1. fejezet

Bevezetés

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

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

A logikai programozás ötlete Robert Kowalskitól származik [3]. Az első Prolog megvalósítást Alain Colmerauer csoportja készítete el a Marseille-i egyetemen 1972-ben [6]. A Prolog Magyarországon is hamar elterjedt, talán azért is mert igény volt egy ilyen magasszintű programozási nyelvre, és az akkoriban leginkább használt deklaratív nyelv, a LISP, itt nem rendelkezett olyan kultúrával, mint pl. az Egyesült Államokban.

Az 1975-ben Szeredi Péter által elkészített Prolog interpreter [4] felhasználásával több tucat, igaz többnyire kisérleti jellegű Prolog alkalmazás készült Magyarországon [7]. A Prolog hatékony megvalósítási módszereinek kidolgozása David H. D. Warren nevéhez fűződik, aki 1977-ben elkészítette a nyelv első fordítóprogramját (az ún. DEC-10 Prolog rendszert), majd 1983-ban kidolgozta a máig is legnépszerűbb megvalósítási modellt, a WAM-ot (Warren Abstract Machine) [11].

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

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

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

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

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

A Prolog magyar nyelvű irodalma meglehetősen szerény, az MProlog rendszert ismertető [12] illetve a Pro-

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

SWI Prolog	`http://www.swi-prolog.org/`
SICStus Prolog	`http://www.sics.se/sicstus`
GNU Prolog	`http://pauillac.inria.fr/~diaz/gnu-prolog/`
The WWW Virtual Library: Logic Programming	`http://vl.fmnet.info/logic-prog/`
CMU Prolog Repository	`http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/lang/prolog/0.html`
Prolog FAQ	`http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/faq/prolog.faq`
Prolog Resource Guide	`http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/faq/prg_1.faq` `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/faq/prg_2.faq`

1.1. táblázat. Prolog információ-források

log eset-tanulmányokat tartalmazó [13] áll rendelkezésre. Egy rövid Prolog fejezet szerepel a Mesterséges Intelligencia c. monográfiában is [5].

A jegyzet felépítése

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

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

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

Jelölések

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

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

Köszönetnyilvánítás

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

Hibajelentés

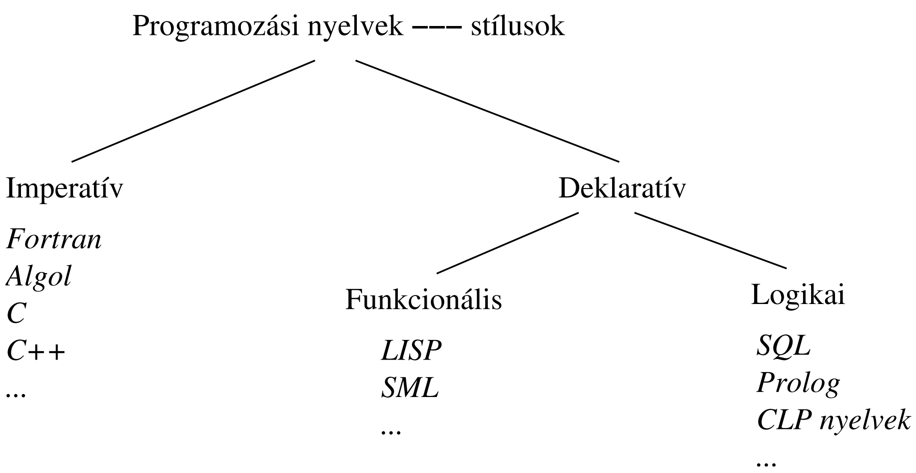
A szerzők köszönettel fogadnak a jegyzettel kapcsolatos bármilyen észrevételt (sajtóhibákat és tartalmi megjegyzéseket egyaránt), a `szeredi@cs.bme.hu` email-címen.

2. fejezet

Deklaratív programozás

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

2.1. Programozási nyelvek osztályozása



2.1. ábra. A PROGRAMOZÁSI NYELVEK OSZTÁLYOZÁSA

Mint a fenti ábra mutatja, a programozási nyelveket alapvetően két csoportba sorolhatjuk. A legtöbb nyelv az ún. *imperatív* nyelvek családjába tartozik: ezeket az jellemzi, hogy *felszólító* módban, parancsok, utasítások segítségével írjuk le az elvégzendő feladatot. Az imperatív nyelvek tipikus példája az assembly, de ilyen a Pascal, C, C++, Java vagy akár — hogy egzotikusabb nyelveket is említsünk — a Perl, a PHP és a Python nyelv is.

Ezzel szemben a *deklaratív* nyelvekben egyenleteket, állításokat írunk le, azaz alapvetően *kijelentő* módban programozunk. Deklaratív nyelv a Prolog, a LISP, a különféle ML megvalósítások, az Ericsson által kifejlesztett Erlang stb.

Míg egy imperatív nyelvű program esetén a hangsúly az algoritmuson van, azaz azon, hogy **hogyan** oldjuk meg a feladatot, addig egy deklaratív programban inkább magát a feladatot írjuk le, azaz azt, hogy **mit** kell megoldani. A deklaratív programozással kapcsolatban sűrün használt jelszó a „MIT és kevésbé HOGYAN" („WHAT rather than HOW"): a cél az, hogy a programozónak inkább azt kelljen leírnia, hogy MIT vár a programtól, és minél kevésbé azt, hogy HOGYAN kell ezt elérni.

Ezt másképpen úgy is mondhatjuk, hogy egy imperatív program esetén viszonylag pontosan át tudjuk látni, hogy a program végrehajtása milyen lépésekből, milyen állapotváltozásokból fog állni. Ezzel szemben egy deklaratív program esetén az „elemi” lépések sokkal összetettebbek: pl. egy egyenletrendszer megoldása, egy következtetési lépés elvégzése stb., ezért a végrehajtás pontos menetét esetleg nem tudjuk követni. De ez sokszor nem is szükséges, hiszen leírtuk a megoldandó feladatot, és a végrehajtást a deklaratív nyelv értelmező- vagy fordítóprogramjára bízhatjuk.

Egy másik fontos különbség az imperatív és deklaratív programozási irányzatok között a változó-fogalomban mutatkozik meg. Az imperatív nyelvekben a változó egy adott memóriahelyen tárolt aktuális értéket jelent. Egy imperatív program „lényege” az, hogy egy változónak ismételten új és új értéket adunk. Erre példa lehet az alábbi faktoriális-kiszámító program, amely egy `f` változó ciklusban ismételt szorzásával állítja elő a kívánt értéket:

```
int faktorialis(int n) {
    int f=1;

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

Ezzel szemben a deklaratív programozási nyelvek változói a matematika változó-fogalmának felelnek meg: egyetlen konkrét, bár a programírás idején[1] még ismeretlen értéket jelölnek. A deklaratív nyelvekben nincs értékadás, egy `x=x+1` alakú programelem értelmetlen vagy hamis, hiszen nem létezhet olyan `x` szám, amelyre a fenti egyenlet teljesülne. Ezért szokás a deklaratív nyelveket az ún. *egyszeres értékadású* nyelvek közé sorolni. Az egyszeres értékadás tulajdonságát a párhuzamos programozás kutatói vezették be, mivel úgy találták, hogy az ilyen tulajdonságú nyelvek párhuzamos végrehajtása sokkal könnyebben megvalósítható, mint a hagyományos, imperatív nyelvek esetén.

Ha deklaratív módon szereténk leírni faktoriális-kiszámító programunkat, akkor rekurziót kell használnunk, mint pl. az alábbi SML nyelvű programban:

```
fun  faktorialis 0 = 1
   | faktorialis n = n * faktorialis (n-1);
```

Ez a program, amely „kijelenti”, hogy nulla faktoriálisa egy, illetve minden más szám faktoriálisa a szám és az eggyel kisebb szám faktoriálisának szorzata.

Könnyen írhatunk a fentihez hasonló C programot is:

```
int faktorialis(int n) {
    if (n<=1)
      return 1;
    else
      return n*faktorialis(n-1);
}
```

Ebből a példából is látszik, hogy az imperatív és deklaratív nyelvek közötti határvonal nem mindig éles, és nem csak maguktól a nyelvektől függ. Azaz bizonyos típusú problémák jobban illeszkednek a deklaratív, míg mások az imperatív szemlélethez. Érdekességként megemlíthető, hogy mivel a számítógépek gépi nyelve szinte kivétel nélkül imperatív, ezért a deklaratív nyelvek implementációit is imperatív nyelveken írják.

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

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

[1] A Prolog nyelv ún. logikai változója még a futás egy részében is meghatározatlan maradhat.

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

2.2. Első példaprogram — családi kapcsolatok

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

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

A fentiek értelmében tehát Imrének szülője István és Gizella, Istvánnak Géza és Sarolt stb. Kérdés tehát, hogy egy konkrét személy esetén kik annak a nagyszülei.

C nyelvű megoldás

Egy lehetséges C megvalósítás lehet az alábbi:

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

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

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

Egy SML megoldás

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

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

Az SML funkcionális nyelvű programban maga az adatbázis is egy függvény, amely a személyekhez a szüleik listáját rendeli (`[Elem1,Elem2, ...,ElemN]` egy `N` $\geq$ 0 elemű listát jelöl). Erre építve definiáljuk a `nagyszulok` függvényt, amely egy személyhez a nagyszülei listáját kell rendelje. SML-ben a függvény meghívását egyszerűen a függvény nevének és az argumentumnak az egymás után írásával jelöljük, pl. a `(szulo g)` a `szulo` függvényt hívja meg `g`-re. Kövessük nyomon a `nagyszulok "Imre"` kiértékelésében levő három egymásba skatulyázott függvényhívást: `List.concat (map szulo (szulo "Imre"))`. Először a legmélyebb hívás történik meg, a `szulo "Imre"`. Végeztessük el a hívást az SML rendszerrel!

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

A `>` jellel kezdődő sor a rendszer válasza, benne az `=` jel után a függvényhívás értékét láthatjuk. Vegyük észre, hogy a sor végén megjelenik az érték *típusa*, esetünkben `string list`, azaz füzérek listája.

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

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

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

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

SQL megoldás

```
SQL> create table szulok (gyerek char(30), szulo char(30));
SQL> insert into szulok values ('Imre', 'István');
SQL> insert into szulok values ('Imre', 'Gizella');
SQL> insert into szulok values ('István', 'Géza');
SQL> insert into szulok values ('István', 'Sarolt');
SQL> insert into szulok values ('Gizella', 'Civakodó Henrik');
SQL> insert into szulok values ('Gizella', 'Burgundi Gizella');

SQL> create view nagyszulok as select fiatal.gyerek, oreg.szulo
  -> from szulok as fiatal, szulok as oreg
```

```
  -> where fiatal.szulo = oreg.gyerek;

View created.
```

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

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

```
SQL> select * from nagyszulok;

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

SQL>
```

Prolog nyelvű megoldás

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

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

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

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

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

A fenti Prolog programot például a következőképpen hívhatjuk meg:

```
| ?- nagyszülője('Imre', NSz).

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

no
```

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

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

```
| ?- nagyszülője(U, 'Géza').

U = 'Imre' ? ;

no
```

A különböző nyelvű megoldások összehasonlítása

Vizsgáljuk meg, hogy az egyes nyelveken milyen módon oldottuk meg ezt a keresési feladatot!

C-ben erre egy (kétszeres) ciklus szolgált, amely egy literálokat tartalmazó C struktúrákból álló tömbön futott végig. SQL-ben ezt a feladatot rábíztuk magára a rendszerre és beépített adatbázis-kereséssel dolgoztunk. Figyeljük meg, hogy az SQL nézet „csak” leírja, hogy milyen párok szerepelhetnek abban, arról, hogy a megoldást ténylegesen hogyan gyűjtjük ki a táblákból, nem szól. Az SML nyelvű megoldás egy magasabbrendű függvényt használ arra, hogy egy műveletet egy lista minden elemére sorra elvégezzen. A Prolog megoldás a Prolog rendszer beépített mintaillesztéses eljáráshívásán alapszik. Az utóbbi két esetben is igazak az SQL-nél elmondottak, azaz például a Prolog kód mindössze csak *deklarálja*, hogy mit értünk `nagyszülője` kapcsolat alatt, a tényleges futás lépéseinek kikövetkeztetése már a rendszer feladata.

Jelentős különbség van a megoldások között abból a szempontból is, hogy hogyan kezelik az összetett feltételeket, azaz azt, hogy nem egyszerűen valaki szüleit, hanem valaki szüleinek a szüleit keressük. A C-nyelvű megoldás erre kétszeres, egymásba skatulyázott ciklust használt, az SML leképezések komponálásával (azaz függvényhívások egymásba skatulyázásával) dolgozott. A Prolog megoldás relációk konjukciójának képzésére épített, láthattuk, hogy a `nagyszülője` szabály két azonos reláció (`szülője`) **és** kapcsolata.

Érdemes még észrevenni ezen kívül, hogy az SML nyelvű funkcionális megoldás a magasabbrendű függvényeknek köszönhetően rendkívül tömör, valamint azt, hogy a Prolog megoldás többirányú. Azaz egy Prolog program sokszor több függvénykapcsolatnak felel meg. Ezeket mi „ingyen” kapjuk, míg például C vagy akár SML esetében a „Kik Géza unokái?” kérdés megválaszolásához egy teljesen új függvényt kellene írnunk.

Fontos mejegyeznünk még azt is, hogy érthetőség, tesztelhetőség, karbantarthatóság szempontjából a deklaratív megvalósítások tömörsége nagy előnyt jelent.

2.3. Második példaprogram — bináris fák bejárása

Ebben az alfejezetben egy, az előzőnél bonyolultabb példa segítségével próbáljuk meg illusztrálni a programozási paradigmák közti különbségeket és hasonlóságokat. Példánkban egy bináris fa levélösszegének kiszámítását tűzzük ki célul. Ehhez szükségünk lesz egy bináris fákat leíró adatstruktúrára. Nyelvfüggetlen módon megfogalmazva egy ilyen bináris fa

- vagy egy levél (`leaf`), amely egy egészet tartalmaz
- vagy egy csomópont (`node`), amely két fára mutat (`left`, `right`)

Ezt C-ben például így tudjuk leírni:

```
enum treetype Node, Leaf;
struct tree {
    enum treetype type;
    union {
        struct {
            int value;
            } leaf;
        struct {
            struct tree *left;
            struct tree *right;
            } node;
    } u;
};
```

Itt tehát definiálunk egy `tree` nevű C struktúrát, amelynek két mezője van. Az első mező az adott elem fajtáját (csomópont vagy levél) leíró enumerációs típusú változó, míg a második egy `u` nevű `union` struktúra. Ez annyit jelent, hogy `u` **vagy** egy `leaf` **vagy** egy `node` struktúrára mutat. Hogy éppen melyikre, azt a `type` mezőben tárolt enumerációs konstans mondja meg.

Megjegyezzük, hogy az ilyen adatstruktúrákat `megkülönböztetett uniónak` nevezzük, mert a fa minden szintjén a `type` mező értéke megkülönbözteti a lehetséges részfa-fajtákat.

SML nyelven egy ilyen megkülönböztetett úniót az alábbi tömör és jól olvasható deklarációval írhatunk le:

```
datatype Tree =
         Leaf of int
       | Node of Tree*Tree
```

Ezekután a `Node(Leaf(2),Node(Leaf(3),Leaf(1)))` SML kifejezés egy olyan bináris fát jelöl, amely két csomópontból és három levélből áll. A 0. szinten egy csomópont, az elsőn egy 2 értékű levél és egy újabb csomópont és végül a második szinten két levél található.

A Prolog nyelvbn nincsen szükség adattípus-deklarációra, mivel a nyelv nem típusos. Ennek ellenére érdemes megjegyzés formájában megadnunk a bináris fa adatstruktúra leírását, valahogy így (a `%` jellel kezdődő sorok Prologban megjegyzések):

```
% :- type tree --->
%        leaf(int)
%      | node(tree,tree).
```

Nézzük ezek után, hogy hogyan tudjuk kiszámítani egy bináris fa levélösszegét! Egy bináris fa levelei értékének összege:

- egy levél esetén a levélben tárolt egész,
- egy csomópont esetén a két részfa levélösszegének összege.

Ezt a *definíciót* valósítjuk most meg különféle nyelveken.

C nyelvű megoldás

```
int sum_tree(struct tree *tree) {
  switch(tree->type) {
  case Leaf:
   return tree->u.leaf.value;
```

```
  case Node:
   return
    sum_tree(tree->u.node.left) +
    sum_tree(tree->u.node.right);
    }
}
```

Látható, hogy ez a megoldás teljesen deklaratív, abban az értelemben, hogy könnyedén kiolvasható belőle a fenti definíció. Vegyük észre, hogy ez azon múlik, hogy nem használtunk értékadást!

Tekintsünk most egy másik, imperatív megoldást. Ez hatékonyabb, hiszen kevesebb rekurzív hívást használ, de ugyanakkor sokkal nehezebben érthető, helyessége nehezebben látható át.

```
int sum_tree(struct tree *tree) {
    int sum = 0;

    while (tree->type == Node) {
        sum =+ sum_tree(tree->u.node.left);
        tree = tree->u.node.right;
    }
    sum =+ tree->u.leaf.value;
    return sum;
}
```

SML megoldás

```
fun sum_tree(Node(Left,Right))
         = sum_tree Left +
           sum_tree Right
  | sum_tree(Leaf(Val)) = Val
```

Az SML megoldás a definíció szinte szó szerinti megismétlése. Látható, hogy egy ilyen feladatot milyen könnyen programozhatunk be egy deklaratív nyelven.

Nézzünk meg ezután egy SML példafutást! Ehhez el kell indítanunk egy SML rendszert, esetünkben a MOSML-t és be kell töltenünk az előbb megírt programunkat:

```
% mosml
Moscow ML version 2.00 (June 2000)
Enter ‘quit();’ to quit.
- use "tree.sml";
[opening file "tree.sml"]
(...)
val sum_tree = fn : Tree -> int
[closing file "tree.sml"]
-
```

A rendszer az általunk megadott függvény kódja alapján előállítja annak típusát: `val sum_tree = fn : Tree -> int`. Ez azt jelenti, hogy `sum_tree` egy olyan függvény, amely egy (általunk definiált) `Tree` típusú bemenő argumentumot vár és `int` típusú értéket szolgáltat.

Ezek után lássuk, hogy a már ismert `Node(Leaf(2),Node(Leaf(3),Leaf(1)))` fára mit mond a `sum_tree` függvény:

```
- sum_tree(Node(Leaf(2),Node(Leaf(3),Leaf(1))));
> val it = 6 : int
```

Azaz az adott fa levélösszege 6.

Befejezésül megmutatjuk, hogy a `quit()` paraméter nélküli függvényhívás segítségével léphetünk ki az SML értelmezőből:

```
- quit();
%
```

Prolog megvalósítás

```
sum_tree(leaf(Value), Value).
sum_tree(node(Left,Right), S) :-
        sum_tree(Left, S1),
        sum_tree(Right, S2),
        S is S1+S2.
```

Az SML megoldáshoz hasonlóan a Prolog kód is a definíció megismétlése. Vegyük észre, hogy itt a a „fa összege” fogalomnak nem egy függvény, hanem egy kétargumentumú, `sum_tree` nevű *reláció* felel meg, amelynek első argumentuma a bináris fa, míg a második, kimenő argumentuma a levélösszeg

A program két állítást tartalmaz (akár csak a definíció). Az első egy tényállítás, amelynek jelentése az, hogy egy egylevelű fa levélösszege megegyezik a levél értékével: a `sum_tree` reláció fennáll a `leaf(Value)` és `Value` Prolog adatok között, tetszőleges `Value` érték esetén.

A második állítás egy szabály, amely kijelenti, hogy egy csomópont levélösszege a részfák levélösszegeinek összege. Ez itt a következőképpen jelenik meg: egy `node(Left,Right)` alakú fa levélösszege `S`, feltéve hogy a `Left` fa levélösszege `S1`, a `Right` fa levélösszege `S2`, és az `S1` és `S2` számok összege `S` (a legutolsó feltétel, a predikátum utolsó sora, egy ún. beépített eljárás meghívása). Vegyük észre, hogy azt a számítási szabályt, amelyet korábban egyetlen függvénykifejezésként tudtunk megfogalmazni, most három egymás mellé helyezett, **és** kapcsolatban lévő relációval írtuk le.

Lássunk most egy példafutást!

```
% sicstus -f
SICStus 3.9.1 (x86-win32-nt-4): Wed Jun 19 13:03:11  2002
Licensed to BME DP course
| ?- consult(tree).
{consulting /home/szeredi/peldak/tree.pl...}
{consulted /home/szeredi/peldak/tree.pl in module user, 0 msec 704 bytes} yes
| ?- sum_tree(node(leaf(2),node(leaf(3), leaf(1))), Sum).
Sum = 6 ? ;
no
```

A SICStus Prolog rendszer indulása után betöltöttük a programunkat, majd feltettük azt a kérdést, hogy mennyi a levélösszege a szokásos fának. Nem meglepő módon a válasz itt is az, hogy a levélösszeg 6.

Beszéltünk arról, hogy egy Prolog program sokszor több függvénykapcsolatnak felel meg. Ugyanezt a programot *minden változtatás nélkül* felhasználhatjuk arra is, hogy ellenőrizzük, hogy „vajon igaz-e az, hogy egy adott fának a levélösszege egy adott érték”. Például kérdezzük meg, hogy igaz-e, hogy a fánk levélösszege 10?

