

A Prolog alapvető adatkezelő művelete: az egyesítés

AZ EGYESÍTÉSI ALGORITMUS

LP-63

Egyesítés: változók behelyettesítése

- A behelyettesítés fogalma
 - A behelyettesítés egy olyan függvény, amely bizonyos változókhoz kifejezéseket rendel.
 - Példa: $\sigma = \{X \leftarrow a, Y \leftarrow s(b, B), Z \leftarrow C\}$. Itt $Dom(\sigma) = \{X, Y, Z\}$
 - A σ behelyettesítés x -hez a -t, y -hoz $s(b, B)$ -t z -hez C -t rendel. Jelölés: $X\sigma = a$ stb.
 - A behelyettesítés-függvény természetes módon kiterjeszthető az összes kifejezésre:
 - $K\sigma$: σ alkalmazása K kifejezésre: σ behelyettesítéseit *egvidejűleg* elvégezzük K -ban.
 - Példa: $f(g(Z, h), A, Y)\sigma = f(g(C, h), A, s(b, B))$
 - A σ és θ behelyettesítések kompozíciója ($\sigma \otimes \theta$) — egymás utáni alkalmazásuk
 - A $\sigma \otimes \theta$ behelyettesítés az $x \in Dom(\sigma)$ változókhoz az $(x\sigma)\theta$ kifejezést, a többi $y \in Dom(\theta) \setminus Dom(\sigma)$ változóhoz $y\theta$ -t rendel ($Dom(\sigma \otimes \theta) = Dom(\sigma) \cup Dom(\theta)$):

$$\sigma \otimes \theta = \{x \leftarrow (x\sigma)\theta \mid x \in Dom(\sigma)\} \cup \{y \leftarrow y\theta \mid y \in Dom(\theta) \setminus Dom(\sigma)\}$$
 - Pl. $\theta = \{X \leftarrow b, B \leftarrow d\}$ esetén $\sigma \otimes \theta = \{X \leftarrow a, Y \leftarrow s(b, d), Z \leftarrow C, B \leftarrow d\}$
- Egy G kifejezés **általánosabb** mint egy S , ha létezik olyan ρ behelyettesítés, hogy $S = G\rho$
 - Példa: $G = f(A, Y)$ általánosabb mint $S = f(1, s(Z))$, mert $\rho = \{A \leftarrow 1, Y \leftarrow s(Z)\}$ esetén $S = G\rho$.

LP-64

Egyesítés: legáltalánosabb egyesítő

- A és B kifejezések egyesíthetők ha létezik egy olyan σ behelyettesítés, hogy $A\sigma = B\sigma$. Ezt az $A\sigma = B\sigma$ kifejezést A és B egyesített alakjának nevezzük.
- Két kifejezésnek általában több egyesített alakja lehet.
 - Példa: $A = f(X, Y)$ és $B = f(s(U), U)$ egyesített alakja pl.
 - $K_1 = f(s(a), a)$ a $\sigma_1 = \{X \leftarrow s(a), Y \leftarrow a, U \leftarrow a\}$ behelyettesítéssel
 - $K_2 = f(s(U), U)$ a $\sigma_2 = \{X \leftarrow s(U), Y \leftarrow U\}$ behelyettesítéssel
 - $K_3 = f(s(Y), Y)$ a $\sigma_3 = \{X \leftarrow s(Y), U \leftarrow Y\}$ behelyettesítéssel
- A és B legáltalánosabb egyesített alakja egy olyan C kifejezés, amely A és B minden egyesített alakjánál általánosabb
 - A fenti példában K_2 és K_3 legáltalánosabb egyesített alakok
- **Tétel:** A legáltalánosabb egyesített alak, változó-átnevezéstől eltekintve egyértelmű.
- A és B legáltalánosabb egyesítője egy olyan $\sigma = mgu(A, B)$ behelyettesítés, amelyre $A\sigma$ és $B\sigma$ a két kifejezés legáltalánosabb egyesített alakja.
 - A fenti példában σ_2 és σ_3 legáltalánosabb egyesítő.
- **Tétel:** A legáltalánosabb egyesítő, változó-átnevezéstől eltekintve egyértelmű.

Az egyesítési algoritmus

- Az egyesítési algoritmus
 - bemenete: két Prolog kifejezés: A és B
 - feladata: a két kifejezés egyesíthetőségének eldöntése
 - eredménye: sikeresség esetén a legáltalánosabb egyesítő ($mgu(A, B)$) előállítás.
- Az egyesítési algoritmus, $\sigma = mgu(A, B)$ előállítása
 1. Ha A és B azonos változók vagy konstansok, akkor $\sigma = \{\}$ (üres behelyettesítés).
 2. Egyébként, ha A változó, akkor $\sigma = \{A \leftarrow B\}$.
 3. Egyébként, ha B változó, akkor $\sigma = \{B \leftarrow A\}$.
 4. Egyébként, ha A és B azonos nevű és argumentumszámú összetett kifejezések és argumentum-listáik A_1, \dots, A_N ill. B_1, \dots, 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 $\sigma = \sigma_1 \otimes \sigma_2 \otimes \sigma_3 \otimes \dots$
 5. Minden más esetben a A és B nem egyesíthető.

Egyesítési példák

- $A = \text{sum_tree}(\text{leaf}(V), V), B = \text{sum_tree}(\text{leaf}(5), S)$
 - (4.) A és B neve és argumentumszáma megegyezik
 - (a.) $mgu(\text{leaf}(V), \text{leaf}(5))$ (4., majd 2. szerint) $= \{V \leftarrow 5\} = \sigma_1$
 - (b.) $mgu(V\sigma_1, S) = mgu(5, S)$ (3. szerint) $= \{S \leftarrow 5\} = \sigma_2$
 - tehát $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{V \leftarrow 5, S \leftarrow 5\}$
- $A = \text{node}(\text{leaf}(X), T), B = \text{node}(T, \text{leaf}(3))$
 - (4.) A és B neve és argumentumszáma megegyezik
 - (a.) $mgu(\text{leaf}(X), T)$ (3. szerint) $= \{T \leftarrow \text{leaf}(X)\} = \sigma_1$
 - (b.) $mgu(T\sigma_1, \text{leaf}(3)) = mgu(\text{leaf}(X), \text{leaf}(3))$ (4., majd 2. szerint) $= \{X \leftarrow 3\} = \sigma_2$
 - tehát $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{T \leftarrow \text{leaf}(3), X \leftarrow 3\}$

Egyesítési példák a gyakorlatban

- Az egyesítéssel kapcsolatos beépített eljárások:
 - $X = Y$ egyesíti a két argumentumát, meghiúsul, ha ez nem lehetséges.
 - $X \backslash= Y$ sikerül, ha két argumentuma nem egyesíthető, egyébként meghiúsul.
- Példák:


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

Az egyesítés kiegészítése: előfordulás-ellenőrzés (*occurs check*)

