hagyva maga után. A beépített eljárások némelyike azonban bizonyos bemenő értékekre nem értelmezhető. Ilyen például az `arg(I,T,A)` hívás, ha `T` egyszerű term vagy `I` negatív; az `X is Exp` cél, ha `Exp` nem aritmetikai kifejezés; vagy a `read(X)` utasítás, ha

formailag hibás termet kellene beolvasnia. Ilyenkor – más magasszintű nyelvekhez hasonlóan – az aktuális eljáráshívás a megfelelő *kivételt* (exception) váltja ki, és a hívás – sikeresség vagy meghiúsulás helyett – ezzel ér véget.

A kivételt mindig egy *állandó* vagy *összetett* term (nonvar term) jelöli, és mesterségesen is kiváltható a `throw(`*Kivétel*`)` utasítás segítségével, ahol *Kivétel* a kivételt jelölő nonvar term.[18]

A kezeletlen kivételek mint futási hibák lépnek fel. Ezeket a Prolog fejlesztői környezet – a megfelelő hibaüzenet kiírásával – saját párbeszédciklusa szintjén kezeli.

A kivételkezelés eszköze a `catch(`*Cél*`,`*KivételMinta*`,`*KezelőCél*`)` utasítás.[19]

A *catch* utasítások pontosan azon kivételek kezelésére szolgálnak, amelyek az első argumentumban levő *Cél* kiértékelése során lépnek fel. A *catch* utasítások első argumentumát ezért *védett* (protected) argumentumnak nevezzük.

A *catch* utasítás tehát először meghívja az első, védett argumentumát paraméterező *Cél* célsorozatot.

Amennyiben a *Cél* kiértékelése során nem lép fel kivétel, vagy ezt (ezeket) a *Cél* maga kezeli, akkor teljesen a *Cél* működése határozza meg a *catch* hívást.

Ha azonban a *Cél* kiértékelése kezeletlen kivételt vált ki (nevezzük ezt *E*-nek), akkor először is készítünk *E*-ről egy friss másolatot[20], majd a *Cél* kiértékelése során végrehajtott összes változóillesztést töröljük, az itt létrehozott választási pontokkal, a hívási verembe betöltött hívási keretekkel, és általában minden visszaléptethető hatással együtt.

Ezután *E* friss másolatát és a *KivételMinta* termet illesztjük.

Ha az illesztés sikeres, akkor a *KezelőCél* célsorozat határozza meg a *catch* utasítás további viselkedését.

Ha azonban *E* friss másolata és a *Kivételminta* illesztése meghiúsul, akkor a környezet szempontjából úgy tekinthető, hogy a *catch* hívás váltotta ki az *E* kivételt. Ilyenkor azt mondjuk, hogy a *catch* utasítás *tovább propagálja* a kivételt.

Végül, ha *E* friss másolatának a *Kivételmintá*val való illesztése ugyan sikeres, de a *KezelőCél* maga is kivált egy kezeletlen *H* kivételt, akkor a környezet szempontjából úgy tekinthető, hogy a *catch* utasítás váltotta ki a *H* kivételt. Ilyenkor is tovább propagálja tehát a kivételt.

A beépített eljárások mindig `error(IsoErr,ImpErr)` alakú kivételeket váltanak ki, ahol `IsoErr` a szabvány hibatermje, míg `ImpErr` rendszerfüggő:

```
| ?- catch( X is a, error(IsoErr,ImpErr), true ).
ImpErr = domain_error(_A is a,2,expression,a),
IsoErr = type_error(evaluable,a/0)
```

Az `olvas(Term)` cél beolvas egy tetszőleges termet a szabványos bemenetről (ez alapesetben a billentyűzet), és az aktuális bejövő sor maradékát eldobja. Ha a bejövő term szintaktikusan hibás, kiírja az ISO szabványnak megfelelő hibatermet, majd újra olvas.

[18] throw = dob, vet, hajít.

[19] catch = elfog, elkap, megragad, megcsíp.