```
| ?- sum_tree(node(leaf(2),node(leaf(3),leaf(1))),10).
no
| ?-
```

Azt is megtehetjük, hogy olyan fát keresünk, amelynek levélösszege egy adott érték:

```
| ?- sum_tree(Tree, 10).
Tree = leaf(10) ? ;
{INSTANTIATION ERROR: _76 is _73+_74 - arg 2}
```

Első megoldásként megkapjuk azt a fát, amely csak egy levelet tartalmaz (és melynek értéke 10). További megoldás kérésekor (erre szolgál a `;`) azonban hibát kapunk. Ennek fő oka az `is` beépített eljárás „gyengesége”, de ezt itt most nem részletezzük.

Fontos, hogy a fenti Prolog kód működéséhez nem szükséges semmilyen előzetes típusdeklaráció. Míg az SML értelmező hibát jelezne, ha a `sum_tree` függvényt nem `Tree` típusú kifejezésre alkalmaznánk (például mert nem egész szám lenne egy levél értéke), a Prolog programunk nem érzékeny arra, hogy egész, vagy lebegőpontos számokat használunk:

```
| ?- sum_tree(node(leaf(2.1),node(leaf(3),leaf(1))),A).
A = 6.1 ? ;
no
```

Befejezésül itt is megmutatjuk, hogy a Prolog rendszerből a `halt` beépített eljárás segítségével léphetünk ki;

```
| ?- halt.
%
```

2.4. A deklaratív programozás jellemzői

Ebben az alfejezetben megpróbáljuk tömören összefoglalni a deklaratív programozási irányzatok alapvető jellemzőit, az előző fejezetekben ismertetett programpéldák alapján.

2.4.1. A funkcionális programozás

A funkcionális programozás alapötlete az, hogy fügvényeket kifejezésekkel definiálunk, ahol a kifejezések függvényhivatkozásokból épülnek fel. A függvény maga is érték, teljesen egyenrangú a többi (pl. szám-)értékkel. Így fontos szerepet kapnak a magasabbrendű, azaz pl. függvényparaméterrel bíró függvények. A nagyszülőket előállító példában ilyen volt a `map` könyvtári függvény, amely az első paraméterében megadott függvényt a másodikban megadott lista minden elemére alkalmazza.

A funkcionális programok fontos tulajdonsága a *hivatkozási átlátszóság* (referential transparency). Ez azt jelenti, hogy egy függvényhivatkozás mindig átirható az adott függvényt definiáló kifejezésre, természetesen a megfelelő paraméter-behelyettesítések elvégzése után.

A funkcionális programozás első megvalósítása a LISP nyelv (LISt Processing language) volt. A nyelv, mint neve is mutatja, egyetlen adatszerkezetre, a lista-fogalomra épít, és ma is a mesterséges intelligencia egyik fő nyelve. Napjainkban a funkcionális programozás egyik modern megvalósítása az SML, amely az eddig elmondottakon kívül nagyon erős típusrendszerrel is rendelkezik. A típusrendszer lehetővé teszi, hogy szinte hiba nélkül programozzunk. Ez a gyakorlatban azt jelenti, hogy ha egy program lefordul, azaz nincs benne (típus-)hiba, akkor nagy valószínűséggel azt csinálja majd, ami a programozó szándéka volt.

Az SML további előnyei közé tartozik, hogy egy függvényt több ún. klózzal definiálhatunk, amelyek közül mintaillesztéssel választ a rendszer. Ennek következtében az adatstruktúrák könnyen és áttekinthetően kezelhetőek, a kód nagyon tömör és jól olvasható lehet.

A funkcionális nyelvek több nagy irányban fejlődnek. Egy újfajta, ún. lusta kiértékelési mechanizmust biztosít a Haskell, illetve a Clean nevű nyelv, a párhuzamosíthatóság kérdéskörére koncentrál a Parallel Haskell, illetve a Concurrent ML. A típusrendszer bővítésével foglalkozik a Objective CAML, valamint a Haskell és a Clean is.

2.4.2. A logikai programozás

A logikai programozás alapgondolata az, hogy a matematikai logika nyelvét, illetve egy a logikán alapuló nyelvet használjunk programozási nyelvként; végrehajtási módszerként pedig logikai következtetési ill. tételbizonyítási eszközöket használjunk. Ez utóbbi már nem a programozó, hanem az adott logikai nyelvet megvalósító rendszer feladata.

A logikai programozás első megvalósítása a Prolog nyelv. Egy Prolog program elemei logikai állításoknak felelnek meg. Emlékezzünk vissza, hogy a `nagyszülője` reláció definíciója Prolog formában így nézett ki:

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

Ezt a definíciót úgy olvastuk ki, hogy „egy gyerek-nek nagyszülője `Nagyszülő`, ha van egy olyan személy (`Szülő`), aki a gyereknek szülője és ezen személy szülője `Nagyszülő`".

Ez tuljdonképpen egy elsőrendű logikai állítás, amely formálisan így írható le:

$$\forall U \forall N \forall Sz(\texttt{nagyszülője}(Gy, N) \leftarrow \texttt{szülője}(Gy, Sz) \wedge \texttt{szülője}(Sz, N))$$

Azt látjuk tehát, hogy amikor Prologban programozunk, akkor valójában elsőrendű logikai állításokat fogalmazunk meg.

Eljárásos értelmezés

A Prolog nyelv esetében az elsőrendű logika nyelvét az ún. *Horn klózokra* szűkítjük le, és a programok futásához egy nagyon egyszerű tételbizonyítási módszert használunk (lásd a C függeléket). A Horn klózok olyan

$$a \leftarrow (b_1 \wedge b_2 \wedge \ldots \wedge b_n)$$

alakú implikációk, ahol mind a, mind a b_i-k elemi állítások (például `szülője(...)`, `nagyszülője(...)`) és az összes előforduló változót univerzális kvantorral lekötöttnek tekintünk. Ez pontosan megfelel a `nagyszülője` nevű szabály logikai átiratánál látottaknak.

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

Például a `nagyszülője` szabály eljárásos értelmezése a következő:

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

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

Az eljárások paraméter-átvétele kétirányú mintaillesztéssel (egyesítéssel) történik. Ennek folytán a bemenő és kimenő paraméterek nincsenek megkülönböztetve, azaz ugyanaz az eljárás többféleképpen is használható. Így egy Prolog program sokszor több függvénykapcsolatnak (relációnak) felel meg. Például:

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

A logikai programozás egyik új irányzata a korlát logikai programozás (CLP), amelynek egyes megvalósításai bizonyos Prolog implementációkban (pl. SICStus, GNU) könyvtárak formájában hozzáférhetőek. A Mercury nevű logikai nyelv egy típusos kiterjesztés, míg az Aurora, Andorra és Mozart (Oz) nyelvek a rugalmasabb vezérlésen kívül, a párhuzamos végrehajtást is támogatják.

3. fejezet

A Prolog nyelv alapjai

Ez a fejezet a Prolog nyelv alapelemeit mutatja be. Az első alfejezet a Prolog nyelv közelítő szintaxisát írja le, mellyel a cél az volt, hogy a viszonylag „száraz" teljes Prolog szintaxis ismertetése helyett érthetőbb bevezetést nyújtson a Prolog nyelvbe. A második alfejezet a Prolog nyelv szemantikáját, az egyesítési algoritmust, a Prolog végrehajtási algoritmus egyszerűsített modelljét, valamint az egyszerűbb Prolog vezérlési szerkezeteket mutatja be. A harmadik alfejezet a Prolog lista-fogalmát ismerteti, bemutatja jelölésmódját és a beépített listakezelő eljárások egy részét. A következő alfejezet feladata a Prologban központi szerepet játszó visszalépéses keresés ismertetése. Itt lesz még szó az indexelésről is. Az ötödik rész foglalkozik a legalapvetőbb beépített eljárásokkal, míg a hatodik a feltételes szerkezeteket és a negáció fogalmát ismerteti. Az utolsó előtti alfejezet mutatja be a teljes Prolog szintaxist, majd a fejezetet a Prolog egy lehetséges típusfogalmának ismertetése zárja.

3.1. A Prolog nyelv közelítő szintaxisa

3.1.1. A Prolog programok elemei

Tekintsük az előző fejezetben megismert levélösszeg-számító program Prolog kódját. Ezen a példán mutatjuk be a Prolog programok legfontosabb elemeit.

```
sum_tree(leaf(Value), Value).        % 1
sum_tree(node(Left,Right), S) :-     % 2
        sum_tree(Left, S1),          % 3
        sum_tree(Right, S2),         % 4
        S is S1+S2.                  % 5
```

A fenti Prolog kód definiálja a `sum_tree` nevű **predikátumot** vagy más néven **eljárást**. A logikai nyelvek és így a Prolog is a predikátum fogalmára építenek. A `sum_tree` predikátum egy relációt ír le egy bináris fa és egy érték között. Nevezetesen azt a relációt, amely egy fához annak levélösszegét rendeli.

A `sum_tree` predikátum két **állításból** áll. Az első azt fejezi ki, hogy egy levél levélösszege a levél értéke, a második azt, hogy egy csomópont levélösszege a részfák levélösszegeinek összege. Ezen két állítás *együtt* definiálja a `sum_tree` nevű relációt.

A Prolog állításoknak több fajtájuk létezik. A legegyszerűbb állítások a relációs adatbázistáblák sorainak felelnek meg. Ilyenek voltak a `szülője` predikátum állításai az előző fejezetben, pl: `szülője('Imre', 'István')`.

Ez egy ún. **tényállítás**, amely azt állítja, hogy Imre szülője István. A tényállítások feltétel nélkül igaz állítások. Változót is tartalmazó tényállítással általános tudást is kifejezhetünk, erre láthatunk példát a `sum_tree` predikátum első sorában:

```
sum_tree(leaf(Value), Value).
```

Itt azt állitjuk, hogy egy **tetszőleges** `Value` érték esetén feltétel nélkül igaz az, hogy az egyetlen `Value` értékű levélből álló fa levélösszege a `Value` érték.

A tényállításokon kívül léteznek bonyolultabb állítások Prologban, ezeket **szabályoknak** hívjuk. A szabály egy ún. **fej**- és **törzs**-részből áll, amelyeket a `:-` jelsorozat választ el egymástól. A Prolog-állítás kifejezés szinonimájaként, azaz a szabály és tényállítás fogalmak gyűjtőneveként használatos a **klóz** elnevezés is.

Mint ahogyan a `sum_tree` predikátum esetében is látszik, egy predikátum általában több klózból áll. A szakasz elején található kód így egy két klózból álló predikátum definíciója. Az első klóz egy tényállítás, a második egy szabály. A szabály feje a második sorban, a törzse a $3-5$. sorban található. Az állítás **funktora** `sum_tree/2`, amelyből kiolvasható a predikátum neve, illetve argumentumainak száma.

Az eddig elhangzottakat az alábbi módon foglalhatjuk össze:

⟨Prolog program⟩	::=	⟨predikátum⟩ ...	
⟨predikátum⟩	::=	⟨klóz⟩ ...	{azonos funktorú}
⟨klóz⟩	::=	⟨tényállítás⟩.␣ |	
		⟨szabály⟩.␣	
⟨tényállítás⟩	::=	⟨fej⟩	
⟨szabály⟩	::=	⟨fej⟩ :- ⟨törzs⟩	
⟨törzs⟩	::=	⟨cél⟩, ...	
⟨cél⟩	::=	⟨kifejezés⟩	
⟨fej⟩	::=	⟨kifejezés⟩	

Egy Prolog program egy vagy több predikátumból áll. Egy predikátum egy vagy több (de azonos funktorú, tehát azonos nevű és argumentumszámú) klózból épül fel. Egy klóz lehet tényállítás vagy szabály. Mindkét esetben a klózt ponttal kell lezárni, ami után kötelezően legalább egy nem látható karakternek (szóköznek, újsornak stb.) kell következnie. Egy tényállítás csak egy fejből áll, míg a szabályok fej- és törzs-részből állnak, amelyeket a `:-` jelsorozat választ el egymástól. Egy törzs célok vesszővel elválasztott sorozata, ahol a vessző **és** kapcsolatot jelent a célok között. A „Prolog cél" kifejezés helyett gyakran használjuk majd a „hívás" szót is, az „eljárástörzs" helyett pedig a „célsorozat" kifejezést.

Tényállítás esetén is beszélhetünk a klóz törzséről, amelyet üresnek tekintünk. Bizonyos helyzetekben viszont a tényállítás törzsének a `true` azonosan igaz beépített eljáráshívást tekintjük.

A fenti szintaxisban megmutattuk, hogy a Prolog program hogyan épül fel célokból és (klóz)fejekből. Ezek mindegyike a Prolog kifejezés szintaktikus kategóriába tartozik. Erről szól a következő szakasz.

3.1.2. Prolog kifejezések

A Prolog nyelv kifejezés-fogalma az alábbi szintaxis szerint épül fel. A jobb oldalon látható angol szavak az adott kifejezésfajta angol nevét, és egyben a kifejezésfajtát ellenőrző beépített eljárás nevét adják meg (pontosabban lásd alább).

⟨kifejezés⟩	::=	⟨változó⟩ |	{var}
		⟨konstans⟩ |	{atomic}
		⟨összetett kifejezés⟩	{compound}
⟨konstans⟩	::=	⟨névkonstans⟩ |	{atom}
		⟨számkonstans⟩	{number}
⟨számkonstans⟩	::=	⟨egész szám⟩ |	{integer}
		⟨lebegőp. szám⟩	{float}
⟨összetett kifejezés⟩	::=	⟨struktúranév⟩ (⟨argumentum⟩, ...)	
⟨struktúranév⟩	::=	⟨névkonstans⟩	
⟨argumentum⟩	::=	⟨kifejezés⟩	

Egy Prolog kifejezés tehát lehet változó, konstans vagy összetett kifejezés. Konstans lehet névkonstans — például `alma`, `'István'` — vagy számkonstans. Ez utóbbi lehet egész vagy lebegőpontos szám, pl. `1234`, `1.234`.

Egy összetett kifejezés egy struktúranévből és egy zárójelbe tett argumentumlistából áll. Az argumentumlista egy vagy több, egymástól vesszővel elválasztott argumentumból épül fel. A struktúranév egy névkonstans, míg az argumentum (rekurzív módon) egy tetszőleges Prolog kifejezés lehet.

Egy összetett kifejezés funktorán a <struktúranév>/<argumentumok száma> szerkezetet értjük. Sokszor érdemes lehet a konstansokat 0 argumentumú összetett kifejezéseknek tekinteni. Ezért egy konstans (akár név- akár számkonstansról legyen is szó) funktora ⟨konstans⟩/0, például `alma/0`, `'István'/0` vagy éppen `12/0`. Változónak nincsen funktora.

Korábban beszéltünk már klózok funktoráról. Ez nem más, mint a klóz *fejének*, mint Prolog kifejezésnek a funktora.

Fontos, hogy amennyiben egy névkonstans nagybetűvel kezdődik aposztrófok közé kell tenni az egész konstanst azért, hogy meg lehessen különböztetni egy Prolog változótól (mint tudjuk a Prolog változók mindig nagybetűvel kezdődnek).

Lássunk egy példát összetett kifejezésre:

```
sum_tree(node(Left,Right), S)
```

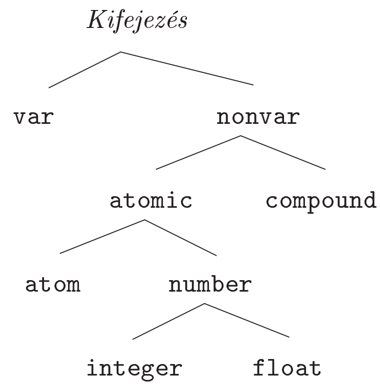
Ez egy olyan összetett kifejezés, amelynek funktora `sum_tree/2`. Figyeljük meg, hogy nincsen záró pont a kifejezés végén, hiszen most nem egy (teljes) klózról, hanem annak csak egy részéről beszélünk! A kifejezés struktúraneve `sum_tree`. Ezen összetett kifejezésnek két argumentuma van, az első szintén összetett (`node` struktúranevű, kétargumentumú), a második egy változó.

Az olvasónak feltünhet, hogy a fejezet elején példaként szereplő `sum_tree` predikátumban szerepel a `S is S1+S2` cél, amely nem illeszkedik a fenti szintaktikus leírásra. Ez egy ún. operátoros kifejezés, amelynek belső, kanonikus alakja: `is(S, +(S1,S2))`. Ez tehát egy struktúra-kifejezés, amelynek második argumentuma szintén struktúra (a `+` jel is megengedett struktúranévként). Az operátoros kifejezéseket részletesen a 3.1.5 alfejezetben ismertetjük.

Fontos hangsúlyozni, hogy a `2+4` kifejezés is egy összetett kifejezés, amelynek funktora `+ /2`, argumentumszáma 2. A Prolog relációs alapú szimbolikus nyelv, a Prolog kifejezések adatstruktúrák, vagy relációk, de semmiképpen sem függvényhívások. Így érthető, hogy a `2+4` kifejezés nem a 2 és 4 számok összegét jelenti, hanem egy olyan adatstruktúrát, amelynek neve a `+`, és két argumentuma a 2 és 4 számkonstans. Egy ilyen kifejezést szimbolikusan is feldolgozhatunk, pl. tükrözhetjük, előállítva a `4+2` struktúrát. Ugyanakkor, ha ezt a kifejezést egy aritmetikai beépített eljárás paraméterében szerepeltetjük, pl. így: `X is 2+4`, akkor a „hagyományos” aritmetikai kiértékelést végeztetjük el, és az `X` változó a 6 számkonstanst kapja értékül.

Kifejezések osztályozása

A kifejezések osztályozását szemlélteti az alábbi fa, ahol a csomópontoknak az előbbi nyelvtani táblázat jobb szélén látható angol nevek felelnek meg.



A Prolog lehetőséget nyújt arra, hogy egy adott kifejezésről eldöntsük, hogy vajon az változó-e, egész szám-e stb. Ezek az ún. osztályozó beépített eljárások:

`var(X)`	`X` változó
`nonvar(X)`	`X` nem változó
`atomic(X)`	`X` konstans
`compound(X)`	`X` struktúra
`atom(X)`	`X` atom
`number(X)`	`X` szám
`integer(X)`	`X` egész szám
`float(X)`	`X` lebegőpontos szám

Az alábbiakban ezek használatára láthatunk néhány futási példát.

```
SICStus 3.10.0 (x86-win32-nt-4): Sat Jan 11 15:04:03  2003
Licensed to BUTE DP course
| ?- atomic(korte).
yes
| ?- atomic('Korte').
yes
| ?- atomic(Korte).
no
| ?- number(korte).
no
| ?- number(12).
yes
| ?- var(12).
no
| ?- var(A).
true ? ;
no
| ?-
```

A Prolog rendszer utolsó válasza eltér a többitől, azaz se nem `yes`-t, se nem `no`-t kaptunk. Ennek oka, hogy olyan esetben, ha a feltett kérdés tartalmaz változót, akkor siker esetén a rendszer válaszában kiírja a keletkezett változóbehelyettesítéseket és felhasználói inputra vár. Jelen esetben a kérdés tartalmaz változót, ugyanakkor nem történik változóbehelyettesítés, ezért kaptuk a `true` választ. A `;` jellel újabb megoldásokat kértünk a rendszertől. A `no` válasz jelzi, hogy nincs több megoldás.

3.1.3. Prolog lexikai elemek

A legtöbb programozási nyelv szintaxisát két szinten adják meg: leírják, hogy egy program hogyan épül fel kisebb, a természetes nyelv szavainak megfelelő egységekből, azaz az ún*lexikai elemek*ből, majd megadják a lexikai elemek, mint karaktersorozatok szintaxisát. Az alábbi táblázat a Prolog nyelv lexikai elemeit tekinti át.

<table>
<tr><td>⟨változó⟩</td><td>::=</td><td>⟨nagybetű⟩⟨alfanum⟩... |
_ ⟨alfanum⟩... |</td></tr>
<tr><td>⟨névkonstans⟩</td><td>::=</td><td>'⟨névkar⟩... ' |
⟨kisbetű⟩⟨alfanum⟩... |
⟨tapadó jel⟩... | ! | ; | [] | {}</td></tr>
<tr><td>⟨névkar⟩</td><td>::=</td><td>{tetszőleges nem ' és nem \ karakter} |
\ ⟨escape szekvencia⟩</td></tr>
<tr><td>⟨alfanum⟩</td><td>::=</td><td>⟨kisbetű⟩ | ⟨nagybetű⟩ | ⟨számjegy⟩ | _</td></tr>
<tr><td>⟨tapadó jel⟩</td><td>::=</td><td>+ | - | * | / | \ | $ | ^ | < | > | = | ‘ | ~ | : | . | ? | @ | # | &</td></tr>
<tr><td>⟨egész szám⟩</td><td>::=</td><td>{előjeles vagy előjeltelen számjegysorozat}</td></tr>
<tr><td>⟨lebegőp. szám⟩</td><td>::=</td><td>{belsejében tizedespontot tartalmazó
számjegysorozat esetleges exponenssel}</td></tr>
</table>

Látszik, hogy a változók mindig nagybetűvel (vagy aláhúzással) kezdődnek, hogy a konstansok vagy kisbetűvel kezdődnek vagy aposztrófok között vannak vagy tapadójelek egymásutánjaiból állnak stb.

Álljon itt néhány példa arra, hogy mik megengedett kifejezések Prologban. Megengedett változónév a `Fakt`, a `FAKT`, a `_fakt`, a `X2`, a `_2` vagy akár a `_` is. Szintaktikailag helyes névkonstansok a `fakt`, a ≡, a `'fakt'`, az `'István'`, a `[]`, a `**`, a `\=` stb. Megengedett számkonstans a `0`, `-123`, `10.0`, `-12.1e8`.

3.1.4. Prolog típusok

A Prolog típustalan nyelv, ennek ellenére a Prolog programozó is többnyire úgy gondolja, hogy eljárásának egy adott paraméter-pozícióján csak bizonyos Prolog kifejezések szerepelhetnek. Így például a `sum_tree/2` eljárás első paramétere bináris fa, a második egész szám lehet. Egy adott paraméter-pozíción értelmes *tömör* Prolog kifejezések halmazát nevezhetjük típusnak. A „tömör” szó itt változómentes kifejezésre utal, tehát olyan Prolog kifejezésre, amelyben nem szerepel (behelyettesítetlen) változó. Ha például az egész számok típusát (azaz halmazát)_ az `integer` szóval jelöljük, akkor a `sum_tree/2` eljárás által kezelt egész-levelű bináris fák `tree` típusa a következő halmazegyenlettel definiálható:

$$\texttt{tree} \equiv \{\ \texttt{leaf}(i)\ \texttt{|}\ i \in \texttt{number}\ \} \cup \{\ \texttt{node}(l,r)\ \texttt{|}\ l,r \in \texttt{tree}\ \}$$

Fontos látni, hogy Prologban általában nincsen típushiba, csak meghiúsulás. Azaz például, ha egy `n(l(4),l(5))` kifejezést adunk a `sum_tree/2` eljárásnak, az nem jelez hibát, csak meghiúsul, hiszen a (az első paraméter miatt) a hívás egyik klózfejre sem illeszthető:

```
| ?- sum_tree(n(l(4),l(5)),S).
no
| ?-
```

Hibajelzést kaphatunk azonban akkor, ha egy beépített eljárást nem megfelelő típusú paraméterrel hívunk, pl.:

```
| ?- X is 1+alma.
! Domain error in argument 2 of is/2
```

```
! expected expression, found alma
! goal:  _79 is 1+alma
| ?-
```

Így a típus a Prolog nyelvben *implicit* módon jelenik csak meg: adatábrázolási döntésünk csak az adatokat felhasználó eljárások alakjában tükröződik. (Vannak típusos logikai programozási nyelvek, pl. a Mercury nyelv, amelyekben típusdeklarációkkal kell leírnunk a használt adatstruktúrákat.) A Prolog jó tulajdonsága, hogy ha nem végzünk egy klózban olyan műveleteket, amelyek feltételeznek valamilyen adattípust, akkor a klóz egy *generikus* állítást fogalmaz meg. Ez kifejezetten alkalmassá teszi a Prologot szimbolikus számítások elvégzésére.

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

Ebben a jegyzetben a típusdeklarációk alakjára a Mercury nyelvhez közeli szintaxist használjuk. Mivel a használt Prolog rendszerek (SICStus, SWI) nem értelmezik a típusdeklarációkat, ezert a deklarációkat kommentbe kell/érdemes ágyazzuk. A predikátumtípusokat célszerű a predikátum fejkommentjében megadni. Hangsúlyozzuk mégegyszer, hogy mivel a Prolog rendszerek nem ismerik a típusokat, ezért a típusinformáció megadása egyáltalán nem kötelező, ellenben nagyon javasolt annak érdekében, hogy a kód könnyen olvasható és áttekinthető legyen.

Az alábbiakban a `sum_tree/2` eljárás által kezelt adatstruktúra-típust írjuk fel, a javasolni használt formális jelöléssel:

```
% :- type tree == {leaf(integer)} \/ {node(tree, tree)}.
```

Az első jelölés a fenti halmaz-alakhoz hasonló. Így pl. `{leaf(integer)}` mindazon struktúrák halmazát jelöli, amelyek funktora `leaf/1` (azaz nevük `leaf` és egy argumentumuk van), és amelyek egyetlen argumentuma egész (azaz az `integer` halmazból való). Hasonlóan a `{node(tree, tree)` azon `node/2` funktorú struktúrák halmazát jelöli, amelyek mindkét argumentuma `tree` típusú. A `\/` jellel a két oldalán álló halmaz *unióját* képezzük. A fenti típusdeklaráció tehát egy rekurzív halmazegyenlet, amelynek *legszűkebb* megoldását tekintjük. Így a `tree` típus, az a legszűkebb halmaz, amely egyrészt tartalmazza az összes `leaf(`i`)` alakú kifejezést (i egész), másrészt, bármely két t_1, t_2 ∈`tree` esetén a `node(`t_1`, `t_2`)` kifejezést is.

A fenti példa egy ún. **megkülöböztetett únió** (discriminated union). Egy megkülönböztetett únió véges sok, különböző funktorú típusból képzett halmaz úniója. Úgy érdemes elképzelni, mint egy olyan C `union` típust, ahol még azt is nyilvántartjuk, hogy éppen melyik mező az érvényes. Jelen esetben a `leaf/1` és a `node/2` a különböző funktorok.

A Mercury nyelvben csak megkülönböztetett úniót szabad használni, erre ott bevezettek egy speciális, a fentinél egyszerűbb jelölést:

```
% :- type tree ---> leaf(integer) | node(tree, tree).
```

A Prolog nyelvben (szemben pl. a Mercury, SML nyelvekkel) megengedett a nem-megkülönböztetett únió. Erre példa az alábbi típus.