- Kérdés: x és $s(x)$ egyesíthető-e?
 - A matematikai válasz: *nem*, egy változó nem egyesíthető egy olyan struktúrával, amelyben előfordul (ez az előfordulás-ellenőrzés).
 - Az ellenőrzés költséges, ezért alaphelyzetben nem alkalmazzák, így ciklikus kifejezések keletkezhetnek.
 - Szabványos eljárásként rendelkezésre áll: `unify_with_occurs_check/2`
 - Kiterjesztés (pl. SICStus): az előfordulás-ellenőrzés elhagyása miatt keletkező ciklikus kifejezések tisztességes kezelése.
- Példák:


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

A PROLOG VÉGREHAJTÁSI MECHANIZMUSA

LP-71

A eljárás-redukciós végrehajtási modell

- A redukciós végrehajtási modell alap gondolata
 - A végrehajtás egy állapota: egy célsorozat
 - A végrehajtás kétféle lépésből áll:
 - redukciós lépés: egy célsorozat + klóz \rightarrow új célsorozat
 - zsákutca esetén visszalépés: visszatérés a legutolsó választási ponthoz
 - Választási pont:
 - létrehozása: olyan redukciós lépés amely nem a legutolsó klózzal illesztett
 - aktiválása: visszalépéskor visszatérünk a választási pont célsorozatához és a **további** klózek között keresünk illeszthetőt (Emiatt a választási pontban a célsorozat mellett az illesztett klóz sorszámát is tárolni kell.)
 - az ún. indexelés segít a választási pontok számának csökkentésében
- A redukciós modell keresési fával szemléltethető
 - A végrehajtás során a fa csomópontjait járjuk be mélységi kereséssel
 - A fa gyökerétől egy adott pontig terjedő szakaszon kell a választási pontokat megjegyezni — ez a választási verem (choice point stack)

A Prolog végrehajtás eljárásos modelljei

- Az azonos funktorú klózek alkotnak egy eljárást
- Egy eljárás meghívása a hívás és klózfej mintaillesztésével (egyesítésével) történik
- A végrehajtás lépéseinek modellezése:
 - Eljárás-redukciós modell
 - Az alaplépés: egy hívás-sorozat (azaz célsorozat) redukálása egy klóz segítségével (ez a már ismert redukciós lépés).
 - Visszalépés: visszatérünk egy korábbi célsorozathoz, és újabb klózzal próbálkozunk.
 - A modell előnyei: pontosan definiálható, a keresési tér szemléltethető
 - Eljárás-doboz modell
 - Az alap gondolat: egymásba skatulyázott eljárás-dobozok kapuin lépünk be és ki.
 - Egy eljárás-doboz kapui: hívás (belépés), sikeres kilépés, sikertelen kilépés.
 - Visszalépés: új megoldást kérünk egy már lefutott eljárástól (újra kapu).
 - A modell előnyei: közel van a hagyományos rekurzív eljárásmodellhez, a Prolog beépített nyomkövetője is ezen alapul.

LP-72

A redukciós modell alapeleme: redukciós lépés

- Redukciós lépés: egy célsorozat redukálása egy újabb célsorozattá
 - egy programklóz segítségével (az első cél felhasználói eljárást hív):
 - 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 **célsorozat** maradékán is
 - Az új célsorozat: a klóztörzs és utána a maradék célsorozat
 - Ha a hívás és a klózfej nem egyesíthető, akkor a redukciós lépés meghiúsul.
 - egy beépített eljárás segítségével (az első cél beépített eljárást hív):
 - A célsorozatot szétbontjuk az első hívásra és a maradékra.
 - A beépített eljárás hívást végrehajtjuk.
 - Ez lehet sikeres (változó-behelyettesítésekkel), vagy lehet sikertelen.
 - Siker esetén a behelyettesítéseket elvégezzük a célsorozat maradékán.
 - Az új célsorozat: az (első hívás elhagyása után fennmaradó) maradék célsorozat.
 - Ha a beépített eljárás hívása sikertelen, akkor a redukciós lépés meghiúsul.

A Prolog végrehajtási algoritmus

1. (Kezdeti beállítások:) A verem üres, $CS := c\acute{e}lsorozat$
2. (Beépített eljárások:) Ha CS első hívása beépített akkor hajtjuk végre,
 - a. Ha sikertelen \Rightarrow 6. lépés.
 - b. Ha sikeres, $CS :=$ a redukciós lépés eredménye \Rightarrow 5. lépés.
3. (Klőzszámoló kezdőértékezése:) $I = 1$.
4. (Redukciós lépés:) Tekintsük CS első hívására vonatkoztható klőzok listáját. Ez indexelés nélkül a predikátum összes klőza lesz, indexelés esetén ennek egy megszárt részsorozata. Tegyük fel, hogy ez a lista N elemű.
 - a. Ha $I > N \Rightarrow$ 6. lépés.
 - b. Redukciós lépés a lista I -edik klőza és a CS célsorozat között.
 - c. Ha sikertelen, akkor $I := I+1 \Rightarrow$ 4. lépés.
 - d. Ha $I < N$ (nem utolsó), akkor veremljük $\langle CS, I \rangle$ -t.
 - e. $CS :=$ a redukciós lépés eredménye
5. (Siker:) Ha CS üres, akkor sikeres vég, egyébként \Rightarrow 2. lépés.
6. (Sikertelenség:) Ha a verem üres, akkor sikertelen vég.
7. (Visszalépés:) Ha a verem nem üres, akkor leemeljük a veremből $\langle CS, I \rangle$ -t, $I := I+1$, és \Rightarrow 4. lépés.

Indexelés (előzetes)

- Mi az indexelés?
 - egy hívásra vonatkoztható (potenciálisan illeszthető) klőzok gyors kiválasztása,
 - egy eljárás klőzainak **fordítási idejű** csoportosításával.
- A legtöbb Prolog rendszer, így a SICStus Prolog is, az első fej-argumentum alapján indexel (first argument indexing).
- Az indexelés alapja az első fejargumentum külső funkora:
 - C szám vagy névkonstans esetén $C/0$;
 - R nevű és N argumentumú struktúra esetén R/N ;
 - változó esetén nem értelmezett (minden funktohoz besoroltatik).
- Az indexelés megvalósítása:
 - Fordítási időben minden funktohoz elkészítjük az alkalmazható klőzok listáját
 - Futáskor lényegében konstans idő alatt elő tudjuk venni a megfelelő klőzlistát
 - *Fontos:* ha egyelemű a részhalmaz, nem hozunk létre választási pontot!
- Például $szuloje('István', X)$ kételemű klőzlistára szűkít, de $szuloje(X, 'István')$ mind a 6 klőzt megtartja (mert a SICStus Prolog csak az első argumentum szerint indexel)

Redukciós modell — előnyök és hátrányok

- Előnyök
 - (viszonylag) egyszerű és (viszonylag) precíz definíció
 - a keresési tér megjeleníthető, grafikus szemléltethető
- Hátrányok
 - az eljárásokból való kilépést elfedi, pl.