[20] friss másolat = fresh copy : olyan másolat, amelyben a változók helyett módszeresen új, eddig nem használt változókat vezetünk be. Egy változó különböző előfordulásai helyett természetesen ugyanazt az új változót vezetjük be, míg különböző változók helyett különbözőket.

```
olvas(Term) :-
    catch( ( read(user,Term), skip_line(user) ) ,
           error(Hiba,_), ( ír(Hiba), olvas(Term) )
         ).

ír(Term) :- writeq(user,Term), nl(user).

| ?- olvas(Term).
|: a*(b+ .
syntax_error('. cannot start an expression')
|: a*(b+c.
syntax_error(') or operator expected')
|: a*(b+c).
Term = a*(b+c)
```

13. Prolog modulrendszerek

Mivel a Prolog modulrendszerét még nem szabványosították, ezt megvalósításfüggően tárgyaljuk, jegyzetünk bevezetője (4. oldal) szellemében a SICStus Prologra összpontosítva [Spl 02]. További összehasonlító elemzés található [SzB 01]-ben. A modulrendszer lehet *név* vagy *eljárás (predikátum)* alapú.

13.1. A név alapú modulrendszer

A név alapú modulrendszerben minden név (atom) alapértelmezés szerint lokális. Például az `alma` név alapértelmezés szerint minden modulban különböző. A lokalitás feloldása rendszerfüggő. Az *LPA Prolog Professional*-ben például kétszintű a modulrendszer.[21] Az alapértelmezett gyökérmodul alatt vannak az esetleges felhasználói modulok. Ezek a neveket a gyökérmodulba exportálhatják, illetve innét importálhatják. A kifejezetten modulba nem zárt Prolog kód is alapértelmezés szerint a gyökérmodulba töltődik. Ez a modulrendszer egyszerű, de ha egy nevet exportálunk illetve importálunk, akkor minden lehetséges aritással, mint adatnevet és mint predikátumnevet is exportáltuk illetve importáltuk. Az adatnevek lokalitása kényelmetlen, és nem világos, mi a különbség módszertanilag egy *szám* (ami globális) és egy *név* (ami alapértelmezés szerint lokális) között.

13.2. Az eljárás alapú modulrendszer

Az eljárás alapú modulrendszerben – amit a SICStus Prolog leírása [Spl 02] alapján tárgyalunk, de használatos még például az SWI és Quintus Prolog rendszerekben is [SzB 01] – a fentiekkel szemben a nevek, és általában minden term globális, a predikátumok (eljárások) azonban forrásmoduljukra nézve lokálisak. Alapértelmezett a `user` modul: ha nem használjuk a modulrendszert, minden kód ebbe töltődik. Ennek ellenére minden modul azonos szinten van.

A modulfájlok első mondata rendesen egy moduldeklaráció
`:- module(` *modulnév*, *exportlista* `).`
formában, ahol a *modulnév* Prolog atom, az *exportlista* pedig *név/aritás* alakú

[21] Ez az MProlog modulrendszerének [SzB 01] továbbfejlesztése.

predikátum-specifikációk valódi listája: az itt felsorolt eljárások lesznek a nevezett modul nyilvános predikátumai. Ezek lehetnek az adott modulban definiált, vagy ide importált predikátumok is. Ezek importálhatók egy másik modulba, például a
`use_module(` *modulfájlnév*, *importlista* `)`
beépített eljárás segítségével, amit általában *direktívaként* (2) hívunk meg, és szükség esetén a fájl betöltését is elvégzi.

Tegyük fel most, hogy adott a `graph.pl` modulfájl, benne a `gráf` modullal, amely – `él/2` nyilvános predikátumában – egy irányított gráf leírását tartalmazza, vagyis az `él(X,Y)` pontosan akkor igaz, ha az `(X,Y)` a gráf egy éle. Feltesszük továbbá, hogy az `él(X,Y)` cél a gráf éleinek ellenőrzésére és visszadására is alkalmas. A legegyszerűbb ilyen gráfleírások Prolog tényekként sorolják fel az éleket:

```
él(a,b).    él(b,c).    él(b,d).
él(c,a).    él(c,e).    él(d,e).
```

Írjunk most egy modult, amely egyetlen nyilvános predikátumával körmentes utakat keres a fenti gráfon:

```
:- module( keres, [ út/3 ] ).

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

% út(A,Z,Út) :-
%     Út egy A -> Z körmentes út (csúcsok listája).
út(A,Z,Út) :- útja(A,[],Z,Út).

% útja(A,Voltak,Z,Út) :-
%     A Voltak csúcslista fordítottjának és egy A -> Z
%     útvonalnak a konkatenáltja az Út körmentes út.
útja(A,Voltak,A,Út) :- reverse([A|Voltak],Út).
útja(A,Voltak,Z,Út) :-
    él(A,B), % él egy közbenső csúcsba
    B\=A,             % nem hurokél
    \+member(B,Voltak), % nem ciklus
    útja(B,[A|Voltak],Z,Út).
```