```
% :- type tree2 == integer \/ {node(tree2, tree2)}.
```

Ez tehát abban különbözik a fenti, `tree` fatípustól, hogy a leveleket nem „csomagoljuk” be egy `leaf/1` funktorú struktúrába. Ez az adattípus könnyebben olvasható, hiszen így tömörebben írható le egy fa. Például `node(leaf(2),node(leaf(3),leaf(1)))` helyett írhatjuk a `node(2,node(3,1))` kifejezést.

Most egy ilyen adattípuson dolgozó Prolog `sum_tree2` eljárást fogunk bemutatni. A működés jobb megértéséhez azonban szükséges, ha előtte az eredeti `sum_tree` predikátumot egy kicsit átalakítjuk:

```
sum_tree(Tree, S) :-
```

```
    Tree = leaf(Value),
    S = Value.
sum_tree(Tree, S) :-
    Tree = node(Left,Right),
    sum_tree2(Left, S1),
    sum_tree2(Right, S2),
    S is S1+S2.
```

Az első klóz továbbra is egy levél levélösszegét számítja ki. Bár első fejargumentumára illeszkedik egy tetszőleges fa is, a törzs első hívása megvizsgálja, hogy egy `leaf/1` funktorú kifejezésről van-e szó. Ugyanígy, a második klóz továbbra is csomópontokkal dolgozik, bár csak a klóz fejét nézve akár azt is hihetnénk, hogy levéllel is elboldogul. Itt is az a „trükk”, hogy a klóz törzseiben az első cél csak akkor teljesül, ha a megfelelő típusú adat érkezett.

A `sum_tree2` predikátum ezek után úgy képezhető, hogy az első klóz első célját az `integer` osztályozó eljárás meghívására cseréljük:

```
sum_tree2(Tree, S) :-
    integer(Tree),
    S = Tree.
sum_tree2(Tree, S) :-
    Tree = node(Left,Right),
    sum_tree2(Left, S1),
    sum_tree2(Right, S2),
    S is S1+S2.
```

Végül következzék a `sum_tree2` eljárás példafutása. Azt is megmutatjuk, hogy ez az eljárás visszafelé még kevésbé használható, mint a megkülönböztetett úniót használó `sum_tree`, hiszen az előbbi a `sum_tree2(T, 10)` hívás esetén végtelen ciklusba esik, míg az utóbbi legalább egy eredményt tud adni.

```
| ?- sum_tree2(node(5,node(3,2)),S).
S = 10 ? ;
no
| ?- sum_tree2(T, 10).
Prolog interruption (h for help)? a
% Execution aborted
| ?- sum_tree(T, 10).
T = leaf(10) ?
yes
| ?-
```

3.1.5. Operátorok

A levélösszeg számító példákban láttunk egy `S is S1+S2` célt a második klózban. Már korábban is utaltunk arra, hogy ez egy operátoros kifejezés. Itt az `is` és a `+` névkonstansok ún. *operátorok*, amelyek hívások ill. adatstruktúrák infix jelöléssel való írását teszik lehetővé. Az operátorok „szintaktikus édesítőszerek”, mert a kifejezés beolvasását követően eltűnnek, a rendszeren belül a kifejezés szabványos alakú lesz. A fenti példa esetén ez az `is(S, +(S1,S2))` alak, tehát az `is/2` eljárás meghívásáról van szó, amelynek második argumentuma egy `+` nevű két-argumentumú rekord-struktúra. Hasonlóképpen a `S1+S2>1000` hívás szabványos alakja: `>(+(S1,S2),1000)`.

Ennek megfelelően az alábbi két Prolog kérdés teljesen ekvivalens egymással — láthatóan a rendszer válasza is megegyezik.

```
| ?- A is 4+2.
A = 6 ? ;
```

```
no
| ?- is(A,4+2).
A = 6 ? ;
no
| ?-
```

Az `is/2`, `>/2`, és a többi aritmetikai beépített eljárás különlegesen kezeli az argumentumait: a bennük szereplő (akár többszörösen is) összetett adatstruktúrákat aritmetikai kifejezésnek tekintik, és ki is értékelik. Nagyon fontos megértenünk (erről már volt szó korábban), hogy Prologban a `4+2` egy közönséges összetett kifejezés és nem `6`. A beépített aritmetikai eljárások azonban képesek aritmetikai kifejezésekkel dolgozni, ezért lehetséges, hogy a fenti példában a Prolog rendszer válasza `6`.

A Prolog programozó definiálhat új operátort a programkódban elhelyezett ún. *operátor-deklaráció* segítségével. Ennek alakja:

`:- op(⟨prioritás⟩, ⟨fajta⟩, ⟨operátornév⟩).`

Az aritmetikai operátorok mind beépítettek, azaz például a SICStus rendszer indulásakor már létezik a `+` nevű operátor adott prioritással és fajtával. Ha nem lennének beépített operátorok, akkor nem írhattunk volna le semelyik eddigi példánkban sem olyat, hogy `S1 + S2`. Helyette az `+(S1,S2)` alakot lettünk volna csak képesek használni.g

Az ⟨operátornév⟩ tetszőleges névkonstans lehet, bár többnyire csak az aposztróf-jel nélkül írható alakok használatosak, azaz írásjelek és ezek sorozatai, ill. alfanumerikus névkonstansok. Megjegyezzük, hogy egy operátor-deklarációban több (azonos prioritású és fajtájú) operátor is létrehozható. Ilyenkor az operátorneveket vesszővel elválasztva és szögletes zárójelbe téve kell a deklaráció harmadik argumentumában megadni, például így:

```
:- op(500, xfx, [alma, körte]).
```

A ⟨prioritás⟩ egy egész szám 1 és 1200 között, amely a több operátort tartalmazó kifejezések zárójelezési sorrendjét határozza meg: a kisebb prioritású operátorok előbb zárójeleződnek mint a nagyobbak. A ⟨fajta⟩ jellemző azt határozza meg, hogy az azonos prioritású operátorok hogyan zárójeleződjenek (pl. a `+` operátor balról-jobbra, míg a `^` jobbról-balra zárójeleződik). A fajta az `xfy`, `yfx`, `xfx`, `fx`, `fy`, `xf`, `yf`névkonstansok egyike, Itt az „`f`” szemlélteti magát az operátort, a tőle balra ill. jobbra álló betű a bal- ill. jobboldali operandust. Az „`x`” és „`y`” betűk jelentése:

- `x`: az adott oldalon nem állhat azonos prioritású operátor zárójelezetlenül
- `y`: az adott oldalon állhat azonos prioritású operátor zárójelezetlenül

Infix operátorok esetén a ⟨fajta⟩ lehet `yfx`, amely balról jobbra való zárójelezést ír elő (ilyen a `+`, `-`, stb.), `xfy`, amely jobbról balra zárójeleződik, vagy `xfx` (pl. a `<`), amely nem engedi az azonos prioritású operátor (zárójelezetlen) használatát egyik oldalán sem.

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

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

Megjegyezzük, hogy a Prolog klózban használt összekötő jelek, így a `:-`, a vessző mind szabványos operátorok, és így maga a klóz is egy Prolog kifejezésként olvasódik be. Ezért van az, hogy az operátor-jelölés használható nemcsak a struktúra-kifejezésekben hanem az eljáráshívásokban is (pl. `is/2`). Azt a tényt, hogy a klózok kifejezésként is felírhatók, az ún. *adatbázis-kezelő* beépített eljárások (ld. 4.11) is kihasználják.

Beépített operátorok

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

3.1. táblázat. A BEÉPÍTETT SZABVÁNYOS OPERÁTOROK

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

3.2. táblázat. EGYÉB BEÉPÍTETT OPERÁTOROK

A 3.1 és a 3.2 táblázat a SICStus Prolog beépített operátordefinícióit adja meg. Itt jegyezzük meg, hogy a vessző jel (`,`) több értelemben is szerepel a Prolog szintaxisában: egyrészt mint 1000-s prioritású operátor, másrészt pedig mint az argumentumokat elválasztó jel. Ezért az argumentum-listában zárójelezés nélkül legfeljebb 999-es prioritású operátorok fordulhatnak elő, az ennél nagyobb prioritású operátort tartalmazó argumentum-kifejezést zárójelezni kell.

Az operátorok felhasználása

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

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

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

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

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

Harmadszor, operátorok segítségével a Prolog programok természetesebbé, olvashatóbbá tehetők. Például a bevezető fejezetben ismertetett nagyszülője predikátum a következő alakba írható át (fontos értenünk, hogy ez csak számunkra olvashatóbb alak, a Prolog rendszer úgyis szabványos alakra hoz mindent, ott nincsenek operátorok):

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

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

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

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

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

sav(kén,h.2-s-o.4).
```

Végezetül az operátorok teszik lehetővé a „klasszikus" szimbolikus kifejezésfeldolgozást, például a szimbolikus deriválást.

Rossz tulajdonságai is vannak az operátoroknak. Az operátorok nem lokálisak az adott Prolog modulra nézve, ezért egy nagyobb projektben gondot jelenthetnek más emberek által megírt vagy átdefiniált operátorok számunkra (például tudtunk nélkül egyik napról a másikra egy általunk használt operátor priorítását valaki más megváltoztathatja).

Bináris fa — operátoros változat

Ebben a szakaszban bemutatjuk az operátorok alkalmazását a jól ismert binárisfa-példánk egyszerűbb, nem-megkülönböztetett úniót használó változatában. Definiálunk egy `--` nevű `xfx` típusú operátort:

```
:- op(500, xfx, --).
```

Ez a - név fogja helyettesíteni az eddigi `node` struktúranevet. Mivel azonban a `--` operátort infixnek deklaráltuk sokkal kényelmesebben írhatunk le egy fát segítségével:

```
5--(3--2)
```

Ez megfelel a

```
--(5,--(3,2))
```

szabványos alaknak, amikoris ha `--` helyére `node`-t írunk, visszajutunk az régebbi alakunkra. Az operátorral tehát csak annyit nyertünk, hogy könnyebben olvashatóbbá tettük a fánk leírását, valamint a kódunkat is:

```
sum_tree3(Left--Right, S) :-
        sum_tree3(Left, S1),
        sum_tree3(Right, S2),
        S is S1+S2.
sum_tree3(Tree, S) :-
        integer(Tree),
        S = Tree.
```

3.2. Prolog szemantika

Ez az alfejezet a Prolog nyelv szemantikáját ismerteti. Az eddigiekben megismerkedtünk a Prolog nyelv felépítésével, láttunk Prolog programot. Nem esett sok szó azonban arról, hogy valójában hogyan történik egy kérdés megválaszolása? Van-e a C-hez hasonlóan egyfajta „futása” egy Prolog programnak, van-e belépési pontja stb.?

Először bemutatjuk a Prolog deklaratív, majd a procedurális szemantikáját. Ismertetjük ezután a Prolog egyesítési algoritmusát, majd a redukciós lépés fogalmát. Megismerkedünk a Prolog végrehajtási algoritmusával, mely keretében bemutatjuk a visszalépéses keresés elvét. Végül rátérünk a Prolog nyelv vezérlési szerkezeteinek ismertetésére.

3.2.1. A Prolog deklaratív szemantikája

A második fejezetben már szó volt róla, hogy egy Prolog programban minden klóz egy elsőrendű logikai állításnak felel meg. Elevenítsük fel egy kicsit az ott elhangzottakat! Láttuk, hogy a

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

predikátum pontosan megfelel az alábbi logikai formulának (a változónevektől eltekintve):

$$\forall U \forall N \forall Sz(\texttt{nagyszülője}(Gy, N) \leftarrow \texttt{szülője}(Gy, Sz) \wedge \texttt{szülője}(Sz, N))$$

Egy Prolog programot nem futtathatunk például egy C kód esetében szokásos módon. Egy Prolog program predikátumok halmaza, mi ezekre vonatkozó kérdéseket tehetünk fel. A feltett kérdés, más néven `célsorozat` vesszőkkel elválasztott Prolog célok sorozata. A célsorozatnak egy bizonyítandó állításnak felel meg. Például, az a kérdés, hogy kik a nagyszülei Imrének...

```
| ?- nagyszuloje('Imre', N).
```

...megfelel az alábbi logikai formulának:

$$\exists N(\texttt{nagyszuloje('Imre'}, N))$$

Itt tehát azt kérdezzük, hogy van-e olyan behelyettesítése az `N` változónak, amely esetén a célsorozat logikai következménye lesz a programunknak. A Prolog rendszer egy ilyen behelyettesítéssel válaszol (`N = 'Géza'`), és az összes ilyen behelyettesítést hajlandó felsorolni.

Sajnos a deklaratív szemantika nem elegendő ahhoz, hogy működőképes Prolog programokat írjunk, hiszen a Prolog rendszer egy speciális, nagyon egyszerű következtetési algoritmust használ, amellyel nem biztos, hogy véges időn belül el lehet állitani az összes következményt.

Ennek ellenére a deklaratív szemantika megléte nagyon fontos, hiszen azt garantálja, hogy a Prolog által szolgáltatott eredmény biztosan logikai következménye programunknak.

3.2.2. A Prolog procedurális szemantikája, a végrehajtási algoritmus

A procedurális szemantika valószínűleg közelebb áll a imperatív nyelvekben jártas olvasókhoz, mint az előző részben látott deklaratív megközelítés.

A procedurális szemantika (a Prolog *végrehajtási algoritmusa*) egy adott Prolog programra vonatkozó kérdés esetén megadja a kérdés végrehajtásának pontos leírását. Ez egy nagyon leegyszerűsített tételbizonyítási algoritmuson, az ún. SLD rezolúción (Linear resolution on Definite clauses with Selection function) alapul. A végrehajtási algoritmus két pilléren nyugszik. Az egyik egy *mintaillesztésen (egyesítésen)* alapuló *eljáráshívási mechanizmus*. Ezen mechanizmus alaplépése az ún. *redukciós lépés*, amely bemeneteként adott egy célsorozat és egy klóz. A redukciós lépés keretében megpróbáljuk egyesíteni a klóz fejét a célsorozat legelső céljával. Siker esetén a klóz törzsét az első cél helyébe rakjuk.

Első közelítésben definiáljuk két Prolog kifejezés *egyesítését* a következőképpen: két kifejezés egyesíthető, ha a bennük levő változók helyébe tetszőleges Prolog kifejezéseket helyettesítve a két kifejezés azonossá tehető. A behelyettesítés szisztematikus, tehát ha egy változó többször is előfordul, akkor minden előfordulását azonos kifejezésre kell cserélni.

A végrehajtási algoritmus másik pillére a *visszalépéses mélységi keresés*, amelyről később lesz szó.

Prolog végrehajtási példa

Tekintsük a levélösszeget kiszámító Prolog program operátoros változatát és a program végrehajtásának lépéseit a `sum_tree3(3--2, A), write(A)` célsorozat esetén! Programunk két klózának sorrendjét felcseréltük, de ez nem változtat a jelentésén. A `write/1` beépített eljárás az argumentumában kapott Prolog kifejezést írja ki (alaphelyzetben a képernyőre).

```
sum_tree3(Left--Right, S) :-                % (1)
        sum_tree3(Left, S1),
        sum_tree3(Right, S2),
        S is S1+S2.
sum_tree3(Tree, S) :-                       % (2)
        integer(Tree),
        S = Tree.
```

A végrehajtás során a kezdeti célsorozat első hívását (`sum_tree3(3--2, A)`) sikeresen egyesíti a Prolog rendszer a `sum_tree/3` predikátum első klózával. A keletkezett változóbehelyettesítések a következők: `Left=3`, `Right=2` és `S=A`. A redukciós lépést a rendszer végrehajtja az első klóz és a célsorozat első hívására, azaz a célsorozat első hívását kicseréli a klóz törzsére. Az új célsorozat a következő (a sor elején álló `(1)` jelzi, hogy melyik klózzal hajtottunk végre redukciós lépést):

```
(1) > sum_tree3(3,B), sum_tree3(2,C), A is B+C, write(A)
```

Ezek után az új célsorozat első hívását próbáljuk meg egyesíteni valamelyik klózzal a kettő közül. Ez nem sikerül az elsővel, mert a `sum_tree3(3,B)` hívás fejében lévő, első argumentumhelyen szereplő `3` nem egyesíthető a klóz első argumentumával, a `Left-Right` struktúrával. A második klózzal azonban sikeresen egyesíthető a hívás. A változóbehelyettesítések és a redukciós lépés elvégzése után az új célsorozat:

```
(2) > integer(3), B=3, sum_tree3(2,C), A is B+C, write(A)
```

A következő redukciós lépés egy beépített eljárással történik. Ezt úgy is tekinthetjük, hogy a célsorozatban legelől álló `integer(3)` hívás egy üres törzsű klózra illeszkedik. Jelen esetben változóbehelyettesítés sem történik, az `integer(3)` hívás egyszerűen minden további nélkül sikerül. Általában egy beépített hívás meg is hiúsulhat (pl. `integer(alma)`), ill. siker esetén változóbehelyettesítéseket is előállíthat.

A legutóbbi redukciós lépés után megmaradó célsorozat a következő — itt a sor elei `BIP` szöveg arra utal, hogy beépített predikátummal (Built-In Predicate) végzett redukció eredményeként állt elő ez a célsorozat:

```
BIP > B=3, sum_tree3(2,C), A is B+C, write(A)
```

Ezután újból egy beépített eljáráshívás[1] a célsorozat első eleme és így tovább. A kialakult célsorozatokat láthatjuk alább:

```
BIP > sum_tree3(2,C), A is 3+C, write(A)
(2) > integer(2), C=2, A is 3+C, write(A)
```

[1]Az `X = Y` beépítetteljárás-hívás két argumentumát egyesíti.

```
BIP > C=2, A is 3+C, write(A)
BIP > A is 3+2, write(A)
BIP > write(5)                          {===> 5}
BIP > []
```

A végső állapot az, amikor elfogy a célsorozat vagyis üres célsorozatot kapunk.

Egyesítés és behelyettesítés fogalma

Az *egyesítés* központi szerepet játszik a redukciós lépésben. Láttuk, hogy a redukciós lépés egy klóz fejét próbálja meg egyesíteni egy hívással, ami nem más, mint egy tetszőleges Prolog kifejezés. Siker esetén a klóz törzsére cseréljük a hívást a célsorozatban. Sikertelenség esetén a redukciós lépés meghiúsul. Fontos tehát az egyesíthetőség kérdésének eldöntése.

Lássunk néhány példát! Praktikusan egy hívás mindig egy adott célsorozat legelső elemét jelenti. Egy célsorozat természetesen lehet egyelemű is: egy kérdés állhat egyetlen Prolog kifejezésből is. A fej mindig egy Prolog szabály fejét vagy egy tényállítást jelöl, amelyet valamilyen (számunkra most még irreleváns) módon választ ki a Prolog végrehajtó a lehetséges klózok közül.

- Bemenő paraméterátadás (a fej változóo kapnak értéket):
  hívás: `nagyszuloje('Imre', Nsz)`,
  fej: `nagyszuloje(Gy, N)`,
  behelyettesítés: `Gy = 'Imre', N = Nsz`
- Kimenő paraméterátadás (a hívás változói kapnak értéket):
  hívás: `szuloje('Imre', Sz)`,
  fej: `szuloje('Imre', 'István')`,
  behelyettesítés: `Sz = 'István'`
- Bemenő/kimenő paraméterátadás (mind a fe, mind a hívás változói kapnak értéket):
  hívás: `sum_tree(leaf(5), Sum)`
  fej: `sum_tree(leaf(V), V)`
  behelyettesítés: `V = 5, Sum = 5`

Térjünk most rá a *behelyettesítés* fogalmának precíz matematikai megfogalmazására. A behelyettesítés egy függvény, amely változókhoz kifejezéseket rendel. Például a

$$\sigma = \{X \leftarrow a, Y \leftarrow s(b,B), Z \leftarrow C\}$$

behelyettesítés `X`-hez `a`-t, `Y`-hoz `s(b,B)`-t stb. rendeli. $K\sigma$-val jelöljük σ alkalmazását K kifejezésre. Például `f(g(Z,h),A,Y)`σ =`f(g(C,h),A,s(b,B))`, mert a jobboldal úgy áll elő, hogy az adott Prolog kifejezésen elvégezzük a megadott behelyettesítéseket (jelen esetben a `Z=C`-t és az `Y=s(b,B)`-t).

Definiálhatjuk két behelyettesítés kompozícióját az alábbi módon (ez megfelel a szokásos függvénykompozíciónak):

$$\sigma \otimes \theta = \{\ x \leftarrow x\sigma\theta \mid x \in D(\sigma)\} \bigcup \{\ x \leftarrow x\theta \mid x \in D(\theta) \setminus D(\sigma)\ \}$$

Behelyettesítések kompozíciója természetesen egy újabb behelyettesítést eredményez. A fenti sor azt írja le, hogy a kompozíció kétféle behelyettesítésből fog állni:

- (az $\bigcup$ elötti rész): mindazon x változókat, amelyek szerepelnek σ értelmezési tartományában, a $x\sigma\theta$ kifejezésre kell helyettesíteni, azaz a σ által előírt helyettesítő értéken még végre kell hajtani a θ behelyettesítést
- (az $\bigcup$ utáni rész): minden olyan változóhoz, amelyre csak a θ behelyettesítés értelmezett azt a kifejezést rendeli, amelyet a θ rendelne önmagában.

Végül definiálhatjuk, hogy mikor általánosabb egy behelyettesítés egy másiknál: σ általánosabb mint θ, ha létezik olyan ρ, hogy $\theta = \sigma \otimes \rho$. Ezen definíció lehetőséget ad arra, hogy definiáljuk a ***legáltalánosabb egyesítő*** (mgu — most general unifier) fogalmát.

A és B kifejezések egyesíthetőek ha létezik egy olyan σ behelyettesítés, hogy $A\sigma = B\sigma$. Ezt a σ behelyettesítést A és B egyesítőjének nevezzük. A és B legáltalánosabb egyesítője σ ($mgu(A, B) = \sigma$), ha σ A és B minden egyesítőjénél általánosabb. Megmutatható, hogy változó-átnevezéstől eltekintve az mgu egyértelmű.

Az egyesítési algoritmus

Az egyesítési algoritmus bemenete két Prolog kifejezés: A és B. Az algoritmus feladata a két kifejezés egyesíthetőségének eldöntése. Ha a két kifejezés egyesíthető, akkor az algoritmus előállítja a legáltalánosabb egyesítőt ($mgu(A, B)$) is.

Az alábbiakban ismertetjük a Prolog egyesítési algoritmusát:

1. Ha A és B azonos változók vagy konstansok, akkor az egyesítés sikeres és $\sigma = \{\}$ (üres behelyettesítés, nem változtat semmit).
2. Egyébként, ha A változó, akkor az egyesítés sikeres és $\sigma = \{A \leftarrow B\}$.
3. Egyébként, ha B változó, akkor az egyesítés sikeres és $\sigma = \{B \leftarrow A\}$.
4. Egyébként, ha A és B azonos nevű és argumentumszámú összetett kifejezések és argumentum-listáik A_1,. . . ,A_N ill. B_1,. . . ,B_N, és
   (a) A_1 és B_1 legáltalánosabb egyesítője σ_1,
   (b) $A_2\sigma_1$ és $B_2\sigma_1$ legáltalánosabb egyesítője σ_2,
   (c) $A_3\sigma_1\sigma_2$ és $B_3\sigma_1\sigma_2$ legáltalánosabb egyesítője σ_3,
   (d) . . .

   akkor az egyesítés sikeres és $\sigma = \sigma_1 \otimes \sigma_2 \otimes \sigma_3 \otimes \ldots$.
5. Minden más esetben a A és B nem egyesíthető.

Az algoritmus működésének jobb megértésének érdekében lássunk néhány példát! Legyen A `= sum_tree(leaf(V), V)` és B `= sum_tree(leaf(5), S)`. Kérdés, hogy hogyan fut le ezen két kifejezés esetén az egyesítési algoritmus:

- (4.) A és B neve és argumentumszáma megegyezik
  - (a.) `mgu(leaf(V), leaf(5))` (4., majd 2. szerint) = {`V`←`5`} = σ_1
  - (b.) `mgu(V`σ_1`, S) = mgu(5, S)` (3. szerint) = {`S`←`5`} = σ_2

Azt kaptuk tehát, hogy $mgu(A, B) = \sigma_1 \otimes \sigma_2$ = {`V`←`5, S`←`5`}.