$p :- q, r.$	$G0: p ?$
$q :- s, t.$	$G1: q, r ?$
$s.$	$G2: s, t, r ?$
$t.$	$G3: t, r ?$
$r.$	$G4: r ? \quad \Leftarrow q\text{-ből való kilépés}$
	$G5: [] ?$
 - nem jól illeszkedik a Prolog megvalósítások tényleges végrehajtási mechanizmusához
 - nem alkalmazható „igazi” Prolog programok nyomkövetésére (hosszú célsorozatok)
- Ezért van létjogosultsága egy másik modellnek:
 - eljárás-doboz (procedure box) modell
 - (szokás még 4-kapus doboz ill. Byrd doboz modellnek is nevezni)
 - a Prolog rendszerek nyomkövető szolgáltatása erre a modellre épül

Az eljárás-doboz modell

- A Prolog eljárás-végrehajtás két fázisa
 - előre menő végrehajtás: egymásba skatulyázott eljárás-belépések és -kilépések
 - visszafelé menő végrehajtás: újabb megoldás kérése egy már lefutott eljárástól
- Egy egyszerű példa

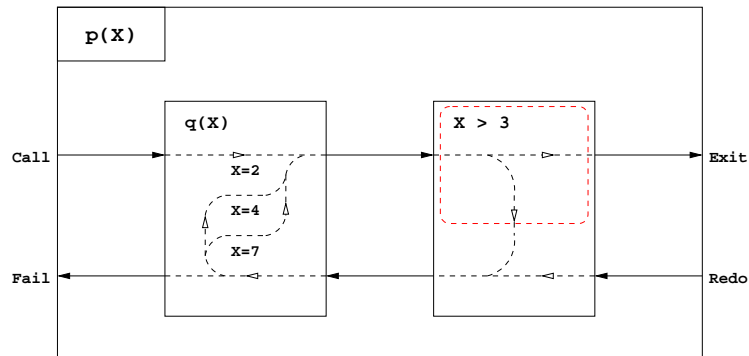
$q(2).$	$q(4).$	$q(7).$	$p(X) :- q(X), X > 3.$
---------	---------	---------	------------------------

 - Belépünk a $p/1$ eljárásba (Hívási kapu, Call port)
 - Belépünk a $q/1$ eljárásba (Call)
 - A $q/1$ eljárás sikeresen lefut a $q(2)$ eredménnyel (Kilépési kapu, Exit port)
 - A $> /2$ eljárásba belépünk a $2 > 3$ hívással (Call)
 - A $> /2$ eljárás sikertelenül fut le (Meghiúsulási kapu, Fail port)
 - (visszafelé menő futás): visszatérünk (a már lefutott) $q/1$ -be, újabb megoldást kérve (Újra kapu, Redo Port)
 - A $q/1$ eljárás sikeresen lefut a $q(4)$ eredménnyel (Exit)
 - A $4 > 3$ eljárás hívással a $> /2$ -be belépünk majd sikeresen kilépünk (Call, Exit)
 - A $p/1$ eljárás sikeresen lefut $p(4)$ eredménnyel (Exit)

Eljárás-doboz modell — grafikus szemléltetés

$q(2).$ $q(4).$ $q(7).$

$p(X) :- q(X), X > 3.$



Eljárás-doboz modell — egyszerű nyomkövetési példa

● Az előző példa nyomkövetése SICStus Prologban

$q(2).$ $q(4).$ $q(7).$

$p(X) :- q(X), X > 3.$

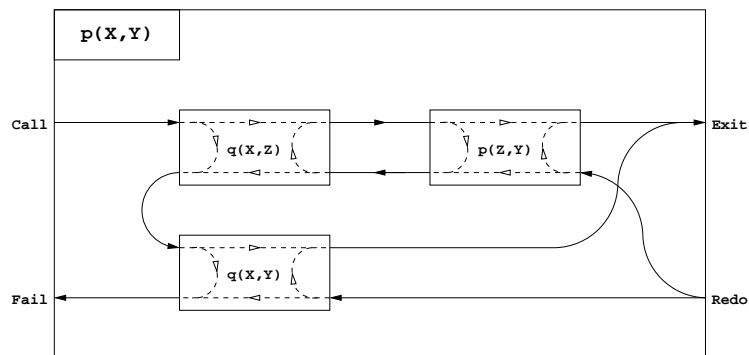
```
| ?- trace, p(X).
      1      1 Call: p(_463) ?
      2      2 Call: q(_463) ?
?      2      2 Exit: q(2) ?           % ? = nondeterminisztikus
kilépés
      3      2 Call: 2>3 ?
      3      2 Fail: 2>3 ?
      2      2 Redo: q(2) ?           % visszafelé menő végrehajtás
?      2      2 Exit: q(4) ?
      4      2 Call: 4>3 ?
      4      2 Exit: 4>3 ?
?      1      1 Exit: p(4) ?
X = 4 ? ;
      1      1 Redo: p(4) ?           % visszafelé menő végrehajtás
      2      2 Redo: q(4) ?           % visszafelé menő végrehajtás
      2      2 Exit: q(7) ?
      5      2 Call: 7>3 ?
      5      2 Exit: 7>3 ?
      1      1 Exit: p(7) ?
X = 7 ? ;
no
```

Eljárás-doboz: egy összetettebb példa

$p(X,Y) :- q(X,Z), p(Z,Y).$

$p(X,Y) :- q(X,Y).$

$q(1,2).$ $q(2,3).$ $q(2,4).$



Eljárás-doboz modell — „kapcsolási” alapelvek

- Hogyan építhető fel egy „szülő” eljárás doboza a benne hívott eljárások dobozaiból?
- Feltehető, hogy a klózfejekben (különböző) változók vannak, a fej-egyesítéseket hívás(okk)á alakítva
- Előre menő végrehajtás:
 - A szülő Hívás kapuját az első klóz első hívásának Hívás kapujára kötjük.
 - Egy rész-eljárás Kilépési kapuját
 - a következő hívás Hívás kapujára, vagy,
 - ha nincs következő hívás, akkor a szülő Kilépési kapujára kötjük
- Visszafelé menő végrehajtás:
 - Egy rész-eljárás Meghiúsulási kapuját
 - az előző hívás Újra kapujára, vagy,
 - ha nincs előző hívás, akkor a következő klóz első hívásának Hívás kapujára, vagy
 - ha nincs következő klóz, akkor a szülő Meghiúsulási kapujára kötjük
 - A szülő Újra kapuját mindegyik klóz utolsó hívásának Újra kapujára kötjük
 - mindig arra a klózra térünk vissza, amelyben legutoljára volt a vezérlés

Eljárás-doboz modell — OO szemléletben

- Minden eljáráshoz tartozik egy osztály, amelynek van egy konstruktor függvénye (amely megkapja a hívási paramétereket) és egy „adj egy (következő) megoldást” metódusa.
- Az osztály nyilvántartja, hogy hányadik klózban jár a vezérlés
- A metódus első meghívásakor az első klóz első Hívás kapujára adja a vezérlést
- Amikor egy rész-eljárás Hívás kapuhoz érkezünk, **létrehozunk** egy példányt a meghívandó eljárásból, majd
- meghívjuk az eljáráspéldány „következő megoldás” metódusát (*)
 - Ha ez sikerül, akkor a vezérlés átkerül a következő hívás Hívás kapujára, vagy a szülő Kilépési kapujára
 - Ha ez meghiúsul, akkor **megszüntetjük** az eljáráspéldányt majd ugrunk az előző hívás Újra kapujára, vagy a következő klóz elejére, stb.
- Amikor egy Újra kapuhoz érkezünk, a (*) lépésnél folytatjuk.
- A szülő Újra kapuja (a „következő megoldás” nem első hívása) a tárolt klózsorszámnak megfelelő klózban az utolsó Újra kapura adja a vezérlést.