Keressünk utakat. Figyeljük meg, hogy a megoldások között az első a *tetszőleges, de egyetlen csúcsból álló út*, mint általános megoldás (vö. 7. oldal):

```
| ?- út(_A,_Z,Út).
Út = [_A] ? ; Út = [a,b] ? ;
Út = [a,b,c] ? ; Út = [a,b,c,e] ? ;
Út = [a,b,d] ? ; Út = [a,b,d,e] ? ;
Út = [b,c] ? <Enter>   % Elég volt!
yes
```

Bizonyos alkalmazásokban szükséges lehet, hogy a gráfot leíró állományt a `keres` modulnak kívülről adhassuk meg. Ehhez elég modulunk első két mondatát megváltoztatni:

```
:- module( keres, [ új_gráf/1, út/3 ] ).
új_gráf(Állomány) :- use_module( Állomány, [él/2] ).
```

Inicializáljuk most modulunkat a fenti gráffal, majd gyűjtsük össze az összes körmentes utat (bárhonnét bárhová), és írassuk ki az eredményt (kiíratáskor a változók belső neve jelenik meg):

```
| ?- új_gráf('../lp/graph').
{module gráf imported into keres}
yes
| ?- findall(Út,út(_,_,Út),_Utak), write(_Utak).
[[_759],[a,b],[a,b,c],[a,b,c,e],[a,b,d],[a,b,d,e],
 [b,c],[b,c,a],[b,c,e],[b,d],[b,d,e],
 [c,a],[c,a,b],[c,a,b,d],[c,a,b,d,e],[c,e],[d,e]]
true ? ;  no
```

13.3. A forrásmodul megváltoztatása

A fent vázolt, szigorú modulrendszer alapvető problémája, hogy a modulok lokális predikátumai nem tesztelhetők közvetlenül, azaz tesztelésükhöz a kód megváltoztatása lenne szükséges. A SICStusban ezért – és még később részletezendő okokból – lehetséges a célok forrásmoduljának megváltoztatása: a célok mindig a megfelelő modulnévvel minősíthetők,

modulnév: *cél*

formában. Például a `keres` modul (41. oldal) `útja/4` predikátumát közvetlenül tesztelhetjük (annak ellenére, hogy ez a modulnak *nem* nyilvános predikátuma):

```
| ?- keres:útja(d,[b,a],E,Út).
E = d,    Út = [a,b,d] ? ;
E = e,    Út = [a,b,d,e] ? ;
no
```

Értelmező módban való betöltés (10.1) esetén ki is listázhatjuk a betöltött alakot a `listing(keres:útja/4)` utasítás segítségével.

A SICStus modulrendszere tehát *nem* a predikátumok elrejtésére, hanem *csak* véletlen ütközéseik megelőzésére szolgál.

Míg a fenti probléma általában mindenféle Prolog modulrendszerre jellemző, a most következő az eljárás alapú modulrendszer sajátja, és a metapredikátumokkal kapcsolatos. Tegyük fel például, hogy szeretnénk egy modul nyilvános szolgáltatásává tenni a `mindre1(P,Q)` predikátumot, amely `P` minden megoldásához végrehajtja a `Q` utasítást, `Q` első megoldására szorítkozva. Kódja megadható

```
% mindre1(P,Q) :- P minden megoldására Q is megoldható
%                  azaz P implikálja Q -t.
mindre1(P,Q) :- \+ ( P, \+Q ).
```

formában. Használatára egy egyszerű példa a következő:

```
| ?- mindre1(member(X,[1,2,3]),(write(X),write(' '))).
1 2 3
```

Ha most a `mindre1/2` az `m` modul nyilvános predikátuma, amit a `h` modulból hívtunk meg, a `P` és `Q` formális paramétereket paraméterező célsorozatok – mint fent a `member(X,[1,2,3])` – az `m` modulban hívódnak meg, ahol lehet, hogy nem léteznek, vagy ami még rosszabb, esetleg egészen más a definíciójuk, mint a `h` modulban. A probléma egy lehetséges megoldása, hogy a hívást paraméterező célsorozatokat minősítjük a hívó modul nevével. Ez azonban nem túl elegáns, és ráadásul hibaveszélyes megoldás, hiszen a hívó modul nevét programfejlesztés közben megváltoztathatjuk. Ezért a predikátumok egyes argumentumaira *modulnév-kiterjesztés* deklarálható. A kiterjesztendő paramétereket ezekben a *metapredikátum-deklarációkban* kettősponttal (vagy egész számmal), a többieket tetszőleges más névvel stb. jelöljük [Spl 02]:

```
:- module( m, [ mindre1/2, 'gyűjt'/3, be/1, be/0 ] ).

:- meta_predicate mindre1(:,:), 'gyűjt'(?,:,?), be(:).
:- dynamic aktuális_fájl/1.

mindre1(P,Q) :- \+ ( P, \+Q ).

'gyűjt'(Mit,Honnan,Mibe) :-
    findall(Mit,Honnan,Hová), sort(Hová,Mibe).

be(F) :- compile(F), legyen_aktuális(F).

be :- aktuális_fájl(F), compile(F).

legyen_aktuális(F) :-
    retractall(aktuális_fájl(_)),
    asserta(aktuális_fájl(F)).
```