Másik példánkban legyen A `= node(leaf(X), T)`, valamint B `= node(T, leaf(3))`. Ekkor:

- (4.) A és B neve és argumentumszáma megegyezik
  - (a.) `mgu(leaf(X), T)` (3. szerint) = {`T`←`leaf(X)`} = σ_1
  - (b.) `mgu(T`σ_1`, leaf(3)) = mgu(leaf(X), leaf(3))` (4, majd 2. szerint) = {`X`←`3`} = σ_2

Azaz $mgu(A, B) = \sigma_1 \otimes \sigma_2$ = {`T`←`leaf(3), X`←`3`}

Az = /2 beépített eljárás

Az `A = B` beépített eljáráshívás akkor és csak akkor sikerül ha a két argumentuma egyesíthető. Az eljárás el is végzi az egyesíthetőséghez szükséges behelyettesítéseket. Például:

```
| ?- f(X) = f(3).
X = 3 ? ;
no
| ?-
```

Itt az `f(X)` és az `f(3)` Prolog kifejezések (mindkettő összetett) egyesítése sikeres és a legáltalánosabb egyesítő a $\{X \leftarrow 3\}$.

Az `= /2` eljárás definíciója `X = X`, azaz nem más, mint a következő Prolog tényállítás:

```
=(X,X).
```

A `=` „mellesleg” beépített operátor, ezért használhatjuk infix pozícióban is.[2]

Lássunk néhány további példát a `=/2` eljárás használatára!

```
| ?- 3--(4--5) = Left--Right.              (1)
        Left = 3, Right = 4--5 ?
| ?- node(leaf(X), T) = node(T, leaf(3)).  (2)
        T = leaf(3), X = 3 ?
| ?- X*Y = 1+2*3.                          (3)
        no
| ?- f(X, 3/Y-X, Y) = f(U, B-a, 3).        (4)
        B = 3/3, U = a, X = a, Y = 3 ?
| ?- f(f(X), U+2*2) = f(U, f(3)+Z).        (5)
        U = f(3), X = 3, Z = 2*2 ?
```

Az (1) esetben az egyesítési algoritmus 4., majd kétszer a 3. lépése hajtódik végre. A (2) eseteben a sorrend: 4, 3, 4, 2. A (3) példában azért hiúsul meg az egyesítés, mert a baloldal kanonikus alakja `*(X,Y)`, míg a jobboldalé `+(1,*(2,3))`, és így az egyesítési algoritmus már a legkülső struktúrák egyesítésénél meghiúsul. A többi példa elemzését az olvasóra bízzuk.

Előfordulás-ellenőrzés

Az egyesíthetőség kapcsán felmerül az a fontos kérdés, hogy vajon például `X` és `s(X)` egyesíthető-e?

Matematikailag a válasz egyértelműen *nem*, mert egy változó nem egyesíthető egy olyan struktúrával, amelyben az előfordul. Ez tehát azt jelenti, hogy mielőtt egy behelyettesítést elvégeznénk (az egyesítési algoritmus 2. ill. 3. lépésében) elvben meg kell vizsgálnunk, hogy a behelyettesíteni kívánt változó előfordul-e a helyettesítő kifejezésben. Ezt a vizsgálatot hívjuk *előfordulás-ellenőrzésnek* (occurs check). Mivel ez egy költséges vizsgálat, ezért a Prolog egyesítési algoritmusa ezt nem tartalmazza. Ennek következtében viszont, ha egy változót egy olyan kifejezéssel helyettesítünk, amelyben ő előfordul, akkor egy végtelen rekurzív kifejezésfa jön létre:

```
| ?- X=s(X).
X = s(s(s(s(s(s(s(s(s(...))))))))) ? ;
no
| ?- X=s(1,X).
X = s(1,s(1,s(1,s(1,s(1,s(1,s(1,s(1,s(1,s(...)))))))))) ? ;
no
| ?-
```

[2]Talán furcsa lehet elsőre, hogy Prologban még az egyenlőség is egy ugyanolyan eljárás, mint amit a programozó írhat.

A Prolog szabvány sem követeli meg az előfordulás-ellenőrzést, így a legtöbb Prolog rendszer sem alkalmazza. Szabványos eljárásként azonban rendelkezésre áll a `unify_with_occurs_check/2`, amely az `=/2` előfordulás-ellenőrzéssel kiegészített változata.

```
| ?- unify_with_occurs_check(X,s(X)).
no
| ?- unify_with_occurs_check(X,s(1,X)).
no
| ?-
```

A SICStus Prolog rendszerben sincsen előfordulás-ellenőrzés, de a SICStus képes az előfordulás-ellenőrzés elhagyása miatt keletkező ciklikus kifejezések véges idejű egyesítésére:

```
| ?- X = s(X), Y = s(s(Y)), X = Y.
        X = s(s(s(s(s(...))))), Y = s(s(s(s(...))))) ?
```

A SICStus tehát képes arra, hogy a megadott célsorozat harmadik tagjaként szereplő egyesítést elvégezze, annak ellenére hogy ott két különbözőképpen képzett, de azonos szerkezetű végtelen fa szerepel. Más Prolog rendszerek sokszor végtelen rekurzióba esnek a fenti egyesítés végrehajtásakor.

Redukciós lépés

Ebben a részben megadjuk a redukciós lépés pontos definícióját.

A redukciós lépés egy célsorozatot és egy klózt kap bemenetéként. A lépés lehet sikeres, és ekkor egy újabb célsorozatot állít elő, vagy sikertelen. Müködését az alábbi módon foglalhatjuk össze:

- A klózt lemásoljuk, minden változót szisztematikusan új változóra cserélve.
- A célsorozatot szétbontjuk az első hívásra és a maradékra.
- Az első hívást egyesítjük a klózfejjel.
- A szükséges behelyettesítéseket elvégezzük a klóz törzsén és a célsorozatot maradékán.
- Az új célsorozat: a klóztörzs és utána a maradék célsorozat.
- Ha a hívás és a klózfej nem egyesíthető, akkor a redukciós lépés meghiúsul.

Láthatjuk, hogy a redukciós lépésben először elkészítjük a klóz egy másolatát, úgy, hogy a benne szereplő összes változónevet szisztematikusan a célsorozatban nem szereplő változónevekre cseréljük. A változócsere a klózok ismételt felhasználásához fontos, hiszen egy újabb eljáráshíváskor (pl. rekurzió estén) a célsorozatba bekerülő változókat a korábbiaktól különbözőnek kell tekinteni.

Ezek után megpróbáljuk a célsorozat (az általunk feltett kérdés) legelső hívását *egyesíteni* a megadott klózfejjel — erre kell az egyesítési algoritmus. [3] Siker esetén a keletkezett változóbehelyettesítéseket elvégezzük a klóz törzsében és a maradék célsorozaton, majd a törzset rakjuk az első hívás helyébe, kialakítva így egy új célsorozatot.

A Prolog végrehajtási algoritmusa

A Prolog végrehajtási algoritmusa nem más, mint egy adott célsorozat futása egy adott Prolog programra vonatkozóan. A végrehajtási algortmus eredménye lehet siker (ilyenkor változóbehelyettesítés is történhet) vagy meghiúsulás. Utóbbi esetben semmilyen körülmények között nem történik változóbehelyettesítés. Azaz például, ha adott egy 5 hívásból álló célsorozat (öt darab, egymástól vesszővel elválasztott Prolog kifejezés,

[3] A redukciós lépés bemeneteként megadott klóz mindig a célsorozat első hívásának megfelelő Prolog predikátum valamelyik klóza. Azt, hogy ez melyik, a végrehajtási algoritmus dönti el.

kérdésként feltéve) és az 5. hívás nem tud sikeresen lefutni, akkor hiába történnének bizonyos változóbehelyettesítések az első négy hívás kapcsán, ezeknek nem lesz látható hatásuk.
A Prolog végrehajtási algoritmusa a következő:

1. Ha a célsorozat első hívása beépített eljárásra vonatkozik:
   - hajtsuk végre a hívást
   - ez lehet sikeres (változó-behelyettesítésekkel) vagy sikertelen
   - siker esetén a behelyettesítéseket elvégezzük a célsorozatot maradékán.
   - az új célsorozat: az első hívás elhagyása után fennmaradó maradék célsorozat.
   - ha a beépített eljárás hívása sikertelen, *visszalépés* következik
2. Ha a célsorozat első hívása felhasználói eljárásra vonatkozik:
   - megkeressük az eljárás első (visszalépés után: következő) olyan klózát, melyre a redukciós lépés sikeresen lefut.
   - Ha nincs ilyen klóz akkor *visszalépés* következik
3. Ha nem történt visszalépés, akkor folytatjuk a végrehajtást 1.-től az új célsorozattal.

A Prolog végrehajtás során *visszalépés* történik tehát akkor, ha

- egy beépített eljárás meghiúsul, vagy
- ha egy felhasználóieljárás-hívást nem lehet (több) klózfejjel egyesíteni (nincs olyan klózfej, amelyre a redukciós lépés sikerrel futna le).

Visszalépés esetén a következőket teszi a Prolog végrehajtó:

- visszatér a legutolsó sikeres redukciós lépéshez
- annak *bemeneti* célsorozatát megpróbálja *újabb* klózzal redukálni (végrehajtás 2. lépése)
- ennek meghiúsulása *további visszalépést* okoz.

Azaz, a visszalépés során visszamegyünk a legutolsó, sikeres redukciós lépésig (ez szükségképpen felhasználói eljárással történt, hiszen a redukciós lépés ilyenekkel dolgozik). Hívjuk ezt `k.` redukciónak. Minden azóta történt változóbehelyettesítés érvényét veszti. A `k.` redukciós lépés bemenete egy célsorozat és egy, a végrehajtási algoritmus által, a 2. lépésben kiválasztott klóz volt. A végrehajtási algoritmus megpróbál egy *további* (azaz a kiválasztott klóz után álló) klózt találni, amelyre a redukciós lépés az adott célsorozat mellett sikeresen elvégezhető. Ha ilyen nincs, meghiúsulás történik és visszalépünk az ezt megelőző legutolsó, sikeres redukciós lépésig stb. Ha ilyen redukciós lépés már nincsen, akkor *meghiúsul* a hívás.
Ha a végrehajtási algoritmus során üres lesz a célsorozat, akkor *sikeres* lesz a hívás.
A visszalépésnek két fajtáját különböztetjük meg:

- *sekély*: egy eljárás egy klózából ugyanezen eljárás egy későbbi klózába kerül a vezérlés
- *mély*: egy már lefutott eljárás belsejébe térünk vissza, újabb megoldást kérve.

Az alábbiakban egy példát mutatunk a sekély visszelépésre és a végrehajtási algoritmus működésére.
Tekintsük a `sum_tree3` eljárásunk azon változatát, ahol felcseréljük (visszacseréljük) az első két klózt:

```
sum_tree4(Tree, S) :-                 (1)
        integer(Tree), S = Tree.
sum_tree4(Left--Right, S) :-          (2)
        sum_tree4(Left, S1),
        sum_tree4(Right, S2),
        S is S1+S2.
```

A kezdeti célsorozat legyen `sum_tree4(5--3, S)`, a levélösszeg nyilvánvalóan 8.

```
| ?- sum_tree4(5--3,S).
S = 8 ? ;
no
| ?-
```

Ekkor a Prolog végrehajtás a következőképpen alakul:

- az első (egyetlen) hívás az (1) klózzal sikeresen redukálható (azaz a klóz feje egyesíthető a hívással stb.)
- a redukciós lépés eredménye: `integer(5--3), S=5--3`
- az új célsorozat első, beépített hívása meghiúsul, sekély visszalépés következik
- visszatérünk a kezdeti célsorozathoz, de a (2) klóztól folytatva az olyan klózok keresését, amelyekre sikeresen fut le a redukciós lépés; ilyen most a második klóz
- ezzel a klózzal redukálva az új célsorozat: `sum_tree4(5, S1), sum_tree4(3, S2), S is S1+S2`
- ...

3.2.3. Vezérlési szerkezetek

Amikor egy programozási nyelv kapcsán vezérlési szerkezetekről esik szó rögtön ciklusokra, elágazásokra, feltételes szerkezetekre gondolunk. Nincsen ez másképpen Prolog esetében sem, bár mint tudjuk, ciklusok nincsenek, ezeket a felhasználó rekurzióval, illetve a rendszer által nyújtott visszalépésekkel válthatja ki. Ez az alfejezet egy áttekintést ad a Prolog vezérlési szerkezeteiről, részletesebb ismertetésük egy későbbi szakasz feladata.

Tekintsük az alábbi példát:

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

Egy `p` nevű, 2 klózból álló predikátum definícióját láthatjuk. Az ne zavarjon minket, hogy `q/1`, `r/1`, `s/1` eljárások definícióját nem tüntettük fel. Fogadjuk csak el, hogy léteznek, bármi is a törzsük.

A fenti programocska azt állítja, hogy `p(X)` igaz, ha `q(X)` *és* `r(X)` igaz, **vagy** `s(X)` igaz. Ez utóbbi a Prolog végrehajtási mechanizmusának ismeretében érthető teljesen: ha például `q(X)` hívás meghiúsul, akkor a sekély visszalépés során `p/2` második klózával probálunk meg illeszteni, ami az `s(X)` hívássá fejtődik ki.

A fenti Prolog kód az alábbi C programnak feleltethető meg (ha feltesszük, hogy `q`, `r` és `s` igaz vagy hamis visszatérésű függvények):

```
BOOL p(x) {
    return q(x) && r(x) || s(x);
}
```

Fogalmazzuk át a fenti példát úgy, hogy értelmesebb legyen (nyilván mindegy, hogy mik az eljárások nevei):

```
beléphet(X) :- látogató(X), van_engedélye(X).
beléphet(X) :- dolgozó(X).
```

A `beléphet` predikátum jelentése: valaki beléphet (pl. egy üzembe), ha látogató és rendelkezik engedéllyel (első klóz), illetve valaki beléphet akkor is, ha dolgozó. Ezt átfogalmazhatjuk úgy is, hogy valaki beléphet, ha látogató és van engedélye, vagy ha dolgozó.

Ez utóbbi kiolvasásnak megfelel a Prolog diszjunkció szerkezete, amely a következőképpen néz ki:

```
beléphet(X) :-
    (   látogató(X), van_engedélye(Y)
    ;   dolgozó(X)
    ).
```

Itt a ; operátor jelenti a **vagy** kapcsolatot. Diszjunkcók formázásánál mindig a fenti minta szerint járjunk el: tegyük zárójelbe a diszjunkciót, de az egyes ágakat ne tegyük zárójelbe (mert a pontosvessző gyengébben köt, mint a vessző).

A fent látott kétklózos és a diszjunktív alak teljesen ekvivalens egymással. A diszjunkció Prologban szintaktikus édesítőszer, (segéd)eljárások bevezetésével mindig kiküszöbölhető.

Jelen esetben a `beléphet/2` eljárás két klózának feje egyforma volt, ezért is volt ilyen könnyű az átalakítás. Ha a `sum_tree/4` eljárást szeretnénk felírni a ; operátor használatával, akkor szükséges egy újabb `= /2` feltétel bevezetése:

```
sum_tree5(Tree, Sum) :-
        (
            integer(Tree),
            Sum = Tree
        ;
            Tree = Left--Right,         <-- új feltétel
            sum_tree5(Left, Sum1),
            sum_tree5(Right, Sum2),
            Sum is Sum1+Sum2
        ).
```

Nézzünk most egy összetettebb példát! Írjunk meg egy olyan Prolog eljárást, amely képes egy polinom-szerű kifejezés helyettesítési értékét kiszámítani. Ez a polinom-szerű kifejezés számokból, az 'x' névkonstansból a '+' és '*' operátorok ismételt alkalmazásával épül fel. Ezt, mint adattípust így írhatjuk le:

```
% :- type kif == {x} \/ number \/ {kif+kif} \/ {kif*kif}.
```

Például `(x+1)*x+x+2*(x+x+3)` egy ilyen kifejezés.

A megirandó `erteke` eljárás kap egy ilyen kifejezést, és megkapja az `x` névkonstans helyébe irandó számértéket. Harmadik, kimenő argumentuma pedig a kifejezés értéke lesz, az adott helyettesítés mellett.

```
% erteke(Kif, X, E): A Kif formula értéke E, az x=X behelyettesítéssel.
erteke(x, X, E) :-
        E = X.
erteke(Kif, _, E) :-
        number(Kif), E = Kif.
erteke(K1+K2, X, E) :-
        erteke(K1, X, E1),
        erteke(K2, X, E2),
        E is E1+E2.
erteke(K1*K2, X, E) :-
        erteke(K1, X, E1),
        erteke(K2, X, E2),
        E is E1*E2.
```

Programunk egy predikátumból áll. Ennek a predikátumnak négy klóza van, amelyek megfelelnek a fenti típus-definíció négy ágának. A program jelentését az alábbi módon tudjuk összefoglalni :

- ha a kifejezés `x`, akkor a kifejezés értéke a második argumentumban adott behelyettesítési érték
- ha a kifejezés szám, akkor mindegy mi a behelyettesítési érték, a kifejezés értéke önmaga

- ha a kifejezés `A+B` alakú, akkor ennek értéke `A` értékének és `B` értékének összege
- ha a kifejezés `A*B` alakú, akkor ennek értéke `A` értékének és `B` értékének szorzata

Az `erteke` eljárás egy példafutása a következő:

```
| ?- erteke((x+1)*x+x+2*(x+x+3), 2, E).
E = 22 ? ;
no
```

3.3. Listák Prologban

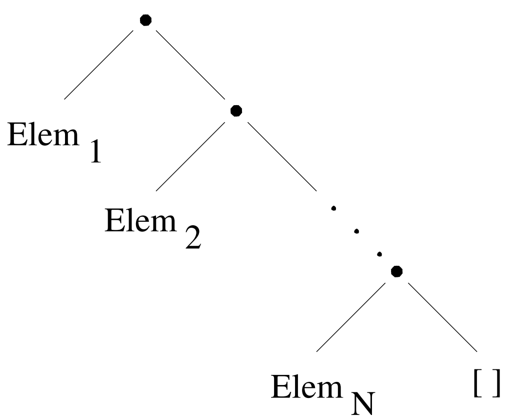
A lista egy közönséges adattípus, amely elemeket tárol adott sorrendben és amelyet az alábbi módon definiálhatunk:

```
% :- type list(T) ---> .(T,list(T)) ; [].
```

Azaz egy `T` típusú elemekből álló lista vagy egy `'.'/2` struktúra vagy a `[]` névkonstans. A struktúra első argumentuma `T` típusú, őt hívjuk a lista fejének (első elemének). A második argumentum `list(T)` típusú, azaz egy újabb lista. Ezt hívjuk az eredeti lista farkának, ez nem más, mint a lista többi eleméből álló lista. Megjegyezzük, hogy a `'.'` név egy közönséges struktúranév. Hívhatnánk akár `alma`-nak is. Ugyanez igaz a `[]` névkonstansra is, amely az üres (0 elemű) listát jelöli. Fogadjuk el, hogy Prolog ezeket a névkonvenciókat vezeti be listák építésére.

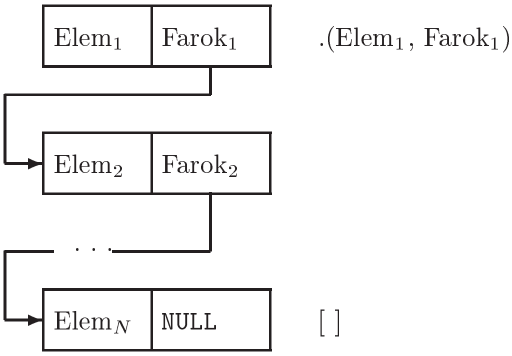
Példa Prolog listára a `.(3,[])`, amely egy egyelemű lista. Hasonlóképpen `.(2,.(4,[]))` is egy Prolog lista. Ez utóbbi két elemű.

A listákat ábrázolhatjuk fastruktúrában, ahol a csomópontok jelölik a `'.'/2` struktúrákat:



3.1. ábra. A LISTÁK FASTRUKTÚRA ALAKJA

A listák megvalósítását úgy képzelhetjük el, hogy a lista farka egy mutató egy másik listára, majd legvégül ez a mutató NULL lesz:



3.3.1. Listák jelölése

A listák, széleskörű alkalmazhatóságukra való tekintettel kitüntetett szerepet játszanak, olyannyira, hogy külön jelölésrendszer könnyíti meg a használatukat. Üres lista jelölésére, konvenciószerűen a `'[]'` névkonstans szolgál, ennek írásakor nem kell az aposztróf-jeleket kitenni: `[]`. Nem üres lista építésére Prologban a `[Fej|Farok]` jelölés szolgál, ahol `Fej` és `Farok` tetszőleges Prolog kifejezések, azaz szám- ill. névkonstansok, változók, listák vagy más összetett adatok lehetnek. Ahhoz, hogy ún. valódi listát kapjunk, `Farok`-nak (üres vagy nem-üres) listának kell lennie, ez esetben ugyanis egy ilyen kifejezés felírható

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

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

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

A Prolog rendszer belsejében mindenképpen a fenti sok-zárójeles alaknak megfelelő fastruktúra (3.1 ábra) tárolódik.

A lista-jelölés egyike a Prolog számos ún. „szintaktikus édesítőszereinek”, azaz olyan írásmódoknak, amelyek a programozó kényelmét szolgálják, de a program beolvasásakor átalakulnak valamilyen szabványos jelöléssé. Foglaljuk össze ezeket a szintaktikus édesítőket (baloldalon láthatóak az édesített alakok):

1. `[Fej|Farok]` $\equiv$ `.(Fej, Farok)`
2. $\texttt{[Elem}_1\texttt{,Elem}_2\texttt{,...,Elem}_N\texttt{|Farok]} \equiv \texttt{[Elem}_1\texttt{|[Elem}_2\texttt{,...,Elem}_N\texttt{|Farok]]}$
3. $\texttt{[Elem}_1\texttt{,Elem}_2\texttt{,...,Elem}_N\texttt{]} \equiv \texttt{[Elem}_1\texttt{,Elem}_2\texttt{,...,Elem}_N\texttt{|[]]}$

Ezeknek megfelelően a `[a,b]` egy pontosan két elemű listát jelöl, hiszen a 3 szabálynak megfelelően a listánk azonos a `[a,b|[]]` listával, ez 2-nek megfelelően azonos a `[a|[b|[]]]` listával, ami 1-nek megfelelően átírható `.(a,.(b,[]))` alakba.

Hasonlóképpen látható, hogy `[a,b|F]` egy *legalább* kételemű, ún. *nyílt végű* listát jelöl. További példákat láthatunk alább, amelyek a listákon kívül az egyesítési algoritmusra is alkalmazást mutatnak.

```
| ?- [1,2] = [X|Y].                    ⇒  X = 1, Y = [2] ?
| ?- [1,2] = [X,Y].                    ⇒  X = 1, Y = 2 ?
| ?- [1,2,3] = [X|Y].                  ⇒  X = 1, Y = [2,3] ?
| ?- [1,2,3] = [X,Y].                  ⇒  no
| ?- [1,2,3,4] = [X,Y|Z].              ⇒  X = 1, Y = 2, Z = [3,4] ?
| ?- L = [1|_], L = [_,2|_].           ⇒  L = [1,2|_A] ? % nyílt végű
| ?- L = .(1,[2,3|[]]).                ⇒  L = [1,2,3] ?
| ?- L = [1,2|.(3,[])].                ⇒  L = [1,2,3] ?
| ?- [X|[3-Y/X|Y]]= .(A, [A-B,6]).     ⇒  A=3, B=[6]/3, X=3, Y=[6] ?
```

3.3.2. Listák összefűzése

Egy klasszikus listakezelő Prolog eljárás a két lista összefűzését végző `append/3`. Ez az eljárás megtalálható a SICStus `lists` könyvtárban, amelyet a következőképpen tölthetünk be:

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

A könyvtárban az `append/3` a következőképpen van definiálva:

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

Az első klóz a rekurzió leállítását végzi: egy üres listát egy `L` lista elé fűzve az eredmény `L` maga. A második klóz arról az esetről szól amikor az első lista nem-üres: tekintsük az első lista farkát (ez lesz `L1`) és ezt fűzzük össze rekurzívan a második listával, ennek eredményét jelöljük `L3`-mal. Ez utóbbi elé helyezve az első lista fejét (`[X|L3]`) kapjuk az eredményt.

Sokan elsőre az `append` egy másik változatát írnák meg, valami ehhez hasonlót:

```
append0([], L2, L):-
    L = L2.
append0([X|L1], L2, L12) :-
    append0(L1, L2, L3), L12 = [X|L3].
```

Itt az egyenlőség mindkét klózban kiküszöbölhető, és így áll elő az előző, tömörebb változat. Pl. a második klózban a fejbeli `L12` helyettesíthető a vele egyenlő `[X|L3]` kifejezéssel. Vegyük észre, hogy mindezt a Prolog logikai változó fogalma teszi lehetővé: a fej harmadik argumentumában az eredmény részlegesen kitöltött: a fejegyesítés pillanatában egy olyan lista áll elő, melynek első eleme, `X`, ismert, de a farka, `L3` még nem. Az eredmény majd úgy áll elő, hogy a rekurzív `append` hívás kitölti az `L3` változót.