OO szemléletű dobozok: p/2 „következő megoldás” metódusának C++ kódja

```

boolean p::next()
{ switch(clno) {
  case 0: // entry point for the Call port
    clno = 1; // enter clause 1: p(X,Y) :- q(X,Z), p(Z,Y).
    qpztr = new q(x, &z); // create a new instance of subgoal q(X,Z)
    redoll:
    if(!qpztr->next()) { // if q(X,Z) fails
      delete qpztr; // destroy it,
      goto cl2; // and continue with clause 2 of p/2
    }
    ppztr = new p(z, py); // otherwise, create a new instance of subgoal p(Z,Y)
  case 1: // (enter here for Redo port if clno==1)
    /* redo12: */
    if(!ppztr->next()) { // if p(Z,Y) fails
      delete ppztr; // destroy it,
      goto redoll; // and continue at redo port of q(X,Z)
    }
    return TRUE; // otherwise, exit via the Exit port
  cl2:
    clno = 2; // enter clause 2: p(X,Y) :- q(X,Y).
    qbpztr = new q(x, py); // create a new instance of subgoal q(X,Y)
  case 2: // (enter here for Redo port if clno==1)
    /* redo21: */
    if(!qbpztr->next()) { // if q(X,Y) fails
      delete qbpztr; // destroy it,
      return FALSE; // and exit via the Fail port
    }
    return TRUE; // otherwise, exit via the Exit port
  } }

```

Visszalépéses keresés — egy aritmetikai példa

- Példa: „jó” számok keresése
- A feladat: keressük meg azokat a kétjegyű számokat amelyek négyzete háromjegyű és a szám fordítottjával kezdődik
- A program:

```

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

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

% Szam négyzete háromjegyű és a Szam fordítottjával kezdődik.
joszam(Szam):-
    decl(A), decl(B),
    Szam is A * 10 + B, Szam * Szam // 10 == B * 10 + A.

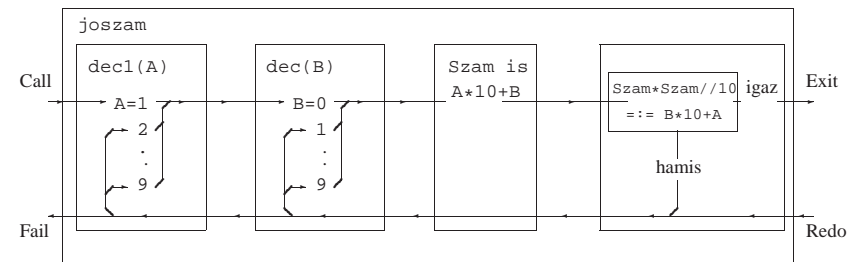
```

Prolog végrehajtás — a 4-kapus doboz modell

```

joszam(Szam):-
    decl(A), decl(B),
    Szam is A * 10 + B, Szam * Szam // 10 == B * 10 + A.

```



Visszalépéses keresés — számintervallum felsorolása

- `dec(J)` felsorolta a 0 és 9 közötti egész számokat
- Általánosítás: soroljuk fel az N és M közötti egészeket (N és M maguk is egészek)

```
% between(M, N, I): M =< I =< N, I egész.
between(M, N, M) :-
    M =< N.
between(M, N, I) :-
    M < N,
    M1 is M+1,
    between(M1, N, I).
```

```
% dec(X): X egy decimális számjegy
dec(X) :- between(0, 9, X).
```

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

A SICStus eljárás-doboz alapú nyomkövetése — legfontosabb parancsok

- Alapvető nyomkövetési parancsok
 - `h <RET>` (help) — parancsok listázása
 - `c <RET>` (creep) vagy `<RET>` — továbblépés minden kapunál megálló nyomkövetéssel
 - `l <RET>` (leap) — csak töréspontnál áll meg, de a dobozokat építi
 - `z <RET>` (zip) — csak töréspontnál áll meg, dobozokat nem épít
 - `+ <RET>` ill. `- <RET>` — töréspont rakása/eltávolítása a kurrens predikátumra
 - `s <RET>` (skip) — eljárástörzs átlépése (Call/Redo \Rightarrow Exit/Fail)
 - `o <RET>` (out) — kilépés az eljárástörzsből
- A Prolog végrehajtást megváltoztató parancsok
 - `u <RET>` (unify) — a kurrens hívást végrehajtás helyett egyesíti egy beolvasott kifejezéssel.
 - `r <RET>` (retry) — újakezdi a kurrens hívás végrehajtását (ugrás a Call kapura)
- Információ-megjelenítő és egyéb parancsok
 - `w <RET>` (write) — a hívás kiírása mélység-korlátozás nélkül
 - `b <RET>` (break) — új, beágyazott Prolog interakciós szint létrehozása
 - `n <RET>` (notrace) — nyomkövető kikapcsolása
 - `a <RET>` (abort) — a kurrens futás abbahagyása

TOVÁBBI VEZÉRLÉSI SZERKEZETEK

Diszjunkció, példa: az „őse” predikátum

- Az „őse” reláció a „szülője” reláció tranzitív lezártja: a szülő ős (1), és az ős őse is ős (2), azaz:

```
% ose0(E, Os): E ose Os.
ose0(E, Sz) :- szuloje(E, Sz). % (1)
ose0(E, Os) :- ose0(E, Os0), ose0(Os0, Os). % (2)
```

- Az `ose0` definíciója matematikailag helyes, de végtelen Prolog keresési teret ad:

```
szuloje(gyerek,apa). szuloje(gyerek,anya). szuloje(anya,nagyapa).

| ?- ose0(gyerek, Os).
    Os = apa ? ; Os = anya ? ; {néhány másodperc után:}
    ! Resource error: insufficient memory
```

- A végtelen rekurzió oka: Az `:- ose0(apa, X)`. cél esetén az (1) klóz megghiúsul, (2) pedig egy `:- ose0(apa, Y), ose0(Y, X)`. célsorozathoz vezet stb.