Ez azt jelenti, hogy a `mindre1(C,D)` hívás mindkét paraméterét a fordító automatikusan minősíti a hívó modul nevével, a `'gyűjt'(M,C,L)` hívásnak (ld. 38. oldal) azonban csak a második paraméterét. A két hívás tehát a modulnév-kiterjesztés után `mindre1(h:C,h:D)`, illetve `'gyűjt'(M,h:C,L)` lesz.

Modulnév-kiterjesztést igényelhetnek még az állításokat és a programfájlneveket (valamint a *blackboard* kulcsokat [Spl 02]) továbbító paraméterek. A SICStus beépített eljárásait ugyanis a `prolog` modulban definiálták, mint annak nyilvános predikátumait. Ezeket minden modul automatikusan látja. A programmódosító (`asserta/1`, `assertz/1`, `retract/1` stb.) predikátumoknak pedig tudniuk kell, hogy melyik modul predikátumait módosítsák. Hasonlóképpen, a programbetöltő (`consult/1`, `compile/1`, `use_module/2`, stb.) predikátumoknak is tudniuk kell, melyik modulba töltsék be a hívásukat paraméterező programfájlt, illetve modulfájl betöltése esetén melyik modulba importálják annak nyilvános predikátumait [Spl 02]. Ez a modul mindegyik esetben alapértelmezés szerint a beépített eljárást hívó forrásmodul, ezért mindegyik esetben elegendő, hogy a beépített eljárások megfelelő argumentumaira a `prolog` modulban modulnév-kiterjesztést deklaráltak [Spl 02]. Így a vonatkozó predikátumhívásokra a fordító modulnév-kiterjesztést végez, és a meghívott eljárások megkapják a szükséges információt. Általában, ha a SICStus egy beépített predikátumának egy argumentuma modulnév-kiterjesztést igényel (ld. [Spl 02, SzB 01]) akkor a

hívó predikátum megfelelő formális paraméterére is modulnév-kiterjesztést célszerű deklarálni.

Tegyük fel például, hogy olyan fordítóutasítást szeretnénk írni, amely emlékszik az utoljára lefordított állomány nevére, és alapértelmezés szerint ezt fordítja újra. Erre vonatkoznak a fenti példában a `be/[0,1]` predikátumok. (A feladattal kapcsolatban tudnunk kell, hogy a programbetöltő predikátumok alapértelmezés szerint ugyan a hívó modulba töltik be a programfájlt, illetve modulfájl betöltése esetén ide importálják ennek nyilvános predikátumait; de ha a programfájl nevét egy modulnévvel minősítjük a szokásos *modulnév:fájlnév* alakban, akkor a betöltés, illetve importálás a minősítő modulba történik.) A `be(F)` cél, mint látjuk, az `F` fájl lefordítása után feljegyzi, azaz aktuálissá teszi annak nevét, és a `be` cél ezután az `F` fájlt fordítja újra. A `be/[0,1]` predikátumokkal lefordított fájlokra az `m` modulban hajtjuk végre a `compile(F)` hívást. A megfelelő metapredikátum-deklaráció nélkül ezért a `be` utasítások az `m` modulba fordítanának, kivéve, ha a `be/1` predikátumot `be(h:f)` alakban hívjuk meg. A deklaráció hatására azonban a `be(F)` cél `F` aktuális paramétere automatikusan minősítve lesz a cél forrásmoduljának `h` nevével, és a cél végül `be(h:F)` alakban hívódik meg. Ha tehát `F` egy programfájl neve, a `h` modulba fordítunk. Ha viszont például `F = k:f`, a cél a modulnév-kiterjesztés után `be(h:k:f)` lesz. Ilyenkor a belső minősítés érvényes, vagyis a `k` modulba fordítunk. Az automatikus modulnév-kiterjesztés tehát az explicit minősítést nem bírálja felül. (Vegyük észre, hogy a fenti példában nem csak a fájlnevet, hanem annak modulnév-előtagját is feljegyezzük. Így a legközelebbi `be` hívás annak forrásmoduljától függetlenül pontosan az utolsó `be(F)` hívás hatását ismétli meg.)