Nézzük meg, hogyan is hajtódik végre az eredeti, tömör-kódú `append` eljárás!

```
    > append([1,2,3],[4],A), write(A)
(2) > append([2,3],[4],B), write([1|B])
(2) > append([3],[4],C), write([1,2|C])
```

```
(2) > append([],[4],D), write([1,2,3|D])
(1) > write([1,2,3,4])
[1,2,3,4]
BIP > []
L = [1,2,3,4] ?
```

A kiinduló célsorozat a `append([1,2,3],[4],A), write(A)` volt. A célsorozat első hívásának az első klózzal való illesztése nyilvánvalóan sikertelen, míg a második klóz fejével sikeres. Az illesztés eredménye egyrészt az, hogy szétszedjük az első listát: `X = 1, L1 = [2,3]`, változatlanul továbbadjuk a második listát: `L2 = [4]`, és végül a harmadik, kimenő argumentumba egy részlegesen kitöltött listát helyettesítünk: `L = [1|L3]`. Ismét hangsúlyozzuk, hogy itt `L3` egy még behelyettesítetlen, ismeretlen mennyiség. Az illesztést követően redukáljuk az eredeti hívást az alkalmazott klóz törzsére valamint elvégezzük a változóbehelyettesítéseket a célsorozat maradékára is (`A=[1|B]`) és így az új célsorozat:

```
append([2,3], [4], B), write([1|B]).
```

A változóneveket a redukciós lépésben cserélte le a Prolog rendszer. Az `L3` változó (ami most `B`) ebben a rekurzív hívásban kap értéket (ezt végigkövethetjük a fenti futási példán). Végül `L3 = [2,3,4]` lesz és ennek következtében alakul ki az eredeti hívás végeredménye: `L = [1|L3] = [1,2,3,4]`. A Prolog változó- és egyesítés-fogalma így azt tette lehetővé, hogy az `1` listaelemet már akkor elhelyezzük az eredménylista fejében, amikor annak farka még nem állt rendelkezésre.

Ez az `append` eljárás esetében azért is különösen fontos, mert így az eljárás *jobbrekurzívvá* vált, azaz saját magát csak a törzs utolsó hívásaként hívja vissza. A jobbrekurzív hívásokat ugyanis a Prolog rendszer nem rekurzióval, hanem a hagyományos nyelvek ciklusának megfelelő módon valósítja meg, amivel lényegesen csökken a futás memória- és időigénye (lásd 4.2). *Ezzel szemben*, ha az `append0` eljárást nézzük, azt találjuk, hogy ott saját magát nem a törzs utolsó hívásaként hívja vissza a második klóz. Lássuk mit is jelent ez a futás szempontjából!

```
    > append0([1,2,3],[4],A)
(2) > append0([2,3],[4],B), A=[1|B]
(2) > append0([3],[4],C), B=[2|C], A=[1|B]
(2) > append0([],[4],D), C=[3|D], B=[2|C],A=[1|B]
(1) > D=[4], C=[3|D], B=[2|C], A=[1|B]
BIP > C=[3,4], B=[2|C], A=[1|B]
BIP > B=[2,3,4], A=[1|B]
BIP > A=[1,2,3,4]
BIP > []
L = [1,2,3,4] ?
```

Látható, hogy egy „oda-vissza” jellegű futást kaptunk. A rekurzió legvégén eljutunk odáig, hogy az üres listát kell a `[4]` lista elé fűznünk. Ez meg is történik, ekkor azonban még nincsen meg az eredmény. A kapott részeredményeket ugyanis (amiket „mellesleg” a rendszernek futás közben végig tárolnia kellett) még sorozatos egyesítések árán kell a végeredménnyé alakítani.

Azt tapasztaljuk tehát, hogy bár mindkét Prolog predikátum ugyanazt a relációt írja le (a harmadik lista az első két lista elemeinek egymás után fűzésével áll elő), az egyik jóval hatékonyabb, mint a másik. Jegyezzük meg, hogy az `append` futása az *első* lista hosszával arányos idejű.

Lássunk most egy másfajta példát, állítsuk elő egy bináris fa leveleinek listáját! Valami ilyesmit szeretnénk:

```
| ?- leaves(5--(3--2), L).
L = [5,3,2] ? ;
no
```

A kód alább látható. A `leaves/2` predikátum első klóza levél esetén egy olyan egyelemű listát ad vissza, amelynek egyetlen eleme a levél értéke. A második klóz kezeli a csomópontokat. Ilyenkor a baloldali részfának és a jobboldali részfának megfelelő levél-listák összefűzésével kapjuk az adott csomóponton érvényes levéllistát.

```
:- op(500, xfx, --).

% :- type tree == integer \/ {tree--tree}.
% leaves(Tree, Leaves): Tree leveleinek listája Leaves.
leaves(Tree, L):-
    integer(Tree),
    L=[Tree].
leaves(Left--Right, L) :-
    leaves(Left, L1),
    leaves(Right, L2),
    append(L1, L2, L).
```

3.3.3. Listák megfordítása

Az előzőekben láttunk Prolog megvalósítást listák összefűzésére. Most olyan eljárást szeretnénk írni, amely megfordít egy listát. Első próbálkozásunk az alábbi lehet:

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

Az első klóz jelentése, hogy üres lista megfordítottja az üres lista. A második klóz azt mondja, hogy egy legalább egyelemű lista megfordítottja az a lista, amit úgy kapunk, hogy megfordítjuk a bemenő lista farkát, majd ennek a végére fűzzük a bemenő lista fejét — egyelemű listaként.

Megoldásunk működik:

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

Ha kicsit jobban belegondolunk rájövünk, hogy a megvalósított algoritmus négyzetes lépésszámú a lista elemeinek számában (n elemű lista esetén n `append` hívás lesz, amelyek rendre 1 ..., n redukciós lépést igényelnek).

Listát meg lehet fordítani lineáris időben is, mint ahogyan ezt teszi az alábbi Prolog kód is:

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

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

Ebben a megvalósításban az az érdekes, hogy felhasználunk egy *segédeljárást* és egy ún. *gyűjtő* vagy más néven *akkumulátor* argumentumot. A program megértéséhez induljunk ki a `revapp/2` eljárásból: ez az első argumentumában megadott lista megfordítottját fűzi a második argumentumában kapott listához, ez lesz a harmadik argumentum. Ha üres lista megfordítottját kell egy lista elé fűzni, akkor az eredmény az, mintha nem csinálnánk semmit. Erről szól a `revapp/2` első klóza. Amennyiben egy legalább egyelemű lista (feje `X`, farka `L1`) megfordítottját kell egy `L2` lista elé fűznünk, akkor az ugyanaz, mintha azt a feladatot szeretnénk megoldani, hogy a lista farkának (`L1`) megfordítását fűzzük hozzá az `[X|L2]` listához.

Ugyanezt más módon magyarázva: a a `revapp/2` a második argumentumában gyűjti az eredményt és a rekurziót leállító klóz teszi ezt át a kimenő, harmadik argumentumba.

A SICStus lists könyvtár a `reverse/2` eljárást is tartalmazza.

3.3.4. Példák listakezelésre

Láttuk, hogy sokszor lehetőségünk van hatékony, jobbrekurzív megoldások készítésére. Ebben az `append/2` esetében az segített, hogy a rekurzív hívást képesek voltunk a klóz utolsó hívásának helyére tenni. A lista megfordítása probléma esetén a gyűjtőargumentumos segédeljárás elve segítette elő a megoldás hatékonyabbá tételét.

Térjünk most vissza a már megszokott bináris fáinkhoz, tegyük jobbrekurzívvá először azt a predikátumot, amely egy bináris fa leveleinek listáját adja vissza!

Első megoldásunk ez volt:

```
leaves(Tree, L):-
    integer(Tree),
    L=[Tree].
leaves(Left--Right, L) :-
    leaves(Left, L1),
    leaves(Right, L2),
    append(L1, L2, L).
```

Jó lenne, ha sikerülne kiküszöbölni az `append/2` hívást, hiszen akkor legalább az egyik `leaves/2` hívás „jobbrekurzív pozícióban” állna. Ezt megtehetjük, ha bevezetünk egy segédeljárást, amelynek második argumentuma egy akkumulátor lesz. Ez kezdetben az üres lista. Ez elé fűzzük mindig be a megfelelő levéllistát, valahogy így:

```
% leaves2(Tree, L): Tree leveleinek listája L
leaves2(Tree, L) :-
        leaves2(Tree, [], L).

% leaves2(Tree, L0, L): Tree leveleinek listáját L0 elé fűzve
% kapjuk az L listát.
leaves2(Tree, L0, [Int|L0]):-
    integer(Tree).
leaves2(Left--Right, L0, L) :-
    leaves2(Right, L0, L1),
    leaves2(Left, L1, L).
```

A jegyzetben eddig szereplő `sum_tree` megvalósítások egyike sem volt jobbrekurzív. Például:

```
% A Tree bináris fa levélösszege Sum
sum_tree(Tree, S) :-
        integer(Tree), S = Tree.
sum_tree(Left--Right, S) :-
        sum_tree(Left, S1),
        sum_tree(Right, S2),
        S is S1+S2.
```

Itt a második klózban a rekurzív hívás nem a legutolsó helyen szerepel. Hogy ezt elérjük, itt is egy gyűjtőargumentumot kell bevezetnünk:

```
% A Tree bináris fa levélösszege Sum
sum_tree6(Tree, S) :-
```

```
        sum_tree6(Tree, 0, S).

% A Tree bináris fa levélösszege Sum0-hoz adva Sum-ot ad
sum_tree6(Tree, Sum0, Sum) :-
        integer(Tree),
        Sum is Sum0+Tree.
sum_tree6(Left--Right, Sum0, Sum) :-
        sum_tree6(Left, Sum0, Sum1),
        sum_tree6(Right, Sum1, Sum).
```

Végezetül nézzünk meg egy örökzöld problémát: adjuk meg egy számlista elemeinek összegét. Első és tipikusan nem hatékony megoldás az alábbi:

```
sum_list([],0).
sum_list([X|Xs],Sum):-
    sum_list(Xs,Sum0),
    Sum is X+Sum0.
```

Eszerint az üres lista összege 0, egy nem üres lista összege a lista farkának az összege plusz a lista feje.

Ennél sokkal hatékonyabb megoldás az, ha egy kezdetben nulla értékű gyüjtőargumentumhoz folyamatosan hozzáadjuk a lista éppen aktuális fejét, amit deklaratív szemlélettel az alábbi módon valósíthatunk meg:

```
sum_list1(L, S) :-
        sum_list1(L, 0, S).

% sum_list1(+L, +S0, -S): Az L számlista összege S0-hoz adva S-t ad.
sum_list1([], S0, S0).
sum_list1([X|L], S0, S) :-
        S1 is S0+X,
        sum_list1(L, S1, S).
```

3.4. Tömör és minta-kifejezések

Ebben a szakaszban definiálunk néhány dolgot, amelyek ismeretére később szükség lesz. Egy Prolog kifejezés *tömör* (ground), ha nem tartalmaz változót. *Mintának* hívunk egy általában nem tömör kifejezést, amely mindazon kifejezéseket „képviseli”, amelyek belőle változó-behelyettesítéssel előállnak. Egy olyan mintát, amely listát is képvisel (azaz elő lehet belőle állítani megfelelő változóbehelyettesítésekkel listát) *lista-mintának* nevezzük.

Nyílt végű listának nevezünk egy olyan lista-mintát, amely bármilyen hosszú listát is képvisel. **Zárt végű** lista egy olyan lista-minta, amely egyféle hosszú mintát képvisel.

A fenti fogalmakra mutatnak példát az alábbiak:

Zárt végű lista	Milyen listákat képvisel
`[X]`	egyelemű
`[X,Y]`	kételemű
`[X,X]`	két egyforma elemből álló
`[X,1,Y]`	3 elemből áll, 2. eleme `1`

Nyílt végű lista	Milyen listákat képvisel
`X`	tetszőleges
`[X|Y]`	nem üres (legalább 1 elemű)
`[X,Y|Z]`	legalább 2 elemű
`[a,b|Z]`	legalább 2 elemű, elemei: `a, b, ...`

3.5. Visszalépéses keresés Prologban

A Prolog végrehajtási algoritmusának ismertetésekor megismerkedtünk a *sekély*, illetve *mély* visszalépés fogalmával. Sekélynek hívtunk egy visszalépést, ha egy eljárás egy klózából ugyanezen eljárás egy későbbi klózába kerül a vezérlés automatikusan, míg mélynek, ha egy már lefutott eljárásba térünk vissza.

Idézzük most fel a bevezetőben megismert családi kapcsolatok példát! Adott 6 tényállítás, amely gyermek-szülő kapcsolatokat ír le. A tömörség kedvéért a `szülője` nevet egyszerű `sz`-re cseréltük, valamint Civakodó Henrik és Burgundi Gizella nevét is rövidítettük.

```
sz('Imre', 'István').  % (1)
sz('Imre', 'Gizella'). % (2)
sz('István', 'Géza').  % (3)
sz('István', 'Sarolt').% (4)
sz('Gizella', 'CH').   % (5)
sz('Gizella', 'BG').   % (6)
```

A tényállításoknak megfelelően István szülei Géza, illetve Sarolt stb.

Definiáltunk egy `nagyszülője` relációt, amely az értelemszerű jelentéssel bír. Ennek a (csak az struktúra- és változónevek tekintetében) rövidebb változatát láthatjuk alább.

```
nsz(Gy, N) :-
    sz(Gy, Sz),
    sz(Sz, N).
```

Tegyük fel azt a kérdést, hogy kik a nagyszülei Imrének és elemezzük a 3.2. ábrát!

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

Láthatjuk, hogy a célsorozatunkban szereplő egyetlen hívást első lépésben redukálta a Prolog rendszer a `sz('Imre',Nsz), sz(Sz,Nsz)` célsorozatra. Itt még nem jött létre ún. *választási pont*, hiszen a kezdeti célsorozatban szereplő egyetlen hívást eleve csak az `nsz/2` predikátum valamely klózának fejével van esélyünk illeszteni és így redukciós lépést végrehajtani. Mivel az `nsz/2` eljárásnak csak egyetlen klóza van, ezt csak egyféleképpen tehetjük meg.

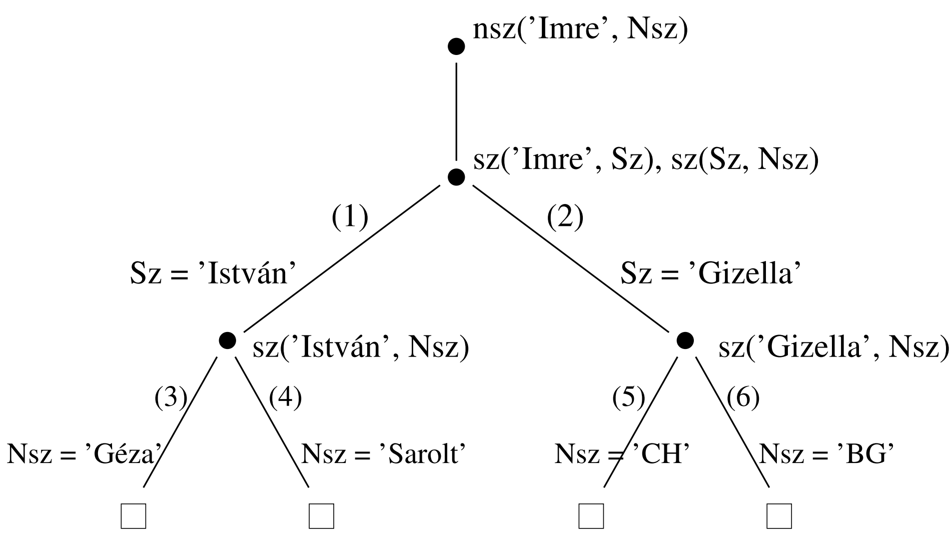
Az új célsorozat első eleme a `sz('Imre',Nsz)` kifejezés. A Prolog végrehajtási algoritmus megpróbál végre-hajtani egy redukciós lépést ezzel a kifejezéssel és az első `sz/1` klózzal. Ez sikerül (jegyezzük meg, hogy itt létrejött egy választási pont). A keletkező változóbehelyettesítést (`Sz = 'István'`) végigvezetjük a célsorozat hátralevő részén. Majd a célsorozat első elemét egyszerűen elhagyjuk, mert jelen esetben nincsen törzs, amire cserélhetnénk. A kialakuló új (egyelemű) célsorozat: `sz('István', Nsz)`.

Ezek után az új célsorozat első elemével és az első `sz/2` klózzal a Prolog végrehajtó megpróbál végrehajtani egy redukciós lépést. Ez azonban nem sikerül, mert sikertelen a fejillesztés (az 'István' névkonstans nem egyesíthető az 'Imrével'). A végrehajtási algoritmusnak megfelelően megkeressük az első olyan klózt, amely redukálható a célsorozatunk első elemével. Ez a 3-as klóz lesz, a keletkező változóbehelyettesítés `Nsz = 'Géza'`, az új célsorozat az üres célsorozat.

A végrehajtás tehát befejeződött, válaszként megkaptuk, hogy Imre nagyszülő relációban áll Gézával. Ez volt a legtávolabbi pont, amíg eddig a jegyzet keretein belül eljutottunk. Igen ám, de minket érdekelhetnek Imre további nagyszülei is. Erre szolgál a `;` a Prolog interaktív felhasználói felületén:

```
| ?- nsz('Imre',Nsz).
Nsz = 'Géza' ? ;
Nsz = 'Sarolt' ? ;
Nsz = 'CH' ? ;
Nsz = 'BG' ? ;
no
| ?-
```

Az újabb megoldás kérését jelző `;` választ úgy lehet felfogni, mint egy „mesterséges meghiúsulást”, amit a felhasználó válthat ki. Ilyenkor *mély* visszalépés történik: egy már lefutott eljárás belsejébe térünk vissza újabb megoldást keresni. Mindemellett pontosan úgy járunk el, mint sekély visszalépés esetén: megkeressük a legutolsó sikeres redukciós lépést — ez megfelel a legutóbbi választási pontnak — és az akkori célsorozat legelső hívását próbáljuk meg másképpen redukálni. A hívás amiről szó van a `sz('István',Nsz)` és a végrehajtó képes ezt a 4. klózzal redukálni, szolgáltatva így egy újabb megoldást. A maradék két megoldáshoz vezető végrehajtási út végiggondolását az olvasóra bízzuk.

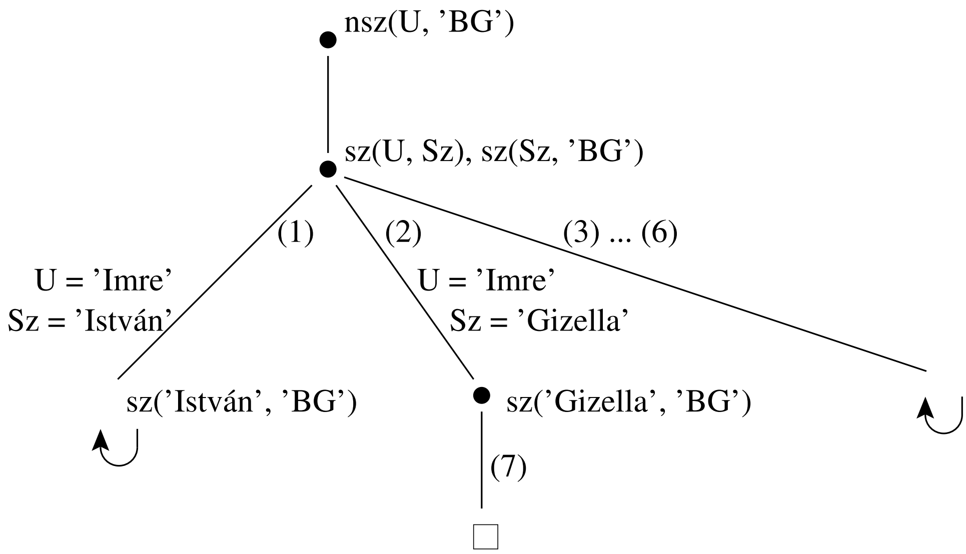


3.2. ábra. Prolog végrehajtási példa - családi kapcsolatok

Vegyük észre, hogy a fenti példában nem volt *sekély* csak *mély* visszalépés. Nézzünk most meg egy újabb futást egy más célsorozatra! Azt kérdezzük, hogy kik az unokái Burgundi Gizellának:

```
| ?- nsz(U,'BG').
U = 'Imre' ? ;
no
| ?-
```

Elemezzük a 3.3. ábrát! A célsorozat az első redukciós lépés után `sz(U,Sz), sz(Sz,'BG')` lesz. Ezt redukáljuk `sz/2` első klózával, ekkor `U = 'Imre'` és `Sz = 'István'` behelyettesítések mellett az új célsorozat `sz('István','BG')` lesz. Ez azonban meghiúsulást okoz, mert nincs olyan klózfej, amivel ez egyesíthető lenne. Így ez *automatikusan* visszalépést okoz (mély visszalépés, mert az első `sz/2` hívás sikeresen lefutott). A legutolsó sikeres redukciós híváskor a célsorozat `sz(U,Sz), sz(Sz,'BG')` volt. Ezt próbálja most meg a Prolog rendszer másképpen végrehajtani: más klózzal próbálja meg illeszteni a célsorozat első hívását. A második klózzal meg is történik a redukció és végül kialakul az egyetlen válasz.



3.3. ábra. PROLOG VÉGREHAJTÁSI PÉLDA (2) - CSALÁDI KAPCSOLATOK

3.5.1. A teljes Prolog végrehajtási algoritmus

Ennyi ismeret után készen állunk arra, hogy az eddigieknél precízebben ismertessük a Prolog végrehajtási algoritmusát. Ez az alábbi:

1. *(Kezdeti beállítások:)* A verem üres, `CS := célsorozat`
2. *(Beépített eljárások:)* Ha `CS` első célja beépített akkor hajtsuk végre,
   a. Ha sikertelen ⇒ 6. lépés.
   b. Ha sikeres, elvégezzük a behelyettesítéseket, `CS`-ből elhagyjuk az első hívást, ⇒ 5. lépés.
3. *(Klózszámláló kezdőértékezése:)* `I = 1`.
4. *(Redukciós lépés:)* `CS` első hívásához tartozó eljárásdefinícióban `N` klóz van.
   a. Ha `I > N` ⇒ 6. lépés.
   b. Redukciós lépés az `I`-edik klóz és a `CS` célsorozat között.
   c. Ha sikertelen, akkor `I := I+1` ⇒ 4. lépés.
   d. Ha `I < N` (nem utolsó), akkor vermeljük `<CS,I>`-t.
   e. `CS :=` a redukciós lépés eredménye
5. *(Siker:)* Ha `CS` üres, akkor sikeres vég, egyébként ⇒ 2. lépés.
6. *(Sikertelenség:)* Ha a verem üres, akkor sikertelen vég.
7. *(Visszalépés:)* Ha a verem nem üres, akkor leemeljük a veremből `<CS, I>`-t, `I := I+1`, és ⇒ 4. lépés.

3.5.2. A 4-kapus doboz modell

A keresési fa mellett van a Prolog végrehajtásnak egy másik megjelenítési formája, az ún. eljárás-doboz modell (procedure box model). Ezt néha Byrd-doboz modellnek is (Lawrence Byrd volt az egyik első Prolog nyomkövető rendszer létrehozója), illetve 4-kapus doboz modellenek is hívják.

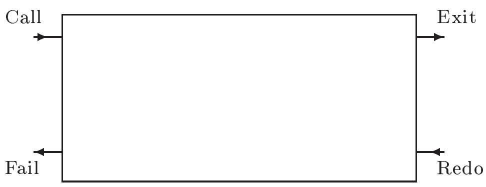
Tekintsük az alábbi példát: keressük meg azokat a kétjegyű számokat amelyek négyzete háromjegyű és a szám fordítottjával keződik.

Erre egy Prolog megvalósítás az alábbi.