- A balrekurziót kiküszöbölve kapjuk:

```
ose1(E, Sz) :- szuloje(E, Sz). % (3)
ose1(E, Os) :- szuloje(E, Sz), ose1(Sz, Os). % (4)

| ?- ose1(gyerek, Os).
    Os = apa ? ; Os = anya ? ; Os = nagyapa ? ; no
```

- Ez minden `szuloje(X, Y)` részcélt kétszer hajt végre: (3)-ban és (4)-ben.

A diszjunkció

- Az `ose1` predikátum hatékonyabbá tehető klózai összevonásával:

```
ose2(E, Os) :- szuloje(E, Sz), maga_vagy_ose(Sz, Os).
```

```
maga_vagy_ose(E, E). (1)
```

```
maga_vagy_ose(E, Os) :- ose2(E, Os).
```

- A `maga_vagy_ose` predikátum egy ún. **diszjunkció** bevezetésével kiküszöbölhető:

```
ose3(E, Os) :-
  szuloje(E, Sz),
  ( Os = Sz
  ; ose3(Sz, Os)
  ).
```

- A SICStus Prolog ténylegesen úgy implementálja a fenti diszjunkciót, hogy bevezet egy `maga_vagy_ose`-vel azonos segéd-predikátumot és az `ose3` klózt `ose2`-vé alakítja.
- (Ismétlés:) Az $X=Y$ beépített predikátum a két argumentumát egyesíti.
- Az $=/2$ eljárás egy tényállítással definiálható: $U = V \equiv (U, U) \vee (V, V)$, vö. (1).

A diszjunkció mint szintaktikus édesítőszert

- A diszjunkció akárhány tagú lehet. A ‘;’ művelet gyengébben köt mint a ‘,’, ezért a diszjunkciót mindig zárójelbe tesszük, míg az ágait nem kell zárójelezni. Példa, „szabványos” formázással:

```
a(X, Y, Z) :-
  p(X, U), q(Y, V),
  ( r(U, T), s(T, Z)
  ; t(V, Z)
  ; t(U, Z)
  ),
  u(X, Z).
```

- A diszjunkció egy segéd-predikátummal mindig kiküszöbölhető

- Megkeressük azokat a változókat, amelyek a diszjunkcióban és azon kívül is előfordulnak
- A segéd-predikátumnak ezek a változók lesznek az argumentumai
- A segéd-predikátum minden klóza megfelel a diszjunkció egy ágának

```
seged(U, V, Z) :- r(U, T), s(T, Z).
seged(U, V, Z) :- t(V, Z).
seged(U, V, Z) :- t(U, Z).
```

```
a(X, Y, Z) :-
  p(X, U), q(Y, V),
  seged(U, V, Z),
  u(X, Z).
```

- A diszjunkció szemantikáját ezzel a segéd-predikátumos átalakítással definiáljuk.

Diszjunkció — megjegyzések

- Az egyes klózok ‘ÉS’ vagy ‘VAGY’ kapcsolatban vannak?

- A program klózai **ÉS** kapcsolatban vannak, pl.

```
szuloje('Imre', 'István'). szuloje('Imre', 'Gizella').
```

jelentése: Imre szülője István **ÉS** Imre szülője Gizella.

- Az **ÉS** kapcsolatban levő klózok alternatív (VAGY kapcsolatban levő) válaszokhoz vezetnek:

```
:- szuloje('Imre', Sz). => Sz = 'István' ? ; Sz = 'Gizella' ? ; no
```

A „Ki Imre szülője?” kérdésre a válasz: István vagy Gizella.

- A fenti két klózos predikátum átalakítható egyetlen klózzá, diszjunkció segítségével:

```
szuloje('Imre', Sz) :-
  ( Sz = 'István' (*),
  ; Sz = 'Gizella' (*),
  ).
```

A konjunkció ezáltal diszjunkcióvá alakult (vö. De Morgan azonosságok).

- Általánosan: tetszőleges predikátum egyklózosá alakítható:

- a klózokat átalakítjuk azonos fejűvé, új változók és egyenlőségek bevezetésével:

```
szuloje('Imre', Sz) :- Sz = 'István'.
szuloje('Imre', Sz) :- Sz = 'Gizella'.
```

- a klóztörzseket egy diszjunkcióvá fogjuk össze, amely az új predikátum törzse (lásd (*)).

Negáció

- Feladat: Keressünk (adatbázisunkban) egy olyan szülőt, aki **nem** nagyszülő!

- Ehhez negációra van szükségünk:

- Meghíúsulós negáció: a `\+` hívás szerkezet lefuttatja hívást, és pontosan akkor sikerül, ha a hívás meghíúsult.

- Egy megoldás:

```
| ?- szuloje(_, X), \+ nagyszuloje(_, X).
X = 'István' ? ;
X = 'Gizella' ? ;
no
```

- Mi történik ha a két hívást megcseréljük?

```
| ?- \+ nagyszuloje(_, X), szuloje(_, X).
no
```


A meghíúsulás negáció (NF — Negation by Failure)

- $A \setminus +$ hívás beépített meta-eljárás (vö. $\not\vdash$ — nem bizonyítható)

- végrehajtja a hívás hívást,
- ha hívás sikeresen lefutott, akkor meghíúsul,
- egyébként (azaz ha hívás meghíúsult) sikerül.

- $\setminus +$ hívás futása során hívás legfeljebb egy megoldása áll elő

- $\setminus +$ hívás sohasem helyettesít be változót

- Gondok a meghíúsulás negációval:

- „zárt világ feltételezése” (CWA) — ami nem bizonyítható, az nem igaz.

```
| ?- \+ szuloje('Imre', X).          ----> no
| ?- \+ szuloje('Géza', X).          ----> true ?
```

- $\setminus + H$ deklaratív szemantikája: $\neg\exists X(H)$, ahol X a H -ban a hívás pillanatában

behelyettesítetlen változókat jelöli.

```
| ?- \+ X = 1, X = 2.                ----> no
| ?- X = 2, \+ X = 1.                ----> X = 2 ?
```

Példa: együttható meghatározása lineáris kifejezésben

- Formula: számokból és az 'x' névkonstansból '+' és '*' operátorokkal épül fel.

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

- Lineáris formula: a '*' operátor legalább egyik oldalán szám áll.

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

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

Együttható meghatározása: többszörös megoldások kiküszöbölése

- negáció alkalmazásával:

```
(...)
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.
```

- hatékonyabban, feltételes kifejezéssel:

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

Feltételes kifejezések

- Szintaxis (felt, akkor, egyébként tetszőleges célsorozatok):

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

- Deklaratív szemantika: a fenti alak jelentése megegyezik az alábbival, ha a felt egy egyszerű feltétel (nem oldható meg többféleképpen):

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

Feltételes kifejezések (folyt.)

● Procedurális szemantika

A (felt->akkor;egyébként), folytatás célsorozat végrehajtása:

- 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.