Befejezésül megjegyezzük, hogy a forrásmodul explicit megváltoztatásának lehetőségével a forrásprogramon belül nem tanácsos gyakran élni, mert ez programunk modulrendszerének lerombolásához vezethet.

14. Kitekintés

Ebben a rövid bevezetőben természetesen nem nyílt mód a logikai programozás átfogó tárgyalására, de még a Prolog nyelvnek és/vagy programozási módszereinek részletes ismertetésére sem. Az alábbi információk, reméljük, megfelelően segítik azt, aki a témában jobban el kíván mélyülni.

14.1. Ajánlott irodalom

A logikai alapok olvasmányos forrásai [Fek 90, Nil 82]. Egy az itt tárgyaltnál általánosabb megközelítést mutat be Kowalsky [Kow 79]. A logikai programozás elméleti vonatkozásait tárgyalja Lloyd [Llo 87]. A Prolog programozás módszertana szempontjából alapvető, kezdőknek és középhaladóknak Sterling és Shapiro könyve [S-S 94], haladóknak O'Keefe műve [Kee 90]. Az ISO Prolog szabvány megismeréséhez ajánljuk Deransart, Ed-Dbali és Cervoni közös alkotását [Der 96]. Warren absztrakt gépét, ami a Prolog megvalósításának szokásos eszköze, és az ezzel kapcsolatos problémákat jól érthetően tárgyalja [Kac 91]. Az ebben a jegyzetben használt SICStus Prolog-változat felhasználói kézikönyve [Spl 02]. Több mint 500, hálózatról letölthető gyakorlati Prolog alkalmazást tartalmaz a Prolog 1000 adatbázis (Prolog 1000 database).

14.2. Klasszikus LP-kiterjesztések

A logikai programozás nem maradt elszigetelt tudományág. A másik alapvető deklaratív programozási megközelítéssel, a funkcionális programozással való egyesítésére kezdettől fogva történtek kísérletek, amelyek jól működő rendszerekhez vezettek [D-L 86, Han 94]. Ezen egyesített, funkcionális-logikai nyelveknek a gyakorlati programozásban való használatát elsősorban talán egy (de facto) szabvány és a megfelelő képzettségű programozók hiánya akadályozza.

Hasonló problémák – meggyőződésünk szerint – a Prolog nyelv széleskörű használatának akadályai is, hiszen a mai napig csak a nyelv magját sikerült szabványosítani, és talán még a modulrendszer szabványának késlekedésénél is nagyobb probléma a szabványos C/C++, Java, adatbáziskezelő, grafikus stb. felületek hiánya. Az, hogy a szükséges felületek a komolyabb Prolog rendszerek részei, csekély vigasz, ha tudjuk, hogy a gyakorlati alkalmazók nem szívesen függnek egy konkrét megvalósítástól, hisz az különböző okokból bármikor kiszorulhat a piacról. Nem véletlen, hogy a Prolog-megvalósítások száma viszonylag magas, de talán fél tucat – évtizedesnél régebbi – Prolog-fejlesztő *van jelen* az üzleti életben. Az, hogy a Prolog programok lassúak, inkább költött, mint valós érvelés. Egyrészt, meglepő módon még mindig a hatvanas évek amerikai próbálkozásainak kudarca kísért [S-S 94], másrészt a túlterhelt progamozóknak a Prolog nélkül is épp elég követni a változásokat. Egy modern Prolog rendszer, teljesítménye alapján a szégyen veszélye nélkül összemérhető mondjuk egy Java- vagy Lisp-megvalósítással. Természetesen egy-egy Prolog-hacker sokat tehet a nyelv rossz hírének fenntartásáért.

Érdekes kísérletek a Prolog objektumorientált kiterjesztései [Mos 94, Spl 02]. Jól alkalmazhatók például szimulációs feladatokban és szakértői rendszerek építésénél. Hasonló elvű, de kimondottan szakértői keretrendszer Phil Vasey *flex* nevű, frame alapú szakértői keretrendszere, amely az *LPA* és a *Quintus* Prologok kiterjesztéseiként áll rendelkezésre. Jelentős üzleti alkalmazásairól tudunk, intelligens lekérdező és nyilvántartó felületek, diagnosztizáló, ütemező stb. rendszerek készítésére használják.