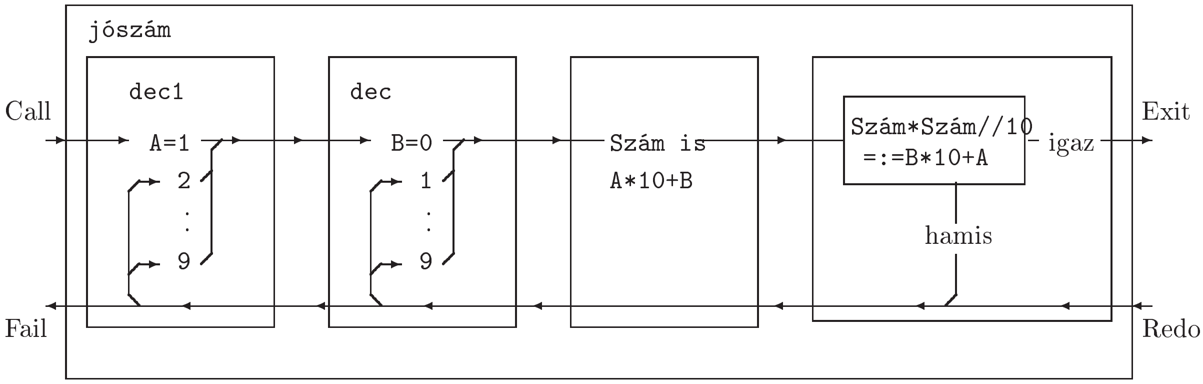
```
% dec1(J): J egy pozitív decimális számjegy.
dec1(1). dec1(2). dec1(3). dec1(4). dec1(5). dec1(6). dec1(7),
dec1(8). dec1(9).

% dec(J): J egy decimális számjegy.
dec(0).
dec(J) :-
    dec1(J).

% Szam négyzete háromjegyű és a Szam fordítottjával kezdődik.
jószám(Szam):-
        dec1(A), dec(B),                  (1)
        Szam is A * 10 + B,               (2)
        Szam * Szam // 10 =:= B * 10 + A.   (3)
```

A program működését a 4-kapus doboz modellen tanulmányozzuk. A működés megértéséhez nagymértékben támaszkodunk arra, hogy már teljesen ismerjük a Prolog végrehajtási mechanizmusát.

A 4-kapus doboz modellben egy eljáráshívást egy olyan dobozzal ábrázolunk amelynek négy ún. kapuja van: hívás (`Call`), kilépés (`Exit`), új-megoldás (`Redo`) és sikertelenség (`Fail`):



Amikor meghívunk egy eljárást, akkor a `Call` kapun megyünk be. Egy futó eljárásból kétféleképp léphetünk ki: sikeresen az `Exit` kapun, illetve sikertelenül a `Fail` kapun. Végül egy már sikeresen lefutott eljárásba visszaléphetünk egy új megoldás keresése végett a `Redo` kapun (mély visszalépés); ezután ismét az `Exit` vagy a `Fail` kapu következik, stb. A kapuk fenti elrendezése a dobozok olyan összekapcsolását teszi lehetővé, amely a Prolog végrehajtási sorrendnek felel meg. Ennek bemutatására tekintsük a fenti program `jószám/1` eljárását szemléltető dobozt:



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

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

Programunk működési elve lényegében egy kétszeres ciklus. A `dec1/1` eljárás 9-féleképpen sikerülhet, a `dec/1` eljárás 10 különféle megoldást képes szolgáltatni. A `jószám/1` predikátum 3. sora nem hoz létre választási pontot, míg a negyedik sor vagy sikerül (ekkor találtunk egy megoldást) vagy meghiúsul. Végül nézzük meg programunk futását:

```
| ?- joszam(A).
A = 27 ? ;
no
| ?-
```

3.5.3. Példák

Az alábbiakban bemutatunk egy Prolog programot, amely egy adott számintervallumban található számokat sorolja fel. Ezután ismertetünk egy Prolog eljárást, amely listákban képes keresni. Végül bemutatunk egy olyat, amely képes listaelemet „törölni” egy listából.

Kezdjük az elsővel! Feladatunk egy `between` nevű eljárás megírása, amely képes adott intervallumba eső egész számokat felsorolni, ahogyan azt az alábbi példafutás mutatja:

```
| ?- between(1,5,A).
A = 1 ? ;
A = 2 ? ;
A = 3 ? ;
A = 4 ? ;
A = 5 ? ;
no
| ?-
```

A megoldást rekurzív módon lehet megfogalmazni:

```
% between(N, M, I): I olyan egész, amely az N és M egész
% számok közé esik (N =< I =< M).
between(M, N, M):-
    M =< N.
between(M, N, I):-
    M < N,
    M1 is M+1,
    between(M1, N, I).
```

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

A müködés kis gondolkodás után magától értetődő: az első klóz mindig szolgáltat egy megoldást, méghozzá az intervallum legbaloldalibb elemét. Ha új megoldást kérünk (mély visszalépés), akkor a második klóz törzsére redukálódik a célsorozat, ami viszont nem más, mint egy újabb `between` hívás eggyel nagyobb alsó határral, ami megint illeszkedik majd az első klózzal stb.

Végül nézzünk egy összetettebb futási példát a `between/3` használatára! Keressük azon számokat, amelyek kétjegyűek, első jegyük 1-es vagy 2-es és a második számjegyük 3-as vagy 4-es:

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

Második feladatunk egy olyan eljárás megírása, amely képes listákban keresni. Ez azt jelenti, hogy képes eldönteni egy listáról, hogy egy adott elem benne van-e.

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

Képes továbbá felsorolni egy listának az elemeit.

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

Az eljárás kódja a következő:

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

A `member/2` eljárásnak mindössze két klóza van. Az első azt állítja, hogy `Elem` benne van egy olyan listában, amelynek első eleme éppen `Elem`. Ez nyilván igaz. A második klóz azt mondja, hogy `Elem` benne van a második argumentumként megadott listában, ha benne van annak farkában.

A fenti két-klózos `member` eljárást átírhatjuk egy-klózossá a `;` operátor (diszjunkció) segítségével:

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

A két program teljesen ekvivalens egymással.

A most megírt `member/2` eljárást az eddigieken kívül még másra is használhatjuk. Segítségével például meghatározhatjuk két lista metszetét (azokat az elemeket, amik mindkét listában benne vannak):

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

Arra is jó a `member/2`, hogy egy ismeretlen listáról kijelentsük, hogy annak valami eleme:

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

Valóban igaz, hogy `1` eleme az `[1|_A]` listának, a `[_A,1|_B]` listának stb. Sajnos viszont így végtelen választási ponthoz jutunk.

Harmadik és utolsó feladatunk egy `select` nevű eljárás megírása, amely képes „törölni" egy elemet egy listából. A megfogalmazás szándékosan zavaró azért, hogy felhívhassuk a figyelmet arra, hogy ilyet Prologban nem lehet csinálni! Egy listába nem lehet új elemet beszúrni, abból elemet elvenni. Mint ahogyan egy változónak sem lehet új értéket adni, miután behelyettesítettük valamivel. Jelen esetben amit tehetünk, hogy *létrehozunk* egy új listát, amely már nem tartalmazza a kiválasztott elemet.

Lássuk tehát a kódot!

```
% select(Elem, Lista, Marad):  Elemet a Lista-ból elhagyva marad Marad
select(Elem, [Elem|Marad], Marad).
select(Elem, [X|Farok], [X|Marad0]) :-
        select(Elem, Farok, Marad0).
```

A működést egyszerűen összefoglalhatjuk. Egy `Elem`-et egy olyan listából, amely `Elem`-mel kezdődik úgy kell elhagyni, hogy egyszerűen a lista farka marad. Erről szól az első klóz. A második klóz azt mondja, hogy az első listaelemet változatlanul hagyjuk és a farokból hagyjuk el `Elem`-et. A két klóz együtt éppen a kívánt működést nyújtja.

Nézzünk néhány példafutást! Látható, hogy a megírt eljárás több módban is használható. A harmadik példában arra használjuk, hogy eldöntsük mi lehetett az a lista, amelyből 3-at elhagyva `[1,2]` maradt?

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

```
        L = [3,1,2] ? ;
        L = [1,3,2] ? ;
        L = [1,2,3] ? ;
        no
| ?- select(3, [2|L], [1,2,7,3,2,1,8,9,4]).
        no
| ?- select(1, [X,2,X,3], L).
        L = [2,1,3], X = 1 ? ;
        L = [1,2,3], X = 1 ? ;
        no
```

Mind a `member/2`, mind a `select/3` része a SICStus Prolog lists könyvtárának.

Láttuk, hogy a `member/2` eljárás végtelen választási pontot eredményezett, amikor $member(1, L)$ módon hívtuk meg. Eddig még nem mutattunk rá, de az `append/3` sem mindig „biztonságos”, például az alábbi esetben végtelen választási pontot okoz:

```
| ?- append(A,[1,2,3],C).
A = [], C = [1,2,3] ? ;
A = [_A], C = [_A,1,2,3] ? ;
A = [_A,_B], C = [_A,_B,1,2,3] ?
...
```

A `select/3` esetében is elő lehet idézni a fentiekhez hasonló anomáliát.

„Biztonságosnak” hívunk egy futást, ha véges a keresési tér. A fenti eljárások esetében a következő esetekben lesz biztonságos a futás:

- `member/2` második argumentuma zárt végű.
- `select/3` 2. és 3. argumentuma közül az egyik zárt végű.
- `append/3` 1. és 3. argumentuma közül az egyik zárt végű.

3.5.4. Indexelés

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

A determinizmus felismerése érdekében a legtöbb Prolog fordítóprogram alkalmazza az ún. *indexelés* módszerét. Ez azt jelenti, hogy amikor egy hívást elkezdünk végrehajtani, először az őt definiáló eljárás klózai közül valamilyen egyszerű ismérv szerint kiszűrjük azokat, amelyek biztosan nem lesznek vele illeszthetők. A legtöbb Prolog rendszer, így a SICStus Prolog is, az első argumentum szerint indexel: ha a hívásban az első argumentum változó, akkor az eljárás mindegyik klózával próbál illeszteni (mindegyik klózra megkísérli a redukciós lépést), ha viszont nem változó, akkor a klózoknak csak a megfelelő rész-sorozatával illeszt.

A rész-sorozatokat *fordítási időben* alakítjuk ki. Minden legalább két klózzal rendelkező eljáráshoz készítünk ilyen csoportosítást. Ehhez az első argumentum legkülső funktorát vesszük figyelembe, azaz az első argumentumpozíción azonos konstans-értékeket, illetve az azonos nevű és argumentumszámú struktúrákat tartalmazó klózokat rakjuk egy csoportba (az első helyen változót tartalmazó klózok mindegyik csoportba belekerülnek).

Futáskor lényegében konstans idő alatt választunk a rész-sorozatok közül. Ennek megfelelően az alábbi módon módosul a Prolog végrehajtási algoritmusa:

1. *(Kezdeti beállítások:)* A verem üres, `CS := célsorozat`

2. *(Beépített eljárások:)* Ha `CS` első célja beépített akkor hajtsuk végre,
   a. Ha sikertelen ⇒ 6. lépés.
   b. Ha sikeres, elvégezzük a behelyettesítéseket, `CS`-ből elhagyjuk az első hívást, ⇒ 5. lépés.
3. *(Klózszámláló kezdőértékezése:)* `I = 1`.
4. *(Redukciós lépés:)* `CS` *első hívásához elkészítjük a potenciálisan illeszthető klózok listáját (indexelés). Tegyük fel, hogy ez a lista* `N` *elemű.*
   a. Ha `I > N` ⇒ 6. lépés.
   b. Redukciós lépés az *indexelési lista* `I`-edik klóza és a `CS` célsorozat között.
   c. Ha sikertelen, akkor `I := I+1` ⇒ 4. lépés.
   d. Ha `I < N` (nem utolsó), akkor vermeljük `<CS,I>`-t.
   e. `CS :=` a redukciós lépés eredménye
5. *(Siker:)* Ha `CS` üres, akkor sikeres vég, egyébként ⇒ 2. lépés.
6. *(Sikertelenség:)* Ha a verem üres, akkor sikertelen vég.
7. *(Visszalépés:)* Ha a verem nem üres, akkor leemeljük a veremből `<CS, I>`-t, `I := I+1`, és ⇒ 4. lépés.

Tehát a negyedik lépésben nem az összes klóz közül, hanem csak egy előzetesen átrostált halmazból választunk jelöltet a redukciós lépés végrehajtásához. Nagyon fontos tudni, hogy ha ez a halmaz csak egyelemű, akkor nem jön létre választási pont. Ez jelentősen megnövelheti a programunk hatékonyságát. Mégegyszer: arról van szó, hogy ha egy hívásról „látszik", hogy az adott eljárás csak egy klózával tud illesztődni (csak azzal a klózzal lehetséges a redukciós lépés), akkor nem jön létre választási pont. Azaz a rendszer, meghiúsulás esetén, nem fog ide még annyi ideig sem visszatérni, hogy észrevegye, hogy nincsen több illeszthető klóz.

Példaként tekintsük a következő eljárásdefiníciót:

```
p(0, a).            % (1)
p(X, t) :- q(X).    % (2)
p(s(0), b).         % (3)
p(s(1), c).         % (4)
p(9, z).            % (5)
```

Lássuk mely klózok lehetnek potenciális jelöltek arra, hogy velük együtt redukciós lépést lehessen végrehajtani a célsorozattal, ha a célsorozat az egy elemű `p(A,B)` kifejezés.

- ha `A` változó, a potenciális jelöltek: (1)(2)(3)(4)(5)
- ha `A = 0`, a jelöltek: (1)(2)
- ha `A` fő funktora `s/1`, azaz például `A = s(alma)` vagy `A = s(s(1))`, a jelöltek: (2)(3)(4)
- ha `A = 9`, a jelöltek: (2)(5)
- minden más esetben csak a (2)-es klóz lesz megjelölt

Vegyük észre, hogy az indexelés a struktúrák argumentumaiban lévő értékeket már nem veszi figyelembe, csak magát a legkülső funktort. Egy `p(s(3),r)` hívás biztosan meghiúsul, hiszen nem lehet egyetlen klózzal sem illeszteni, mégis az indexelés szerint a (2),(3) és (4)-es klózok potenciális jelöltek maradnak.

Végül következzen itt néhány megfontolandó érdekesség különböző hívások esetén. Ehhez tegyük fel, hogy a `q/1` eljárás a következő:

```
q(1).
q(2).
```

Ekkor

- `p(1, Y)` nem hoz létre választási pontot, hiszen a jelöltek halmaza egyelemű: (2)
- `p(s(1), Y)` létrehoz választási pontot, de azt lefutás előtt megszünteti, mert mielőtt redukciós lépést hajtana végre a rendszer a 4. klózon, megpróbálja ugyanezt megtenni a 3. klózzal is. Mivel azzal nem sikerül (nem sikeres a fejillesztés) már csak egy elemre, a 4. klózra szűkül a jelöltek halmaza.
- `p(s(0), Y)` választási pontot hagy a lefutása után is, mert a 4. klózban még mindig potenciális jelöltet lát a rendszer.

A fenti három dolog azért volt érdekes, mert bár mind a három hívás determinisztikus mégis egészen másképpen viselkednek.

3.5.5. Listák szétbontása, variációk appendre

Ebben az alfejezetben az `append/3` eljárás néhány alkalmazását mutatjuk be. Idézzük fel a predikátum definícióját:

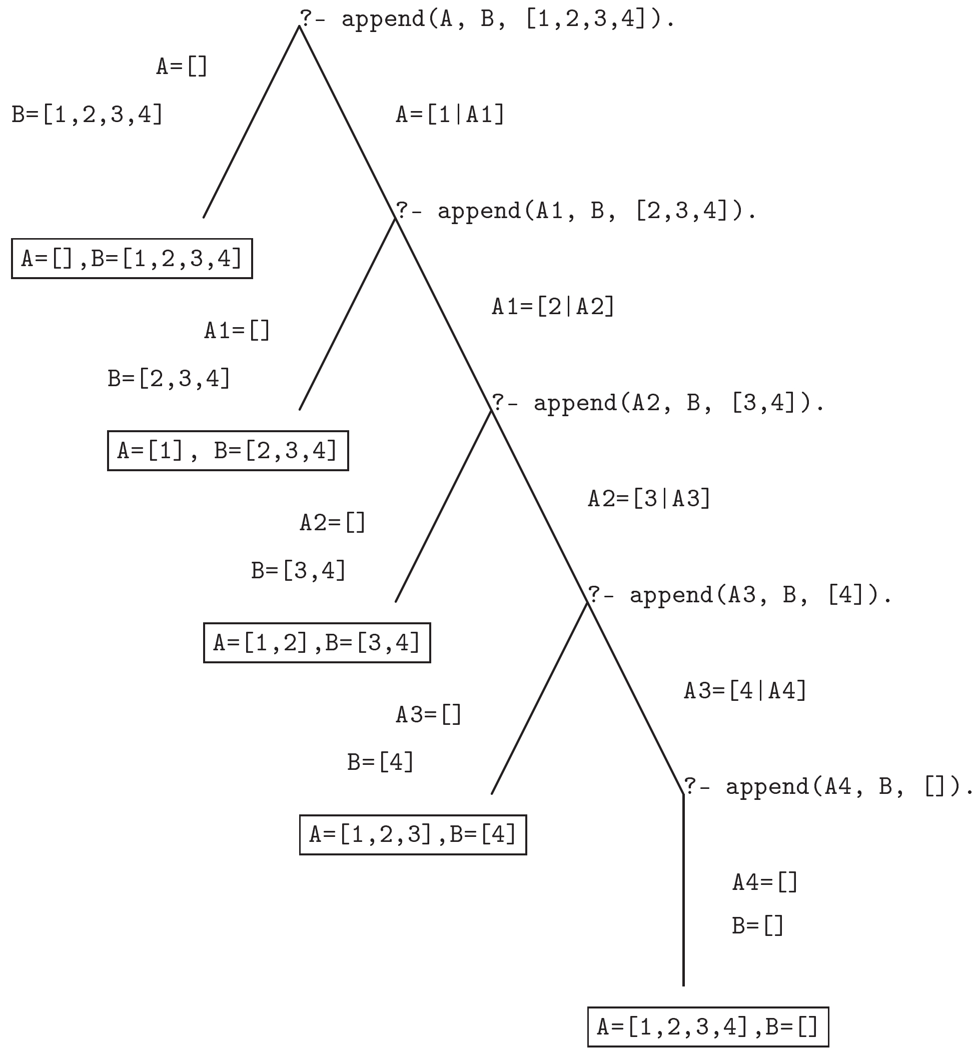
```
% append(L1, L2, L3): Az L3 lista az L1 és L2 listák elemeinek egymás
% után fűzésével áll elő.
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).
```

Az `append/3` eljárás használható ún. *szétszedő* módban is, azaz feltehető kérdésként, hogy egy adott lista hogyan állhat elő más listák összefűzéseként:

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

Azt látjuk, hogy a `[1,2,3,4]` lista öt különféle módon bontható szét két listára.

A fenti kérdés végrehajtását láthatjuk alább, fastruktúraként ábrázolva:



Második alkalmazásként írjunk egy olyan eljárást, amely képes három listát összefűzni. Első ötletünk lehetne a következő:

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

Azaz a feladatot visszavezetjük két `append` hívásra: az első összefűzi az első két listát egy köztes listába, majd a második ehhez fűzzi hozzá a harmadik listát.

Ez a megoldás azonban nem hatékony, például az `append([1,...,100],[1,2,3],[1], L)` 103 helyett 203 lépést igényel. Ráadásul ez a változat szétszedésre sem alkalmas, mert végtelen választási pontot hoz létre, hiszen ilyenkor az első hívás mindhárom argumentuma behelyettesítetlen. Az eljárásunk futása csak akkor biztonságos, ha mindkét `append/3` hívásban az 1. és 3. argumentum legalább egyike zárt végű lista.

Az alábbi megvalósítás amellett, hogy hatékony. szétszedésre is alkalmas:

```
% L1, L2 és L3 összefűzése L123, ahol vagy L1 és L2 vagy L123 adott (zárt végű).
```

```
append2(L1, L2, L3, L123) :-
        append2(L1, L23, L123), append2(L2, L3, L23).
```

Ezen változat működése talán kicsit nehezebben követhető. A megoldás lényege, hogy az első `append2/3` hívás egy nyílt végű listát állít elő. Ezen lista végét tölti fel a második `append2/3` hívás.

Következő feladatként írjunk egy olyan eljárást, amely egy listából kikeresi azon elemeket, amelyek párban fordulnak elő:

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

Maga az eljárás mindössze egyetlen sorból áll:

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

Az `_` olyan változót jelent, amelynek nem fontos az értéke. Ezért a rendszer az `_` minden előfordulását különböző változókkal helyettesíti. A program ennek megfelelően úgy működik, hogy az egyetlen `append/3` hívás minden lehetséges módon megpróbálja összerakni a bemenetként megadott listát úgy, hogy a két részlista közül a második két azonos elemmel kezdődjön.

Zárásként írjunk egy olyan eljárást, amelyet az alábbi fejkomment specifikál:

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

A kódot és egy futási példát láthatunk alább. Az eljárás működésének megértését az olvasóra bízzuk.

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

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

3.6. Legalapvetőbb beépített eljárások

Ebben a fejezetben megpróbáljuk összefoglalni a legfontosabb aritmetikai, programfejlesztési és egyéb beépített eljárásokat.

3.6.1. Aritmetikai beépített eljárások

Szó volt arról, bár elsőre furcsának tűnhet, hogy Prologban `4+2` egy közönséges összetett kifejezés, funktora `+/2`, és semmi köze sincsen a `6` számkonstanshoz. Ez azt is jelenti természetesen, hogy a két dolog nem is egyesíthető egymással.

```
| ?- 4+2=6.
no
| ?-
```

Ezzel szemben a *beépített aritmetikai eljárások* aritmetikai kifejezéseknek tekintik argumentumukat és ennek megfelelően kezelik azokat. Ilyen aritmetikai kifejezésekről és ezek közül is a legfontosabbakról szól ez az alfejezet.

is/2

Az `is/2` beépített eljárással már sokat találkoztunk. Az `X is Kif` hívás a `Kif` aritmetikai kifejezés értékét *egyesíti* `X`-szel. Például $1 * 2 + 3$-at a következőképpen számolhatjuk ki Prologban:

```
| ?- X is 1*2+3.
X = 5 ? ;
no
| ?-
```

A `Kif` Prolog kifejezésnek kötelezően aritmetikai kifejezésnek kell lennie. Ellenkező esetben a rendszer hibát jelez.

```
| ?- A is 4*alma.
! Domain error in argument 2 of is/2
! expected expression, found alma
! goal:  _56 is 4*alma
| ?-
```

Általános tévhit, hogy az `X is X+1` hívás hibás és ilyet nem szabad leírni deklaratív nyelven. Semmi ilyesmiről nincsen szó. Egy `X is X+1` hívás esetén az `is/2` eljárás kiértékeli az `X+1` kifejezést és megpróbálja egyesíteni azt `X`-szel. Ez persze nem sikerülhet, mert ha `X` szám[4] és értéke mondjuk `6`, akkor a rendszer a `6 = 7` egyesítést próbálja meg végrehajtani. Ami nyilván nem sikerül.

aritmetikai relációk

A következő eljárások mind beépítettek a Prolog nyelvben: `Kif1 < Kif2, Kif1 =< Kif2, Kif1 > Kif2, Kif1 >= Kif2, Kif1 =:= Kif2, Kif1 =\= Kif2.`

Jelentésük, hogy a `Kif1` és `Kif2` aritmetikai kifejezések értéke a megadott relációban van egymással. Különösebb megjegyzést csak két eljáráshoz kell fűzni. A `=:=` beépített eljárás kiértékeli mind a bal, mind a jobboldalán található aritmetikai kifejezést és a kapott értékeket egyesíti egymással. Ez megfelel az aritmetikai kifejezések esetén használatos `egyenlő` fogalomnak. A `=\=` beépített eljárás kiértékeli a bal- és jobboldalán álló aritmetikai kifejezéseket és akkor sikerül, ha azok nem egyenlőek.