● Többszörös elágaztatás skatulyázott feltételes kifejezésekkel:

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

● Az egyébként rész elhagyható, alapértelmezése: fail.

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

A PROLOG SZINTAXIS

Feltételes kifejezés — példák

● Faktoriális

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

● Jelentése azonos a sima diszjunkciós alakkal (lásd komment), de annál hatékonyabb, mert nem hagy maga után választási pontot.

● Szám előjele

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

A Prolog szintaxis összefoglalása

● A Prolog szintaxis alapelvei

- Minden programelem kifejezés!
- A szükséges összekötő jelek (', ', ';', ':- -->): szabványos operátorok.
- A beolvasott kifejezést funktora alapján osztályozzuk:
 - *kérdés*: *?- Cél*.
Célt lefuttatja, és a változó-behelyettesítéseket kiírja (ez az alapértelmezés az ún. top-level interaktív felületen).
 - *parancs*: *:- Cél*.
A Célt csendben lefuttatja. Pl. deklaráció (operátor, ...) elhelyezésére.
 - *szabály*: *Fej :- Törzs*.
A szabályt felveszi a programba.
 - *nyelvtani szabály*: *Fej --> Törzs*.
Prolog szabállyá alakítja és felveszi (lásd a DCG nyelvtan).
 - *tényállítás*: *Minden egyéb kifejezés*.
Üres törzsű szabályként felveszi a programba.

A Prolog nyelv-változatok

- A SICStus rendszer két üzemmódja
 - iso Az ISO Prolog szabványnak megfelelő.
 - sicstus Korábbi változatokkal kompatibilis.
 - Állítása: `set_prolog_flag(language, Mód)`.
 - Különbségek:
 - szintaxis-részletek, pl. a `0x1ff` szám-alak csak ISO módban,
 - beépített eljárások viselkedésének kisebb eltérései.
 - az eddig ismertetett eljárások hatása lényegében nem változik.

Szintaktikus édesítőszer — összefoglalás, gyakorlati tanácsok

- Operátoros kifejezések alapstruktúra alakra hozása
 - Zárójelezzük be a kifejezést, az operátorok prioritása és fajtája alapján, például $-a+b*2 \Rightarrow ((-a)+(b*2))$.
 - Hozzuk az operátoros kifejezéseket alapstruktúra alakra:
 $(A \text{ Inf } B) \Rightarrow \text{Inf}(A,B)$, $(\text{Pref } A) \Rightarrow \text{Pref}(A)$, $(A \text{ Postf}) \Rightarrow \text{Postf}(A)$
 Példa: $((-a)+(b*2)) \Rightarrow (-a)+*(b,2) \Rightarrow +(-a),*(b,2)$.
 - Trükkös esetek:
 - A vesszőt névként idézni kell: pl. $(pp,(qq;rr)) \Rightarrow ', '(pp,;(qq,rr))$.
 - - Szám \Rightarrow negatív számkonstans, de - Egyéb \Rightarrow prefix alak.
 Példa: $-1+2 \Rightarrow +(-1,2)$, de $-a+b \Rightarrow +(-a),b$.
 - $Név(\dots) \Rightarrow$ struktúrakifejezés;
 $Név(\dots) \Rightarrow$ prefix operátoros kifejezés. Példák:
 $-(1,2) \Rightarrow -(1,2)$ (változatlan), de
 $-(1,2) \Rightarrow -(','(1,2))$.

Szintaktikus édesítőszer — listák, egyebek

- Listák alapstruktúra alakra hozása
 - Farok-megadás betoldása.
 $[1,2] \Rightarrow [1,2|[]]$. $[[X|Y]] \Rightarrow [[X|Y]|[]]$
 - Vessző (ismételt) kiküszöbölése $[Elem1,Elem2\dots] \Rightarrow [Elem1|[Elem2\dots]]$.
 $[1,2|[]] \Rightarrow [1|[2|[]]]$
 $[1,2,3|[]] \Rightarrow [1|[2,3|[]]] \Rightarrow [1|[2|[3|[]]]]$
 - Struktúrakifejezéssé alakítás: $[Fej|Farok] \Rightarrow .(Fej,Farok)$.
 $[1|[2|[]]] \Rightarrow .(1,.(2,[]))$, $[[X|Y]|[]] \Rightarrow .((X,Y),[])$
- Egyéb szintaktikus édesítőszer:
 - Karakterkód-jelölés: $0'Kar$.
 $0'a \Rightarrow 97$, $0'b \Rightarrow 98$, $0'c \Rightarrow 99$, $0'd \Rightarrow 100$, $0'e \Rightarrow 101$
 - Fűzér (string): $"xyz\dots" \Rightarrow$ az $xyz\dots$ karakterek kódját tartalmazó lista
 $"abc" \Rightarrow [97,98,99]$, $" " \Rightarrow []$, $"e" \Rightarrow [101]$
 - Kapcsos zárójelezés: $\{Kif\} \Rightarrow \{ \}(Kif)$ (egy $\{ \}$ nevű, egyargumentumú struktúra — a $\{ \}$ jelpár egy önálló lexikai elem, egy névkonstans).
 - Bináris, hexa stb. alak (csak iso módban), pl. $0b101010$, $0x1a$.

Kifejezések szintaxisa — kétszintű nyelvtanok

- Egy részlet egy „hagyományos” nyelv kifejezés-szintaxisából:

$$\langle \text{kifejezés} \rangle ::= \langle \text{tag} \rangle$$

$$| \langle \text{kifejezés} \rangle \langle \text{additív művelet} \rangle \langle \text{tag} \rangle$$

$$\langle \text{tag} \rangle ::= \langle \text{tényező} \rangle$$

$$| \langle \text{tag} \rangle \langle \text{multiplikatív művelet} \rangle \langle \text{tényező} \rangle$$

$$\langle \text{tényező} \rangle ::= \langle \text{szám} \rangle | \langle \text{azonosító} \rangle | (\langle \text{kifejezés} \rangle)$$
- Ugyanez kétszintű nyelvtannal:

$$\langle \text{kifejezés} \rangle ::= \langle \text{kif } 2 \rangle$$

$$\langle \text{kif } N \rangle ::= \langle \text{kif } N-1 \rangle$$

$$| \langle \text{kif } N \rangle \langle N \text{ prioritású művelet} \rangle \langle \text{kif } N-1 \rangle$$

$$\langle \text{kif } 0 \rangle ::= \langle \text{szám} \rangle | \langle \text{azonosító} \rangle | (\langle \text{kif } 2 \rangle)$$

{az additív ill. multiplikatív műveletek prioritása 2 ill. 1 }

Prolog kifejezések szintaxisa

```

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

```

Kifejezések szintaxisa — folytatás

```

⟨ 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 ⟩

```

Kifejezések szintaxisa — megjegyzések

- A ⟨ kifejezés N ⟩-ben ⟨ köz ⟩ csak akkor kell ha az őt követő kifejezés nyitó-zárójellel kezdődik.

```

| ?- op(500, fx, succ).
yes
| ?- write_canonical(succ(1,2)), nl, write_canonical(succ(1,2)).
succ(' ','(1,2))
succ(1,2)