Majdnem minden modern Prolog-megvalósításban rendelkezésünkre állnak – a nyelv beépített kiterjesztéseként – a DCG nyelvtanok (Definit Clause Grammars). Ezek a logikai nyelvtanok elméleti szempontból a nullás típusú nyelvtanok, gyakorlati szempontból az attribútum nyelvtanok kifejezőerejével bírnak. Alkalmazásuk nagyban megkönnyíti nyelvi elemzők, szövegelőállítók, fordítóprogramok, formális és élő nyelvű programfelületek stb. készítését, amelyek a Prolog nyelvhez amúgy is jól illeszkedő és alapvető alkalmazási területeket jelentenek. Még a hagyományos listakezelési feladatok egy része is elegánsan és hatékony kóddal kezelhető a segítségükkel.

14.3. Az ötödik generációs számítógépek és programjaik

Talán a legismertebb, a Prolog alkalmazásaként elhíresült kezdeményezés az az 1981-ben elindított japán kutatási program, amely új típusú, úgynevezett *ötödik generációs*, logikai alapú gépi nyelven programozható sokprocesszoros hardver, és erre épülő intelligens számítógépek, robotok kifejlesztését tűzte maga elé. A fejlesztés alapnyelve egy idő után valójában egyfajta GHC (Guarded Horn Clause) nyelv lett, amely egyrészről támogatja ugyan a részcélok párhuzamos

végrehajtását, de nem támogatja a visszalépéses keresést.

Japán ugyan 1993-ban bejelentette a kutatás sikeres lezárását, de az valójában nem hozta meg a remélt eredményeket. Egyrészről nyilvánvalóvá lett, hogy az általános célú hardverek mai fejlődési üteme mellett nem érdemes egyedi hardvert készíteni (mire elkészül, elavul), másrészt, a visszalépéses keresésről való lemondással az LP éppen az intelligens keresést igénylő alkalmazásokban veszíti el előnyét a funkcionális programozással szemben.

14.4. Újabb irányzatok

A logikai programozás újabb irányzatai közül ígéretes fejlesztést jelent a Mercury nyelv. Ez egy tiszta LP nyelv, amit kifejezetten nagy programok hatékony fejlesztésére dolgoztak ki. Még a környezetkezelő predikátumokat is logikai alapon oldották meg. Modulrendszere, erős, polimorfikus típusrendszere, erős módrendszere (egy hívásban egy paraméter csak tisztán *in* vagy *out* módú lehet), és a kötelezően megadandó determinizmus-információk alapján fordítója kiemelkedően hatékony tárgykódot hoz létre. A típusosság növeli a tárgykód biztonságát, a Prologhoz viszonyítva több programozói hibára derül már fordítási időben fény. Hátránya, hogy a Prologhoz képest megszorított LP modellre épül, ami megnehezíti a programírást, jobban behatárolja a programozó lehetőségeit, és meggátolja néhány általánosan elfogadott, hatékony LP módszer alkalmazását, mint például az eredmény fokozatos közelítéssel való előállítása (5.2). Ezek a hiányosságok a nyelvnek és programozási módszertanának további fejlesztését kívánják.

Erőteljes kutatási irányzat a párhuzamos LP visszalépéses kereséssel párosított változata. A két alapvető megközelítés a *vagy-párhuzamos*, amikor tehát a keresési fa egyes ágait derítjük fel, amennyire lehet párhuzamosan, valamint az *és-párhuzamos*, amikor a célsorozat egyes részcéljainak megoldását párhuzamosítják. Előbbire példa az *Aurora* LP rendszer (Warren, Gigalips terv), és a SICStus Prolog *Muse* változata, a kettő egyesítésére Warren *Andorra-I* rendszere.