Ha a fenti beépített eljárások meghívásakor `Kif1, Kif2` valamelyike nem aritmetikai kifejezés, akkor a rendszer hibát jelez.

aritmetikai operátorok

Legfontosabb aritmetikai operátorok: `+, -, *, /, mod, //` (egész-osztás)

Néhány példa az elmondottakra:

```
| ?- X =:= 1*2+3.
    ! Instantiation error in argument 1 of =:= /2
| ?- 1+2*3 > 2*3+1.
no
```

[4]Ha `X` nem szám, akkor eleve hibát jelez a rendszer

3.6.2. Programfejlesztési beépített eljárások

Néhány, a programfejlesztéshez hasznos eljárást ismertetünk itt.

programok betöltése: Erre szolgál a `consult/1` eljárás. A paraméterként megadott Prolog nyelvű állományt (vagy egy listában megadott összes állományt) beolvassa és értelmezendő alakban eltárolja. A `consult` hívásnév elhagyható, a `[file,...].` alak ekvivalens a `consult([file,...]).` alakkal. Ha az állománynév `user`, akkor a rendszer a szabványos inputról, azaz a terminálról várja, hogy begépeljük neki a programot (az állományvége jelet UNIX-on a ctrl+d, Windowson a ctrl+z lenyomásával érhetjük el).

```
| ?- [user].
% consulting user...
| barátja('Verka','Attila').
|
% consulted user in module user, 0 msec 184 bytes
yes
| ?- barátja(A,B).
A = 'Verka',
B = 'Attila' ? ;
no
| ?-
```

predikátumok kilistázása: Erre a `listing/0` vagy `listing/1` eljárások szolgálnak. Előbbi az értelmezendő alakban eltárolt összes, utóbbi a paraméterként adott nevű predikátumokat listázza ki a képernyőre.

programok fordítása: Erre való a `compile/1` eljárás. A `consult/1`-hez hasonló módon a paraméterként megadott állományokban levő programokat beolvassa, lefordítja.

kilépés: Ezt a `halt/0` eljárással tehetjük meg. A Prolog rendszer ekkor befejezi működését. Alternatív módon, UNIX-on a ctrl+d, Windowson a ctrl+z lenyomásával is kiléphetünk.

Végül álljon itt néhány példa az elmondottakra.

```
> sicstus
SICStus 3.9.1 (x86-win32-nt-4): Wed Jun 19 13:03:11  2002
| ?- consult(fakt).
% consulted /home/user/fakt.pl in module user, 0 msec 712 bytes
| ?- listing(fakt).
fakt(0, 1).
fakt(A, B) :-
    A>0,
    C is A-1,
    fakt(C, D),
    B is D*A.
| ?- halt.
>
```

3.6.3. Kiíró és egyéb beépített eljárások

kifejezés kiíratása: Erre szolgál a `write/1` eljárás, amely a paraméterként megadott Prolog kifejezést kiírja a kiválasztott kimenetre (alaphelyzetben a szabványos kimenetre, azaz a képernyőre). Figyelembe veszi az operátorokat is.

```
| ?- write(+(1,*(2,3))).
1+2*3
yes
| ?-
```

kifejezések alapstruktúra-alakjának kiíratása: Erre való a `write_canonical/1` beépített eljárás. A paraméterként kapott Prolog kifejezést alapstruktúra alakban írja ki a képernyőre.

```
| ?- write_canonical(1*2+3-5).
-(+(*(1,2),3),5)
yes
| ?-
```

újsor: Az `nl/0` eljárás kiír egy újsort.

true, fail A `true/0` eljárás mindig sikerül, a `fail/0` mindig meghiúsul.

```
| ?- szülője('István', X), write(X), nl, fail.
Géza
Sarolt
no
| ?-
```

Láthatjuk, hogy *természetesen* nem történt változóbehelyettesítés, hiszen célsorozat az utolsó `fail/0` hívás miatt meghiúsult. Az, hogy mégis látjuk kiírva Gézát és Saroltot annak köszönhető, hogy a `write/1` eljárás ún. *mellékhatásos* eljárás.

nyomkövetés: A nyomkövetőt a `trace` hívással kapcsoljuk be és a `notrace` hívással kapcsoljuk ki. A Erre láthatunk példát az alábbiakban:

```
| ?- trace.
% The debugger will first creep -- showing everything (trace)
yes
% trace
| ?- notrace.
% The debugger is switched off
yes
| ?-
```

Amennyiben nyomkövetés módban vagyunk a célsorozat "lépésről-lépésre" fut le. Töréspontot a `spy/1` eljárással helyezhetünk el egy predikátumon és a `nospy/1` hívással szedhetjük le róla. Példaként helyezzünk el töréspontot a `próba` nevű, 1 argumentummal rendelkező eljárásra:

```
| ?- spy(próba/1).
% The debugger will first zip -- showing spypoints (zip)
% Plain spypoint for user:próba/1 added, BID=1
yes
% zip
| ?-
```

3.7. Példák, negáció, feltételes szerkezet

Ebben a fejezetben néhány, az eddigiekhez képest nagyobb lélegzetvételű feladatot tűzünk ki célul és mutatjuk be Prolog nyelvű megoldásukat. Eközben ismertetjük a *negáció* fogalmát és az eddigieknél részletesebben szólunk a feltételes kifejezésekről.

Az első példánk egy útvonalkeresási feladat, amelyet többféleképpen is megoldunk. Második feladatként együtthatókat határozunk meg lineáris kifejezésekben. A fejezetet végül két, kisebb példával zárjuk.

3.7.1. Az útvonalkeresési feladat, negáció

A feladatot a következőképpen fogalmazhatjuk meg: tekintsük (autóbusz)járatok egy halmazát. Mindegyik járathoz a két végpont és az útvonal hossza van megadva. Írjunk Prolog eljárást, amellyel megállapítható, hogy két pont összeköthető-e pontosan N csatlakozó járattal, valamint adjuk meg ezen N szakaszból álló útvonal összhosszát! N bemenő argumentum.

Legyenek adottak továbbá az alábbi tényállítások, amelyek azt állítják, hogy létezik buszjárat Budapest és Prága között (és a járat hossza 515km), Budapest és Bécs között stb.

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

Feladatunk megoldásában segíthet, ha látunk egy példafutást. Azt a kérdést tesszük fel, hogy vajon létezik-e 2 hosszú útvonal Párizsból Budapestre és ha igen, mennyi annak az összhossza?

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

Látható, hogy ilyen útvonal létezik és az összhossz 1510km. Ez a Párizs-Bécs-Budapest útszakasznak felel meg. Ez annak ellenére igaz, hogy olyan tényállítás, amely például Bécs-Budapest járatot írna le nincsen. Azaz a tényállítások kétirányúak, így valószínűleg szükségünk lesz az alábbihoz hasonló eljárásra:

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

Azaz A-ból B-be eljuthatunk busszal, ha vagy A-ból B-be vagy B-ből A-ba létezik buszjárat.

Ezek után lássuk a megoldásunkat:

```
% útvonal(N, A, B, H): A és B között van (pontosan)
% N szakaszból álló útvonal, amelynek összhossza H.
útvonal(0, Hová, Hová, 0).
útvonal(N, Honnan, Hová, H) :-
    N > 0,
    N1 is N-1,
    útszakasz(Honnan, Közben, H1),
    útvonal(N1, Közben, Hová, H2),
    H is H1+H2.
```

Az `útvonal/4` eljárás a következőképpen működik. Az első klóz azt állítja, hogy 0 darab szakaszból álló úton egy `A` városból eljuthatunk az `A` városba 0km megtételével. Azaz nem megyünk sehova sem. A második állítás egy szabály, eszerint ha `N > 0` szakaszból álló utat keresünk, a kezdőpontunkból van egy útszakasz egy `Közben`-be, és innen egy `N1 = N-1` szakaszból álló útvonallal eljuthatunk a célunkhoz, akkor létezik egy `N` szakaszból álló útvonal kezdő- és célpontunk között, amelynek hossza a szakasz és a folytatás-útvonal hosszának összege.

Körmentes útvonal keresése

Sajnos megoldásunk nem tökéletes. Ha ugyanis azt a kérdést tesszük fel, hogy „létezik-e 2 hosszú útvonal Párizsból valahova?", akkor az alábbi választ kapjuk a Prolog rendszertől:

```
| ?- útvonal(2, 'Párizs', Hová, H).
H = 1900, Hová = 'Berlin' ? ;
H = 2530, Hová = 'Párizs' ? ;
H = 1510, Hová = 'Budapest' ? ;
no
```

Látjuk, hogy 3 megoldásunk van. Az első és a harmadik nem meglepő, de a második megoldás talán igen. Ez azt állítja, hogy Párizsból két lépésből álló úton eljuthatunk Párizsba. Ez persze igaz, de mi szeretnénk egy olyan „útvonaltervezőt", amelyik kizárja a köröket.

Módosított programunk így nézhet ki (további szolgáltatásként ez a változat a tényleges útvonalat is visszaadja):

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

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

% útvonal_2(N, A, B, K, Út, H): A és B között van pontosan
% N szakaszból álló körmentes, K elemein át nem menő H hosszú Út út.
útvonal_2(0, Hová, Hová, Kizártak, Út, 0) :-
    reverse(Kizártak, Út).
útvonal_2(N, Honnan, Hová, Kizártak, Út, H) :-
    N > 0, N1 is N-1,
    útszakasz(Honnan, Közben, H1),
    \+ member(Közben, Kizártak),
    útvonal_2(N1, Közben, Hová, [Közben|Kizártak], Út, H2),
    H is H1+H2.
```

Elsőként betöltjük a `lists` könyvtárat és *importálunk* belőle két eljárást. Ezeket felhasználjuk a megoldásban. Programunk két eljárást tartalmaz, az `útvonal_2/5`-t és az `útvonal_2/6`-t. Ez utóbbi két klózból áll, kezdjük az ismerkedést ezzel az eljárással.

Ez az eljárás működését tekintve nagyon hasonlít az `útvonal/4`-re. Az első klóz foglalkozik azzal, hogy 0 darab szakaszból álló úton eljuthatunk önmagunkba, valamint ez a klóz adja vissza a tényleges útvonalat (ez nem más, mint a Kizártak lista megfordítottja) is. A második klóz azt állítja, hogy ha egy `N > 0` szakaszból álló utat keresünk, a kezdőpontunkból van egy útszakasz egy `Közben`-be úgy, hogy `Közben` nincsen a `Kizártak` között, és innen egy `N1 = N-1` szakaszból álló útvonallal (mely többet már nem mehet át `Közben`-en) eljuthatunk a célunkhoz, akkor létezik egy `N` szakaszból álló körmentes útvonal kezdő- és célpontunk között, amelynek hossza a szakasz és a folytatás-útvonal hosszának összege.

A már meglátogatott városokat a `Kizártak` nevű listában tároljuk. Ez a lista folyamatosan bővül, mindig eléje fűzzük az aktuális `Közben` várost. Azt, hogy egy város nincsen benne a listában a `\+ member` szerkezettel vizsgáljuk, erről hamarosan részletesebben lesz szó.

`útvonal_2/5` eljárás feladata mindösszesen az, hogy inicializálja a `Kizártak` listát. Ez kezdetben egy egyelemű lista, amelynek egyetlen eleme `Honnan`.

Negáció

Térjünk akkor rá a \+ ismertetésére. A `(\+)/1` egy (beépített) eljárás, jelentése az, hogy „nem bizonyítható". Egyetlen argumentumának egy meghívható kifejezésnek kell lennie, azaz például az alábbi esetben a rendszer hibát jelez:

```
| ?- \+ 4.
! Type error in (\+)/1
! callable expected, but 4 found
! goal:  user:(\+4)
| ?-
```

A \+ végrehajtja az argumentumában kapott hívást. Ha ez sikeresen fut le, \+ meghiúsul, egyébként sikerül. Ilyen esetben sem történik változóbehelyettesítés. $\backslash + H$ matematikai jelentése a következő: $\neg\exists X(H)$, ahol X a H-ban a *hívás pillanatában* behelyettesítetlen változókat jelöli.

Eddig még nem esett szó a \= beépített operátorról. Ez definíció szerint:

```
X \= Y :-
    \+ X = Y.
```

Jelentése: az argumentumok nem egyesíthetőek.

Lássunk néhány példát (feltesszük, hogy érvényesek a „szokásos” gyermek-szülő kapcsolatokat leíró `szülője` tényállítások)!

```
| ?- \+ szülője('Imre', X).            ----> no
| ?- \+ szülője('Géza', X).            ----> true ?
| ?- \+ X = 1, X = 2.                  ----> no
| ?- X = 2, \+ X = 1.                  ----> X = 2 ?
```

Az első példában azt kérdezzük, hogy igaz-e, hogy nem bizonyítható, hogy Imrének van szülője. Ez nyílván nem igaz, hiszen bizonyítani tudjuk, hogy Imrének van szülője, van ilyen tényállítás. A második kérdés megegyezik az elsővel, csak Gézára vonatkozik. Ebben az esetben nem tudjuk bizonyítani, hogy Gézának lenne akár csak egy szülője is, mert nincsen ilyen tényállítás, ami erről szólna. Ez az ún. *zárt-világ feltételezés*, amely azt mondja ki, hogy ami nem bizonyítható, az nem igaz. A harmadik és negyedik példa szorosan összefügg. A harmadik példa azt mutatja be, hogy amíg egy változónak nincsen értéke, addig a \+ hívás meghiúsul (hiszen az egyesítés természetes sikerül).

Végül definiáljuk a `testvére` eljárást, amely azt mondja ki, hogy ha `T1` és `T2` szülője ugyanaz az `A` személy és `T1` nem ugyanaz, mint `T2`, akkor `T1` és `T2` testvérek:

```
testvére(T1, T2) :-
    szülője(T1, A),
    szülője(T2, A),
    T1 \= T2.
```

Útvonalkeresés gráfokkal

Térjünk vissza az útvonalkeresési feladatunkhoz. Az olvasó bizonyára észrevette a városok közötti útvonalkeresés valójában egy irányítatlan gráfban való útkeresést jelent. Módosítsuk tehát annyiban a megoldásunkat, hogy a meglévő buszjáratokat ne tényállítások formájában tároljuk, hanem egy súlyozott gráfot leíró adatstruktúrában (a súlyok az útvonal-hosszak). Ekkor nyílvánvalóan a programot is módosítanunk kell.

A gráfot a következőképpen ábrázolhatjuk:

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

Ezt a következő típusdefiníciós-kommenttel írhatjuk le:

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

Ennek megfelelően az alábbi kifejezés egy gráfot ír le:

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

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

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

% útvonal_3(N, G, A, B, L, H): A G gráfban van egy  A-ból
% B-be menő N szakaszból álló L út, amelynek összhossza H.
útvonal_3(0, _Gráf, Hová, Hová, [Hová], 0).
útvonal_3(N, Gráf, Honnan, Hová, [Honnan|Út], H) :-
        N >0, N1 is N-1,
        select(Él, Gráf, Gráf1),
        él_végpontok_hossz(Él, Honnan, Közben, H1),
        útvonal_3(N1, Gráf1, Közben, Hová, Út, H2),
        H is H1+H2.

% él_végpontok_hossz(Él, A, B, H): Az Él irányítatlan él
% végpontjai A és B, hossza H.
él_végpontok_hossz(él(A,B,H), A, B, H).
él_végpontok_hossz(él(A,B,H), B, A, H).
```

3.7.2. Feltételes kifejezések

Ebben az alfejezetben az eddigieknél részletesebben beszélünk a feltételes kifejezésekről. A mondandó illusztrálásához először egy példát oldunk meg. Feladatunk egy lineáris kifejezésben az `x` változó együtthatójának meghatározása. Precízebben fogalmazva, a kifejezés számokból és az 'x' névkonstansból '+' és '*' operátorokból épül fel. Ennek megfelel az alábbi típusdeklaráció:

```
% :- type kif == {x} \/ number \/ {kif+kif} \/ {kif*kif}.}
```

A kifejezésnek lineárisnak kell lennie, ami azt jelenti, hogy a '*' operátor legalább egyik oldalán szám áll.
A programkód a következő:

```
% egyhat(Kif, E): A Kif lineáris formulában az x együtthatója E.
egyhat(x, 1).
egyhat(Kif, E) :-
    number(Kif), E = 0.
egyhat(K1+K2, E) :-
    egyhat(K1, E1),
    egyhat(K2, E2),
    E is E1+E2.
```

```
egyhat(K1*K2, E) :-
    number(K1),
    egyhat(K2, E0),
    E is K1*E0.
egyhat(K1*K2, E) :-
    number(K2),
    egyhat(K1, E0),
    E is K2*E0.
```

Az `egyhat` eljárás a következőképpen működik. Az első klóz azt állítja, hogy `x` együtthatója `1`. A második azt, hogy egy szám együtthatója `0`. A harmadik klóz `K1+K2` alakú kifejezésekről szól, és azt állítja, hogy ha `x` együtthatója `K1`-ben `E1`, valamint `K2`-ben `E2`, akkor a teljes `K1+K2` kifejezésben `x` együtthatója `E1+E2`. Az utolsó két klóz hasonló logikát követ `K1*K2` alakú kifejezések esetén.

Az alábbiakban néhány példafutást láthatunk:

```
| ?- egyhat(((x+1)*3)+x+2*(x+x+3), E).
E = 8 ? ;
no
| ?- egyhat(2*3+x, E).
E = 1 ? ;
E = 1 ? ;
no
```

Az első esetben minden rendben, könnyen ellenőrízhető, hogy az eredmény helyes. A második esetben azonban kétszer kaptuk meg az `1` megoldást és ez semmiképpen sem elfogadható viselkedés.

A problémát az okozza, hogy a `K1*K2` alakú kifejezéseket kezelő klózok (a két utolsó) „nem zárják ki egymást”, azaz például a `2*3` kifejezés mindkét klózzal redukálható. Így az `egyhat/2` eljárás az ilyen esetben többszörös megoldást ad:

```
| ?- egyhat(2*3, E).
E = 0 ? ;
E = 0 ? ;
no
```

Ennek kiküszöböléséhez többféleképpen állhatunk neki. Használhatunk például negációt

```
(...)
egyhat(K1*K2, E) :-
        number(K1),
        egyhat(K2, E0),
        E is K1*E0.
egyhat(K1*K2, E) :-
        \+ number(K1),
        number(K2),
        egyhat(K1, E0),
        E is K2*E0.
```

vagy akár feltételes kifejezést is:

```
(...)
egyhat(K1*K2, E) :-
        (
            number(K1) -> egyhat(K2, E0), E is K1*E0
        ;
            number(K2), egyhat(K1, E0), E is K2*E0
        ).
```

Ez utóbbi nem más, mint egy „if-then-else” szerkezet. A következőképpen kell értelmezni: *ha* `K1` szám, *akkor* `K2` együtthatója `E0` és `E is K1*E0`, **`különben`** `K2` szám stb. Általában egy ilyen szerkezet az alábbi módon épül fel:

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

Egy „ha-akkor-különben” szerkezetnek megadhatjuk mind a deklaratív, mind a procedurális szemantikáját. Következzen most először a deklaratív szemantika: a fenti alak jelentése megegyezik az alábbival, ha a `felt` egy egyszerű feltétel (nem oldható meg többféleképpen):

```
(...) :-
        (...),
        (
            felt, akkor
        ;
            \+ felt, egyébként
        ),
        (...).
```

Amennyiben `felt` többféleképpen is megoldható, akkor nem adható meg deklaratív szemantika az „if-then-else” szerkezethez.

Az alábbiakban megadjuk a `(felt->akkor;egyébként),folytatás` célsorozat végrehajtásának procedurális jelentését. Itt nem kell kikötni azt, hogy `felt` egyszerű feltétel legyen.

- Végrehajtjuk a `felt` hívást.
- Ha `felt` sikeres, akkor az `akkor,folytatás` célsorozatra redukáljuk a fenti célsorozatot, a `felt` *első* megoldása által eredményezett behelyettesítésekkel. A `felt` cél többi megoldását nem keressük meg.
- Ha `felt` sikertelen, akkor az `egyébként,folytatás` célsorozatra redukáljuk, behelyettesítés nélkül.

Lehetőségünk van többszörös elágaztatás létrehozására is. Ilyenkor skatulyázott feltételes kifejezéseket használunk:

```
        (   felt1 -> akkor1
        ;   felt2 -> akkor2
        ;   ...
        )
```

Az egyébként rész elhagyható, alapértelmezése: `fail`.

Az elmondottakat szemlélteti az alábbi példa:

```
% Num szám előjele Sign
sign(Num, Sign) :-
        (   Num > 0 -> Sign = 1
        ;   Num < 0 -> Sign = -1
        ;   Sign = 0
        ).
```

```
| ?- sign(5,S).
S = 1 ? ;
no
| ?- sign(-2,S).
S = -1 ? ;
no
| ?- sign(0,S).
S = 0 ? ;
no
| ?-
```

Végül nézzük meg a jól ismert faktoriális-számító program felételes kifejezést használó változatát:

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

Vegyük észre, hogy a fenti programban (1) helyére írhattunk volna `N = 0, F = 1`-t is, anélkül, hogy változott volna az eljárás jelentése. Ezt az `N > 0` feltétel miatt tehetjük meg. Ennek ellenére érdemes használni a `(felt->akkor;egyébként)` szerkezetet, mert ez nem hoz létre választási pontot — így ez hatékonyabb, mint a sima diszjunkciós alak.

Feltételes kifejezések és a negáció kapcsolata

A `\+ felt` negáció kiváltható a `( felt -> fail ; true)` feltételes kifejezéssel.

Példaként ellenőrizzük, hogy egy adott kifejezés nem eleme egy listának (pontosabban nem egyesíthető a lista egyik elemével sem). Ezt a fent elmondottaknak megfelelően így tehetjük meg:

```
nem_eleme(E, L) :-
        (   member(E, L) -> fail
        ;   true
        ).
```

Szóban ez azt jelenti, hogy ha `E` eleme a listának akkor meghiúsulunk, különben sikeresen visszatérünk. Változóbehelyettesítés csak a `member` hívás kapcsán történhet, de a meghiúsulás miatt ezeknek nem lesz nyoma.

Szó volt róla, hogy a `(\=)/2` operátor jelentése az, hogy az argumentumok nem egyesíthetőek. Ezen operátor felhasználásával is megírhatjuk a `nem_eleme/2` eljárásunkat:

```
nem_eleme(E, []).
nem_eleme(E, [X|L]) :-
        E \= X,
        nem_eleme(E, L).
```

Az első klóz azt állítja, hogy `E` nem eleme az üres listának. A második azt, hogy ha `E` nem egyesíthető a lista fejével és `E` nem eleme a lista farkának, akkor `E` nem eleme az egész listának sem.

3.8. A Prolog szintaxis

Az eddigiekben megismerkedtünk a Prolog nyelv közelítő szintaxisával, megadtuk a Prolog programok elemeinek (eljárás, szabály, cél stb.) szintaxisát, leírva hogy ezek hogyan épülnek fel Prolog kifejezésekből a

'`:-`', '`,`', '`;`' stb. összekötő jelek segítségével. Tudjuk például, hogy egy szabály fejből és törzsből áll, ahol a fej egy Prolog kifejezés, a törzs Prolog kifejezések '`,`'-vel elválasztott sorozata, míg a fejet a törzzsel a '`:-`' jel köti össze. Ha visszaemlékszünk láthatjuk, hogy a Prolog kifejezés fogalmát mintegy építőkövet használtuk arra, hogy „bonyolultabb” strukúrákat hozzunk létre:

⟨Prolog program⟩	::=	⟨predikátum⟩ . . .	
⟨predikátum⟩	::=	⟨klóz⟩ . . .	{azonos funktorú}
⟨klóz⟩	::=	⟨tényállítás⟩.␣ |	
		⟨szabály⟩.␣	
⟨tényállítás⟩	::=	⟨fej⟩	
⟨szabály⟩	::=	⟨fej⟩ :- ⟨törzs⟩	
⟨törzs⟩	::=	⟨cél⟩, . . .	
⟨cél⟩	::=	⟨kifejezés⟩	
⟨fej⟩	::=	⟨kifejezés⟩	

Most már megmondhatjuk, hogy eddig egy kicsit csaltunk, mert a Prolog nyelvben valójában *minden* programelem Prolog kifejezés.

Ez azért van így (azért lehet így), mert a használt összekötő jelek mind szabványos operátorok. A programszöveg beolvasott kifejezéseit a (legkülső) funktoruk alapján osztályozzuk mint szabályt, tényállítást stb. Ez például azt jelenti, hogy egy `p :- q.` szabályt programjainkban írhatunk akár `:-(p, q).` alakban is, ami láthatóan egy közönséges összetett kifejezés!