```

- A { ⟨ kifejezés ⟩ } azonos a { } (⟨ kifejezés ⟩) struktúrával, ez pl. a DCG nyelvtanoknál hasznos.

```

| ?- write_canonical({a}).
{ } (a)

```

- Egy ⟨ füzér ⟩ " jelek közé zárt karaktorsorozat, általában a karakterek kódjainak listájával azonos.

```

| ?- write("baba").
[98,97,98,97]

```

A Prolog lexikai elemei 1. (ismétlés)

- ⟨ név ⟩
 - kisbetűvel kezdődő alfanumerikus jelsorozat (ebben megengedve kis- és nagybetűt, számjegyeket és aláhúzásjelet);
 - egy vagy több ún. speciális jelből (+-*/\\${^<=>~:~.?@#&}) álló jelsorozat;
 - az önmagában álló ! vagy ; jel;
 - a [] { } jelpárok;
 - idézőjelek (') közé zárt tetszőleges jelsorozat, amelyben \ jellel kezdődő escape-szekvenciákat is elhelyezhetünk.
- ⟨ változó ⟩
 - nagybetűvel vagy aláhúzással kezdődő alfanumerikus jelsorozat.
 - az azonos jelsorozattal jelölt változók egy klónon belül azonosaknak, különböző klónokban különbözőeknek tekintődnek;
 - kivétel: a semmis változók (_) minden előfordulása különböző.

A Prolog lexikai elemei 2.

- \langle természetes szám \rangle
 - (decimális) számjegysorozat;
 - 2, 8 ill. 16 alapú számrendszerben felírt szám, ilyenkor a számjegyeket rendre a 0b, 0o, 0x karakterekkel kell prefixálni (csak iso módban)
 - karakterkód-konstans $0'c$ alakban, ahol c egyetlen karakter (vagy egy ilyet jelölő escape-szekvencia)
- \langle lebegőpontos szám \rangle
 - mindenképpen tartalmaz tizedespontot
 - mindkét oldalán legalább egy (decimális) számjeggyel
 - e vagy E betűvel jelzett esetleges exponens

PROLOG PÉLDÁK

Megjegyzések és formázó-karakterek

- Megjegyzések (comment)
 - A % százalékjeltől a sor végéig
 - A /* jelpártól a legközelebbi */ jelpárig.
- Formázó elemek
 - szóköz, újsor, tabulátor stb. (nem látható karakterek)
 - megjegyzés
- A programszöveg formázása
 - formázó elemek (szóköz, újsor stb.) szabadon elhelyezhetők;
 - kivétel: struktúrakifejezés neve után nem szabad formázó elemet tenni;
 - prefix operátor és (közé kötelező formázó elemet tenni;
 - \langle záró-pont \rangle : egy . karakter amit egy formázó elem követ.

A régi jegyzet bevezető példája: útvonalkeresés

- A feladat:
 - Tekintsük (autóbusz)járatok egy halmazát.
 - Mindegyik járhoz a két végpont és az útvonal hossza van megadva.
 - Írjunk Prolog eljárást, amellyel megállapítható, hogy két pont összeköthető-e pontosan N csatlakozó járáttal!
- Átfogalmazás: egy súlyozott irányítatlan gráfban két pont közötti utat keresünk. Élek:


```
% járat(A, B, H): Az A és B városok között van járat, és hossza H km.
járat('Budapest', 'Prága', 515).
járat('Budapest', 'Bécs', 245).
járat('Bécs', 'Berlin', 635).
járat('Bécs', 'Párizs', 1265).
```
- Irányított élek:


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

Az útvonalkeresési feladat — folytatás

- Adott lépésszámú útvonal (él-sorozat) és hossza:

```
% 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.
```

- Futási példa:

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

Körmentes út keresése

- Könyvtár betöltése, adott funktorú eljárások importálásával:

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

- Segéd-argumentum: az érintett városok listája, fordított sorrendben

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

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

- Példa-futás:

```
| ?- útvonal_2(2, 'Párizs', Hová, H).
    H = 1900, Hová = 'Berlin' ? ;
    H = 1510, Hová = 'Budapest' ? ; no
```

Továbbfejlesztés: körmentes út keresése, útvonal-gyűjtéssel

- Az alapötlet: a `Kizártak` listában gyűlik a (fordított) útvonal.

- A rekurzív eljárásban szükséges egy **új argumentum**, hogy az útvonalat kiadjuk!

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

% útvonal_3(N, A, B, Út, H): A és B között van (pontosan)
% N szakaszból álló körmentes Út útvonal, amelynek összhossza H.
útvonal_3(N, Honnan, Hová, Út, H) :-
    útvonal_3(N, Honnan, Hová, [Honnan], FÚt, H),
    reverse(FÚt, Út).

% útvonal_3(N, A, B, FÚt0, FÚt, H): A és B között van pontosan
% N szakaszból álló körmentes, FÚt0 elemein át nem menő H hosszú út.
% FÚt = (az A → B útvonal megfordítása) ⊕ FÚt0.
útvonal_3(0, Hová, Hová, FordÚt, FordÚt, 0).
útvonal_3(N, Honnan, Hová, FordÚt0, FordÚt, H) :-
    N > 0, N1 is N-1, útszakasz(Honnan, Közben, H1),
    \+ member(Közben, FordÚt0),
    útvonal_3(N1, Közben, Hová, [Közben|FordÚt0], FordÚt, H2), H is H1+H2.

| ?- útvonal_3(2, 'Párizs', _, Út, H).
    H = 1900, Út = ['Párizs', 'Bécs', 'Berlin'] ? ;
    H = 1510, Út = ['Párizs', 'Bécs', 'Budapest'] ? ; no
```

Súlyozott gráf ábrázolása éllistával

- A gráf ábrázolása

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

- Típus-definíció

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

- Példa

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

Ismétlődésmentes útvonal keresése listával ábrázolt gráfban

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

% útvonal_4(N, G, A, B, L, H): A G gráfban van egy A-ból
% B-be menő N szakaszból álló L út, melynek összhossza H.
útvonal_4(0, _Gráf, Hová, Hová, [Hová], 0).
útvonal_4(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_4(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).

| ?- hálózat(_Gráf), útvonal_4(2, _Gráf, 'Budapest', _, Út, H).
    H = 880, Út = ['Budapest', 'Bécs', 'Berlin'] ? ;
    H = 1510, Út = ['Budapest', 'Bécs', 'Párizs'] ? ;
    no
```

Deklaratív programozás. BME VIK, 2008. őszi félév

(Logikai Programozás)