A CLP (Constraint Logic Programming) irányzat a nyolcvanas évek végétől fejlődött ki, és napjainkra a logikai programozás legsikeresebb irányzatává nőtte ki magát [Hen 86, Mar 98, SzB 01, Spl 02]. Elsősorban a legkülönfélébb optimalizációs feladatok megoldására használják. Lényege, hogy a feladatleírás deklarativitását és a keresés intelligenciáját úgy növeli, hogy egy-egy speciális feladatkörben felhasználja más tudományterületek megoldó módszereit. A CLP program különös alapelemei olyan célok, amelyek démonként viselkednek, előre adott értelmezési tartományokkal; és változóik behelyettesítésekor, illetve változóik értelmezési tartományának szűkülésekor egymással kölcsönhatásban szűkítik tovább azok értelmezési tartományát, és így a keresési teret. Mindez ráépül a hagyományos, visszalépéses keresésre. Ha egy változó értelmezési tartománya üressé válik, ez is visszalépést eredményez. Ilyen módon a SICStus Prolog is háromféle különböző CLP-kiterjesztést egyesít magában [Spl 02]. Nagyban növelte a CLP népszerűségét, hogy a francia Ilog cég CLP rendszerét a C++ nyelvre építette.

14.5. Összefoglalás

Napjainkra, a sikerek és kudarcok eredőjeként, a logikai programozás helye az összetett, több (egymással kapcsolatot tartó) aktív objektumot tömörítő rendszerek elemeinek *gondolkodó összetevőiben* látszik körvonalazódni. Segítségével elegánsan és hatékonyan fejleszthetők intelligens adatbázis-felületek, keresést, akár hálózati keresést igénylő rendszerek, különböző fajta, nyelvi feldolgozást, élő nyelvi felületet, fordítást igénylő programok, szakértői rendszerek, természetes intelligenciánkat fejlesztő játékprogramok, szimbolikus számításokat igénylő, képlet-átalakító, illetve egyszerűsítő programok, egyenlet és egyenlőtlenségrendszer-megoldók, optimalizáló programok stb.

A logikai programozás a problémáknak a hagyományos programozási nyelvektől eltérő megközelítését kívánja, de éppen ezért új, elegáns megoldásokhoz segít bennünket.

15. Feladatok

1. Alkalmazzuk a Prolog gép célredukciós algoritmusát (6.1) a (5.2) részben föltett `append_(Xs,Ys,[1,2,3])` Prolog kérdésre.

2. Írjunk listakezelő predikátumokat egy lista kezdőszeleteinek, zárószeleteinek, folytonos részlistáinak és részsorozatainak előállítására. A predikátumok adott listakezdőszelet stb. esetén a kérdéses tulajdonság elllenőrzésére is alkalmasak legyenek. Figyeljük meg, hogy a predikátumok megírásakor mikor milyen programozási módszert (az eredmény fokozatos közelítése, akkumulátor módszer stb.) alkalmaztunk. Vizsgáljuk meg a kapcsolódó predikátumhívásokra a keresési fák végességét.

3. Tekintsük a bináris fáknak a (4) rész végén tárgyalt ábrázolását. Írjuk meg ezzel az ábrázolással a keresőfák szokásos műveleit (Üres fa létrehozása ill. ellenőrzése, adott (kulcsú) elem keresése, beszúrása és törlése.)

4. Implementáljuk az általunk ismert listarendező algoritmusokat Prologban. Ügyeljünk arra, hogy az algoritmusok aszimptotikus hatékonysága ne romoljon.

5. Tekintsük a `member_/2` predikátumot (6.3, 21. oldal). Adjunk alternatív, elégséges feltételeket a `member_(Y,Ys)` cél NSTO tulajdonságához. (Vegyük észre, hogy ha egy állításfej kettőzött változót tartalmaz, de a rá vonatkozó célban az állításfej minden változója legfeljebb egy előfordulásának nem alapterm felel meg, a cél NSTO. Megfigyelhetjük azt is, hogy amennyiben `member_(Y,Ys)` célban `Ys` valódi lista, akkor az adott esetben az STO tulajdonság csak `Y`, és az `Ys` egy elemének illesztésével jöhet létre.)

6. Adjunk elégséges feltételeket az `append_(Xs,Ys,Zs)` (5.2), a `rev_app(Xs,Ys,Zs)` (5.3) és a `reverse(Xs,Ys)` (5.4) célok NSTO tulajdonságához.

7. A (6.3) rész végén (22. oldal) felsorolt beépített Prolog predikátumok STO módon is meghívhatók, mert ezekben a program bizonyos termjeit illesztjük egymással. Fogalmazzunk meg elégséges feltételeket a megfelelő célok NSTO tulajdonságára.

8. Vizsgáljuk meg a (7.1) rész predikátumaira hivatkozó `nincs_benne(Xs,Y)`, `nonmember_(X,Xs)`, `metszet(Xs,Ys,Zs)` és `unió(Xs,Ys,Zs)` célokat a keresési tér végessége szempontjából. Adjunk erre elégséges feltételeket. Fogalmazzunk meg elégséges feltételeket a fenti célok NSTO tulajdonságára is.