Az alábbiakban megadjuk, hogy egy Prolog kifejezést a (legkülső) funktora alapján hogyan osztályozhatunk, valamint azt, hogy a Prolog rendszer mit tesz ilyen kifejezések beolvasásakor.

- *kérdés*

  alakja: `?- Cél.`
  jelentése: *Célt* lefuttatja, és a változó-behelyettesítéseket kiírja (ez az alapértelmezés az ún. top-level interaktív felületen). A jegyzetben számos helyen találkozhatunk vele. Itt `Cél` nem más mint Prolog kifejezések '`,`'-vel elválasztott sorozata. Példa `Cél`-ra a következő célsorozat: `sum_tree3(3--2,A), write(A).`

- *parancs*

  alakja: `:- Cél.`
  jelentése: A *Célt* csendben lefuttatja. Pl. deklaráció (operátor, . . .) elhelyezésére. Parancsot láthatunk például az útvonalkereső programunk második változatánál — így töltöttük be ugyanis a `lists` könyvtárat

- *szabály*

  alakja: `Fej :- Törzs.`
  jelentése: A szabályt felveszi a programba.

- *nyelvtani szabály*

  alakja: `Fej --> Törzs.`
  jelentése: Prolog szabállyá alakítja és felveszi (lásd a DCG nyelvtan).

- *tényállítás*

  alakja: `Minden egyéb kifejezés.`
  jelentése: Üres törzsű szabályként felveszi a programba.

A nyelvtani szabályokról a későbbiekben lesz szó.

3.8.1. Kifejezések szintaxisa

A Prolog nyelv szintaxisát ún. *kétszíntű* nyelvtannal fogjuk megadni. Ahelyett, hogy részletekbe menően ismertetnénk, hogy mi is az, megadunk egy részletet egy „hagyományos” nyelv kifejezés-szintaxisából, majd megadjuk ugyanezt kétszíntű nyelvtannal.

⟨kifejezés⟩ ::=		⟨tag⟩
	|	⟨kifejezés⟩ ⟨additív művelet⟩ ⟨tag⟩
⟨tag⟩ ::=		⟨tényező⟩
	|	⟨tag⟩ ⟨multiplikatív művelet⟩ ⟨tényező⟩
⟨tényező⟩ ::=		⟨szám⟩ | ⟨azonosító⟩ | (⟨kifejezés⟩)

Ugyanez kétszíntű nyelvtannal megadva (az additív ill. multiplikatív műveletek prioritása 2 ill. 1):

⟨kifejezés⟩ ::=		⟨kif 2⟩
⟨kif `N`⟩ ::=		⟨kif `N-1`⟩
	|	⟨kif `N`⟩ ⟨`N` prioritású művelet⟩ ⟨kif `N-1`⟩
⟨kif `0`⟩ ::=		⟨szám⟩ | ⟨azonosító⟩ | (⟨kif 2⟩)

A Prolog nyelv szintaxisa kétszíntű nyelvtannal a következőképpen írható le:

⟨programelem⟩ ::=		⟨kifejezés 1200⟩ ⟨záró-pont⟩
⟨kifejezés `N`⟩ ::=		⟨op `N` fx⟩ ⟨köz⟩ ⟨kifejezés `N-1`⟩
	|	⟨op `N` fy⟩ ⟨köz⟩ ⟨kifejezés `N`⟩
	|	⟨kifejezés `N-1`⟩ ⟨op `N` xfx⟩ ⟨kifejezés `N-1`⟩
	|	⟨kifejezés `N-1`⟩ ⟨op `N` xfy⟩ ⟨kifejezés `N`⟩
	|	⟨kifejezés `N`⟩ ⟨op `N` yfx⟩ ⟨kifejezés `N-1`⟩
	|	⟨kifejezés `N-1`⟩ ⟨op `N` xf⟩
	|	⟨kifejezés `N`⟩ ⟨op `N` yf⟩
	|	⟨kifejezés `N-1`⟩
⟨kifejezés 1000⟩ ::=		⟨kifejezés 999⟩ , ⟨kifejezés 1000⟩
⟨kifejezés 0⟩ ::=		⟨név⟩ (⟨argumentumok⟩)
	|	(⟨kifejezés 1200⟩) | { ⟨kifejezés 1200⟩ }
	|	⟨lista⟩ | ⟨füzér⟩
	|	⟨név⟩ | ⟨szám⟩ | ⟨változó⟩

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

Általános megjegyzések a fentiekkel kapcsolatban:

- Fontos, hogy A ⟨név⟩ és a `(` közvetlenül egymás után kell, hogy álljon a ⟨kifejezés 0⟩ definíciójában.
- A ⟨kifejezés N⟩-ben ⟨köz⟩ csak akkor kell ha az őt követő kifejezés nyitó-zárójellel kezdődik.

3.8.2. Lexikai elemek

A Prolog kétszintű nyelvtanában szereplő ⟨név⟩, ⟨változó⟩, ⟨természetes szám⟩ és ⟨lebegőpontos szám⟩ kifejezések lexikai vonatkozását adjuk itt meg. Ezek egyrésze már ismerős lehet, hiszen a fejezet elején, a Prolog közelítő szintaxisának ismertetésekor, már kitértünk erre.

⟨név⟩

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

A ⟨változó⟩ egy nagybetűvel vagy aláhúzással kezdődő alfanumerikus jelsorozat. Az azonos jelsorozattal jelölt változók egy klózon belül azonosaknak, különböző klózokban különbözőeknek tekintődnek. Kivételt képeznek a semmis változók (`_`). Ezek minden előfordulása különböző változót jelöl.

⟨természetes szám⟩

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

A ⟨lebegőpontos szám⟩ mindenképpen tartalmaz tizedespontot, mindkét oldalán legalább egy (decimális) számjegy áll, valamint `e` vagy `E` betűvel jelzett esetleges exponenst tartalmazhat.

3.8.3. Megjegyzések és formázó-karakterek

Megjegyzéseket Prologban kétféleképpen írhatunk. Az egyik esettel már sokszor találkoztunk a jegyzetben, ilyenek voltak az eljárások elé írt fejkommentek. Ezek a `%` százalékjeltől a sor végéig tartanak, több soros megjegyzés esetén minden sor elejére ki kell tenni a százalékjelet. A megjegyzések készítésének másik módja a `/* */` jelpáros használata, amely a C nyelvből már ismerős lehet.

Egy programot általában formázottan írunk, azaz szeretünk olvashatóan, „szépen” programozni. Ehhez tudni kell, hogy milyen formázó karakterket, formázó elemeket használhatunk és hol. Prologban formázó elemnek számít a szóköz, az újsor, a tabulátor és minden nem látható karaker. Formázó elem ezenkívül a megjegyzés is.

Egy Prolog programban formázó elemek szabadon elhelyezhetőek, azaz nem számít, hogy hány darab szóköz, újsor van egy adott helyen.[5] Néhány dologra azonban oda kell figyelni, ezek a következők:

- struktúrakifejezés neve után nem szabad formázó elemet tenni
- prefix operátor és `(` közé kötelező formázó elemet tenni
- a kétszintű nyelvtanban szereplő ⟨záró-pont⟩ egy olyan `.` karakter amit (legalább) egy formázó elem követ

Az alábbiakban néhány tanácsot adunk Prolog programok javasolt formázásához.

Az egy predikátumhoz tartozó klózokat (azonos funktorú klózok) folyamatosan, üres sor közbeiktatása nélkül írjuk. A predikátumok közé egy üres sort teszünk.

A klóz fejét a sor elején kezdjük, a törzset néhány szóközzel beljebb. A fej és a `:-` nyakjel közé rakunk egy szóközt, ugyanígy szóközt rakunk a törzsbeli hívásokat elválasztó vesszők és a hívások argumentumait elválasztó vesszők után. Viszont nem rakunk szóközt a listaelemeket, ill. a struktúrák argumentumait elválasztó vesszők után.

A diszjunkciót és a feltételes kifejezést mindenképpen zárójelbe tesszük, úgy, hogy a nyitó zárójel, az alternatívákat kezdő pontosvesszők és a végzárójel pontosan egymás alá kerüljön. Az alternatívák célsorozatait néhány szóközzel beljebb kezdjük.

Példa:

```
él_végpontok_hossz(él(A,B,H), A, B, H).
él_végpontok_hossz(él(A,B,H), B, A, H).

párban(L, E) :-
        append(_, [E,E|_], L).

member(Elem, [Fej|Farok]) :-
        (   Elem = Fej
        ;   member(Elem, Farok)
        )
```

Mindenképpen tilos szóközt rakni az eljárás- ill. struktúranév után!

Az egyszer előforduló változókat egyetlen aláhúzásjellel jelöljük, vagy aláhúzásjellel kezdjük. Ugyanannak a „mennyiségnek” a különböző állapotait rendre a 0, 1, ... változó-végződéssel különböztetjük meg, a végállapot végződés nélküli:

```
hozzaado([E|L], Ford0, Ford) :-
        hozzaad(E, Ford0, Ford1),
        hozzaado(L, Ford1, Ford).
```

[5] Léteznek olyan nyelvek is (például a Python), ahol olyannyira kötött egy program szerkezete, hogy például adott formázás adott vezérlési szerkezetet jelent.

3.8.4. Prolog nyelv-változatok

A SICStus Prolog rendszernek két üzemmódja van. Az `iso` az ISO Prolog szabványnak megfelelő, míg a `sicstus` a korábbi változatokkal kompatibilis. Az alapértelmezett mód a `sicstus`. A mód állítására a `set_prolog_flag/2` eljárás szolgál, például így váltunk át `iso` módra:

```
set_prolog_flag(language,iso).
```

A `set_prolog_flag/2` eljárás számos, a rendszer működését befolyásoló beállítás elvégzésére képes, az első argumentumában kapja meg névkonstans formájában a módosítani kívánt belső jellemzőt, ami jelen esetben `language`.

A két mód közötti különbség kiterjed szintaxis részletekre, a beépített eljárások viselkedésének kisebb eltéréseire. A jegyzetben eddig ismertetett eljárások hatása lényegében azonos a két módban.

3.8.5. Szintaktikus édesítőszerek - gyakorlati tanácsok

Az alábbiakban pontokba szedett gyakorlati tanácsokat adunk meg szintaktikus édesítőszerek használatával kapcsoltban. Elsőként megmutatjuk, hogy hogyan érdemes operátoros kifejezéseket alapstruktúra alakra hozni, majd ugyanerről beszélünk listák esetében is, végül néhány egyéb érdekességről szólunk.

Operátoros kifejezések alapstruktúra alakra hozása

- Zárójelezzük be a kifejezést, az operátorok prioritása és fajtája alapján. Például a `-a+b*2` kifejezés bezárójelezett alakja `((-a)+(b*2))`. Ehhez tudni kellett, hogy a `-` operátor `sicstus` módban `500 fx` és hogy a '`*`' kisebb prioritású, mint a '+'.
- Hozzuk az operátoros kifejezéseket alapstruktúra alakra. Például egy `(A Inf B)` infix operátoros kifejezésből `Inf(A,B)` lesz, prefix operátoros kifejezés esetén, mint amilyen a `(Pref A)`, `Pref(A)` lesz. Végül `(A Postf)` esetben `Postf(A)` az átalakított forma. Előbbi példánkat a következőképpen alakíthatjuk tehát át:
  `((-a)+(b*2))` ⇒ `(-(a)+ *(b,2))` ⇒ `+(-(a),*(b,2))`.
- Trükkös esetek:
  - Ha a vesszőt névként akarjuk használni, akkor ezt a jelet idézni kell, mint például akkor, ha a `(pp,(qq;rr))` kifejezést szeretnénk alapstruktúra alakra hozni:
    `','(pp,;(qq,rr))`
  - Jegyezzük meg, hogy `-` ***Szám*** egy negatív számkonstanst jelöl, de `-` ***Egyéb*** egy prefix operátoros kifejezés. Például `-1+2` alapstruktúra alakja `+(-1,2)`, ellenben `-a+b`-nek már `+(-(a),b)`, mint ahogy már feljebb is láthattuk.
  - Egy ***Név***`(...)` kifejezés egy közönséges összetett kifejezés (a zárójelek előtt nincsen szóköz). Ezzel szemben egy ***Név*** `(...)` már egy prefix operátoros kifejezést. Például `-(1,2)` alapstruktúra alakja `-(1,2)` (azaz változatlan), ugyanakkor `- (1,2)` alapstruktúra alakja `-(',')(1,2))`.

Listák alapstruktúra alakra hozása

- Egy `[Elem1,Elem2,..,Elemn]` lista azonos az `[Elem1,Elem2,...Elemn|[]]` listával. Például `[1,2]` nem más, mint `[1,2|[]]`, illetve `[[X|Y]]` megfelel `[[X|Y]|[]]`-nek.
- Egy `[Elem1,Elem2...]` listából a következő módon küszöbölhejük ki a legelső vesszőt: `[Elem1|[Elem2...]]`. Ezt az eljárást folytatva teljesen eltüntethetjük a vesszőket. Lássunk két példát:
  `[1,2|[]]` ⇒ `[1|[2|[]]]`
  `[1,2,3|[]]` ⇒ `[1|[2,3|[]]]` ⇒ `[1|[2|[3|[]]]]`

- Egy `[Fej|Farok]` listát a következőképpen alakítunk át struktúrakifejezéssé: `.(Fej,Farok)`. Két gyors példa:
  `[1|[2|[]]]` ⇒ `.(1,.(2,[]))`
  `[[X|Y]|[]]` ⇒ `.(.(X,Y),[])`

Egyéb szintaktikus édesítőszerek

- Karakterkód-jelölésére a `0'Kar` szerkezet szolgál.
  `0'a` ⇒ `97`, `0'b` ⇒ `98`, `0'c` ⇒ `99`, `0'd` ⇒ `100`, `0'e` ⇒ `101`
- Egy füzér vagy másnéven string, mint például a `"xyz..."` nem más, mint az `xyz...` karakterek kódját tartalmazó lista. Azaz például

  ```
  | ?- A="abc".
  A = [97,98,99] ? ;
  no
  | ?-
  ```

  és ugyanígy `"" = []`, illetve `"e" = [101]`.
- A kapcsos zárójelezés is egy szintaktikus édesítőszer. `{Kif}` megegyezik `{}(Kif)`-vel, ami egy `{}` nevű, egyargumentumú struktúra. A `{}` jelpár egy önálló lexikai elem, egy névkonstans. Ugyanolyan mint például bármelyik név, ahogyan elnevezzük eljárásainkat.
- Bináris, oktális, hexadecimális stb alakot (csak `iso` módban) jelölhetünk a következőképpen is: `0b101010, 0o52, 0x1a` stb.

3.9. Típusok Prologban

Sokszor említettük már, hogy a Prolog típustalan nyelv. Ennek ellenére az eljárások csak bizonyos adathalmazokon képesek dolgozni, így implicit módon ugyan, de megjelenik a típusfogalom. Ezt érdemes feltüntetni valamilyen formában. Ez egyrészt elősegíti egy Prolog program működésének jobb megértését, másrészt bizonyos Prolog kiterjesztések használnak típusokat, ezért érdemes megismerkedni velük. A jegyzet számos pontján találkozhatunk Prolog megjegyzésként megadott típusleírással. Ez a fejezet ilyen típusleírásokkal foglalkozik.

Típusleírásnak tömör Prolog kifejezések egy halmazának megadását értjük. Emlékeztetünk arra, hogy egy Prolog kifejezést akkor hívunk tömörnek, ha nem tartalmaz változót. A típusleírás, nevéhez hűen, egy típust definiál.

Alaptípusok leírására használhatjuk az `integer, float, number, atom, atomic` és `any` konstansokat. Az első öt típusnév a megfelelő konstanshalmazt jelöli, az `any` típusnév pedig az összes Prolog kifejezés halmazát.

Az alaptípusokból kiindulva definiálhatunk összetett típusokat. Ehhez meg kell adnunk egy struktúranevet, valamint minden argumentumáról meg kell mondanunk, hogy milyen típusú. A struktúranevet és az argumentumok típusait megadó kifejezést kapcsos zárójelbe téve kapunk egy összetett típust leíró kifejezést. Például a `{személy(atom,atom,integer)}` kifejezés egy típust definiál. Nevezetesen minden olyan Prolog kifejezés, melynek funktora `személy/3`, első két argumentuma atom és a harmadik egész szám, ilyen típusú.

Ezt precízebben és általánosabban úgy írhatjuk le, hogy a

$$\{\ \texttt{valami}(\texttt{T}_1,\ \ldots,\ \texttt{T}_n)\ \}$$

halmazkifejezés ekvivalens a

$$\{\ \texttt{valami}(e_1,\ \ldots,\ e_n)\ |\ e_1 \in \texttt{T}_1,\ \ldots,\ e_n \in \texttt{T}_n\ \}, n \geq 0$$

kifejezéssel, azaz a halmaz minden olyan `valami` nevű struktúrát tartalmaz, amelynek argumentumai rendre T_1, T_2 stb. típusúak.

Egy típust képezhetünk halmazok uniójaként a `\/` operátor felhasználásával. Például helyes típusdefiníció az alábbi:

```
{személy(atom,atom,integer)} \/ {atom-atom} \/ atom
```

Azaz például az `alma-alma` Prolog kifejezés ilyen típusú, de a `személy('Nagy','Béla',24)` is.

Azért, hogy hivatkozni tudjunk a típusra el kell neveznünk azt. Ezt az alábbi módon tehetjük meg (Prolog megjegyzésként):

```
% :- type ⟨típusnév⟩ == ⟨típusleírás⟩ .
```

Lássunk is rögtön két példát! Vegyük észre, hogy a második típust rekurzív módon írtuk fel, a típusleírás hivatkozik ugyanis a típusnévre:

```
% :- type t1 == {atom-atom} \/ atom.
% :- type ember == {ember-atom} \/ atom.
```

Az eddig látott példákban a típusleírásban mindig csak az `atom` típusnév szerepelt a `{}` zárójelpáron kívül. Ez nem szükségszerű, tetszőleges típusnév szerepelhet így, olyan is, amelyet mi definiáltunk. Ennek megfelelően az alábbi két (végtelenül egyszerű) példa mindegyike helyes.

```
% :- type új_típus1 == ember.
% :- type új_típus2 == {ember}.
```

A két típus nem egyenlő. Az `új_típus1` típus pontosan ugyanazt a halmazt jelöli, mint az `ember` típus, `új_típus2` azonban az egyetlen `ember` névkonstanst tartalmazó halmazt. (Esetünkben `új_típus2` valódi részhalmaza `új_típus1`-nek.)

A megkülönböztetett únió fogalmáról már esett szó. Egy megkülönböztetett únió csupa különböző funktorú *összetett* típus úniója. Fontos, hogy nem a strukúranévnek, hanem a funktornak kell különböznie. Azaz nyugodtan lehet két, azonos struktúranevű, de különböző argumentumszámú típus. Megkülönbözetett úniót jelölhetünk a szokásos

:- type T == { S_1 } \/ ...\/ { S_n }

helyett így is:

:- type T ---> S_1 ; ...; S_n.

Fontos, hogy a megkülönböztetett únió is típus, csak éppen speciális. Két példa megkülönböztetett únióra:

```
% :- type ember ---> ember-atom ; semmi.
% :- type egészlista ---> [] ; [integer|egészlista]
```

3.9.1. Paraméteres típusok

Az előzőekben láttuk, hogy hogyan definiálhatunk saját típust. Legutolsó példaként megadtunk egy olyan listát, amely egészeket tartalmaz. Jó lenne, ha megadhatnánk egy lista-mintát is, azaz egy olyan típust, amely tetszőleges (de egyforma) típusú elemek listája lehet. Ugyanígy, bár tudunk definiálni olyan típust, amelyet az `atom-atom` alakú struktúrák határoznak meg, szükségünk lehet egy olyan típusra, amelyet tetszőleges típusú elemek párjai alkotnak. Erre szolgálnak az ún. *paraméteres típusok* és erre láthatunk példát az alábbiakban.

```
% :- type list(T) ---> [] ; [T|list(T)].  (1).
% :- type pair(T1, T2) ---> T1 - T2.      (2)
% :- type assoc_list(KeyT, ValueT)        (3)
%           == list(pair(KeyT, ValueT)).
% :- type szótár == assoc_list(szó, szó). (4)
% :- type szó == atom.                    (5)
```

(1) `T` típusú elemekből álló listákat foglal magába, (2) minden olyan '-' nevű kétargumentumú struktúrát, amelynek első argumentuma `T1`, második `T2` típusú.

(3) egy olyan típust definiál, amelybe `KeyT` és `ValueT` típusú párokból álló listák tartoznak. Végül (4) egy olyan, `szótár` nevű, típust határoz meg, amelybe ((5) alapján) atomokból képzett párokból álló listák tartoznak. Ha belegondolunk, ez tényleg felfogható úgy, mint egy szótár.

A szakasz zárásaként megadjuk a típusdeklarációk formális szintaxisát:

⟨típusdeklaráció⟩	::=	⟨típuselnevezés⟩ | ⟨típuskonstrukció⟩
⟨típuselnevezés⟩	::=	`:- type` ⟨típusazonosító⟩ `==` ⟨típusleírás⟩ `.`
⟨típuskonstrukció⟩	::=	`:- type` ⟨típusazonosító⟩ `--->` ⟨megkülönb. únió⟩ `.`
⟨megkülönb. únió⟩	::=	⟨konstruktor⟩ `;` ...
⟨konstruktor⟩	::=	⟨névkonstans⟩ | ⟨struktúranév⟩`(`⟨típusleírás⟩, ...`)`
⟨típusleírás⟩	::=	⟨típusazonosító⟩ | ⟨típusváltozó⟩ | `{` ⟨konstruktor⟩ `}` | ⟨típusleírás⟩ `\/` ⟨típusleírás⟩
⟨típusazonosító⟩	::=	⟨típusnév⟩ | ⟨típusnév⟩`(`⟨típusváltozó⟩, ...`)`
⟨típusnév⟩	::=	⟨névkonstans⟩
⟨típusváltozó⟩	::=	⟨változó⟩

3.9.2. Predikátumtípus-deklarációk

Egy *predikátumtípus-deklaráció* leírja, hogy egy predikátum milyen típusú adatokat képes fogadni, illetve visszaadni az egyes argumentumaiban. Egy ilyen deklaráció általánosan a következőképpen néz ki:

`:- pred` ⟨eljárásnév⟩`(`⟨típusleírás⟩, ...)

Lássunk néhány példát az elmondottakra! Az első esetben a `member/2` eljárásról jelentjük ki, hogy első argumentuma `T` típusú, míg a második `T` típusú elemeket tartalmazó lista. Másodikként az `append/3` eljárást írjuk le hasonló módon.

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

Nyilvánvaló, hogy egy predikátumtípus-deklarációban használhatóak az előzetesen megadott típusleírások.

Eljárásokkal kapcsolatban van még egy fontos fogalom, amit **predikátummód-deklarációnak** hívunk. Egy ilyen deklaráció leírja, hogy az egyes argumentumok kimenő vagy bemenő módban használatosak. Egy eljáráshoz több ilyen móddeklaráció is megadható annak megfelelően, hogy az eljárás milyen különböző módokban képes működni:

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

A predikátummód-deklarációk általános felépítése a következő:

`:- mode` ⟨eljárásnév⟩`(`⟨módazonosító⟩, ...)

ahol

⟨módazonosító⟩ ::= `in` | `out`.

Arra is van lehetőség, hogy egyetlen deklarációba fogjuk össze a típus- és móddeklarációt is, például:

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

Ilyen esetben az általános alak:

`:- pred` ⟨eljárásnév⟩`(`⟨típusazonosító⟩`::`⟨módazonosító⟩, ...)

A SICStus kézikönyv a fentiektől eltérő jelölést használ a bemenő/kimenő argumentumok jelzésére. Az `append/3` esetén például:

```
append(+L1, ?L2, -L3). (1)
append(?L1, ?L2, +L3). (2)
```

(1) jelöli az ellenőrzésre és két lista összefűzésére is alkalmas megvalósítást, míg (2) a szétszedésre is használható. Ennek megfelelően a `+`, `-` és `?` jelentése:

- `+` bemenő argumentum (behelyettesített)
- `-` kimenő argumentum (behelyettesítetlen)
- `?` tetszőleges argumentum