9. Keressük meg a szakirodalomban a `setof/3` ISO Prolog predikátum leírását [Der 96, Spl 02, S-S 94, SzB 01]. Megállapíthatjuk, hogy hasonló a 11. rész `'gyűjt'/3` predikátumához, de kifejezőereje nagyobb. (A szerző mérései szerint azonban kevésbé hatékony.)

10. Tegyük fel, hogy (11) `'gyűjt'/3` eljárása egy modul nyilvános predikátuma. Adjuk meg az ehhez szükséges metapredikátum-deklarációt.

11. Próbáljuk meg egy sorkezelő modulban megvalósítani a *sor* adattípus szokásos műveleteit. (Ha a nyilvánvalóan adódó valódi listás ábrázolást választjuk, nem minden művelet költsége lesz állandó. Ha az ábrázolásban rendezett párokat használunk, amelyek első tagja egy részleges lista (5), második tagja pedig a részleges lista végén lévő változó, sikerülhet a sor hosszától független, állandó műveletigény megvalósítása.)

12. A (13.2) részben megírtuk az `út/3` predikátumot, amely képes egy tetszőleges véges irányított gráf mélységi bejárására. Írjunk hasonló predikátumot, amely a gráfokat adott csúcstól kezdve szélességi módon járja be. Használjuk fel a 11. feladatban elkészített sorkezelő modult. Vegyük észre, hogy ez az eljárás képes két csúcs között a legrövidebb út meghatározására.

Hivatkozások

[Col 82] Colmerauer A. (author); Clark K.L., Tarnlund S.-A. (eds): *Prolog and Infinite Trees.* **In:** *Logic Programming* pp. 231–251, 1982. Academic Press, London.

[D-L 86] DeGroot D., Lindstrom G.: *Logic Programming: Functions, Relations and Equations.* Prentice Hall, 1986.

[Der 96] Deransart P., Ed-Dbali A.A., Cervoni L.: *Prolog: The Standard (Reference Manual).* Springer-Verlag, Berlin, 1996.

[D-M 93] Deransart P., Maluszyński J.: *A Grammatical view of Logic Programming.* The MIT Press, 1993.

[Fek 90] Fekete I., Gregorics T., Nagy S.: *Bevezetés a mesterséges intelligenciába.* Műszaki Könyvkiadó, Budapest, 1990.

[Han 94] Hanus M.: *The Integration of Functions into Logic Programming: From Theory to Practice.* **In:** *The Journal of Logic Programming,* 1994. Elsevier Science Publishing Co., Inc., New York.

[Hen 86] Henterick P.V.: *Constraint Satisfaction in Logic Programming.* The MIT Press, 1986.

[Kac 91] Kaci H. A.: *Warren's Abstract Machine,* The MIT Press, Cambridge, Massachusetts, London, England, 1991.

[Kee 90] O'Keefe R.A.: *The Craft of Prolog.* The MIT Press, Cambridge, Massachusetts, 1990.

[Kow 79] Kowalski B.: *Logic for Problem Solving.* North-Holland, New York, Amsterdam, Oxford, 1979.

[Llo 87] Lloyd, J.W.: *Foundations of Logic Programming.* Sringer Verlag, second edition, 1987.

[Mar 98] Marriot K., Stuckey P.J.: *Programming with Constraints.* The MIT Press, 1998.

[Mos 94] Moss C.: *Prolog++, The Power of Object-Oriented and Logic Programming.* Addison-Wesley Publishing Company, Wokingham, England, 1994.

[Nil 82] Nilsson N. J.: *Principles of Artificial Intelligence.* Springer-Verlag, 1982.

[Spl 02] *SICStus Prolog 3.9 User's Manual.* Swedish Institute of Computer Science, 2002. http://www.sics.se/isl/sicstuswww/site/documentation.html

[Rob 65] Robinson J. A.: *A Machine-Oriented Logic Based on the Resolution Principle.* J. ACM 12, pp. 23–41, January 1965.

[S-S 94] Sterling L., Shapiro E.: The Art of Prolog (Second Edition). The MIT Press, London, England, 1994.

[SzB 01] Szeredi Péter, Benkő Tamás:
Deklaratív programozás: Bevezetés a logikai programozásba.
Budapesti Műszaki Egyetem, Budapest, 2002.
http://www.inf.bme.hu/∼dp/documents.html
http://www.inf.bme.hu/∼szeredi