Összetett adatstruktúrák konjunktív és diszjunktív bejárása

- Prologban egy összetett adatstruktúrát kétféleképpen lehet bejárni:
 - konjunktívan: a részek bejárása ÉS kapcsolatban van, általában egy eredményt ad
 - pl. fa összegzése (sum_tree), fa ellenőrzése (itree), fa kiírása:


```
% faki(Fa): Fa kiírható (mindig teljesül :-). Mellékhatásként kiírja a Fa fát.
faki(leaf(V)) :-
    write(@), write(V). % A write(X) beépített pred. kiírja az X kifejezést.
faki(node(L,R)) :-
    write(' '), faki(L), write(' -- '), faki(R), write(' ').

| ?- faki(node(node(leaf(1),leaf(8)),leaf(7))). => (@1 -- @8) -- @7
yes
```
 - diszjunktívan: a részek bejárása VAGY kapcsolatban van, visszalépéskor új eredmény
 - pl. fa leveleinek felsorolása (fa_level)
 - A diszjunktív, felsoroló bejárás könnyen kiegészíthető további feltételekkel
 - Keressük egy fának az (5,10) intervallumba eső leveleit:


```
| ?- _Fa = node(node(leaf(1),leaf(8)),leaf(7)), fa_level(_Fa, E), 5 < E, E < 10.
=> E = 8 ? ; E = 7 ? ; no
| ?- _Fa = (...), fa_level(_Fa, E), 5 < E, E < 10, write(E), write(' '), fail.
=> 8 7 => no
```
 - A fail beépített predikátum mindig megghiúsul, pl. ún. visszalépéses ciklus szervezésére jó.

Deklaratív programozás. BME VIK, 2008. őszi félév

(Logikai Programozás)

Bináris fákra vonatkozó példasor — fa levele

- Ismétlés: egészekből álló bináris fa:


```
:- type itree == {node(itree, itree)} \\/ {leaf(int)}.
:- type itree ---> node(itree, itree) | leaf(int).
```
- Írjunk egy predikátumot annak eldöntésére, hogy egy adott érték szerepel-e egy fa levelében (vö. member/2)!


```
% fa_level(Fa, Ertek): A Fa bináris fa levelében szerepel az Ertek szám.
fa_level(leaf(V), V). % ha a fa egyetlen levélből áll és a levélbeli
% érték megegyezik a keresettel, akkor 'siker'

fa_level(node(L,_) , V) :-
    fa_level(L, V). % ha a bal részében van, akkor az egészben is
fa_level(node(_,R) , V) :-
    fa_level(R, V). % ha a jobb részében van, akkor az egészben is
```
- Az aláhúzásjel egy ún. semmis (void) változó, ennek minden előfordulása különböző változó!
- Példák: ellenőrzés (1), adott fa leveleinek felsorolása (2), adott levelű fák felsorolása, (3) (∞ keresési tér).


```
| ?- fa_level(node(node(leaf(1),leaf(2)),leaf(7)), 2). => yes (1)
| ?- fa_level(node(node(leaf(1),leaf(2)),leaf(7)), 3). => no (1)
| ?- fa_level(node(leaf(1),leaf(7)), E). => E = 1 ? ; E = 7 ? ; no (2)
| ?- fa_level(Fa, 3). => Fa = leaf(3) ? ; Fa = node(leaf(3),_A) ? ; ... (3)
```

Deklaratív programozás. BME VIK, 2008. őszi félév

(Logikai Programozás)

Levél elhagyása bináris fából

- Írjunk egy predikátumot annak eldöntésére, hogy egy adott érték szerepel-e egy összetett fa levelében! A predikátum adja vissza a levél elhagyása után fennmaradó fát!


```
% flm(Fa, Ertek, Marad): A Fa összetett bináris fa egy Ertek értékű
% levelének elhagyása után marad a Marad fa. (flm = fa_level_maradek)
flm(node(leaf(V),T), V, T). % ha a bal részfa a keresett levél
% akkor a jobb részfa a maradék
flm(node(T,leaf(V)), V, T). % ugyanez jobboldali levél esetére
flm(node(L0,R), V, node(L,R)) :-
    flm(L0, V, L). % ha a bal részfából elhagyható a levél
% akkor ennek maradéka, kiegészítve
% a jobb részfával, lesz a teljes fa maradéka
flm(node(L,R0), V, node(L,R1)) :-
    flm(R0, V, R1). % ugyanez jobb részfa esetére
```
- Az flm/3 predikátum használható ellenőrzése, de fa szétbontására is:


```
| ?- flm(node(leaf(1),node(leaf(2),leaf(3))), 2, T). =>
    T = node(leaf(1),leaf(3)) ? ; no
| ?- flm(node(leaf(1),node(leaf(2),leaf(3))), 7, T). => no
| ?- flm(node(leaf(1),node(leaf(2),leaf(3))), X, T). =>
    T = node(leaf(2),leaf(3)), X = 1 ? ;
    T = node(leaf(1),leaf(3)), X = 2 ? ;
    T = node(leaf(1),leaf(2)), X = 3 ? ; no
```

Deklaratív programozás. BME VIK, 2008. őszi félév

(Logikai Programozás)

Levél beszúrása bináris fába

- Írjunk egy predikátumot arra, hogy egy adott értékű levelet egy fába minden lehetséges módon beszúrjon!

- Nem kell írunk, már megírtuk! Az `flm` predikátum erre is jó:

```
% flm(Fa, Ertek, Marad): A Fa összetett bináris fa egy Ertek értékű
% levelének elhagyása után marad a Marad fa. Röviden: Fa - Ertek = Marad.

% flm(Fa, Ertek, Marad): A Fa (összetett) bináris fa úgy áll elő, hogy
% a Marad fába beszúrunk egy E értékű levelet. Fa = Marad + Ertek.
flm(node(leaf(V),T), V, T). % Egy T fába beszúrhatunk egy levelet
(...) % úgy, hogy az egylevelű fát T elé tesszük
```

- Példák:

```
| ?- flm(Fa, 2, leaf(1)), faki(Fa), write(' '), fail.
(@2 -- @1) (@1 -- @2) => no
| ?- flm(Fa0, 2, leaf(1)), flm(Fa, 3, Fa0), faki(Fa), write(' '), fail.
(@3 -- (@2 -- @1)) (@2 -- @1) -- @3) ((@3 -- @2) -- @1) ((@2 -- @3) -- @1)
(@2 -- @3 -- @1) (@2 -- @1 -- @3) (@3 -- @1 -- @2) ((@1 -- @2) -- @3)
((@3 -- @1) -- @2) ((@1 -- @3) -- @2) (@1 -- @3 -- @2) (@1 -- @2 -- @3)) => no

negylevelu(X, Y, Z, U, Fa) :- % Fa az X, Y, Z, U levelekből áll
    flm(Fa0, Y, leaf(X)), flm(Fa1, Z, Fa0), flm(Fa, U, Fa1).

| ?- findall(Fa, negylevelu(1,3,4,6,Fa), Fak), length(Fak,Db). => Db = 120, Fak = (...)
```

Példa: adott értékű kifejezés előállítása

- A feladat: írjunk Prolog programot a következő feladvány megoldására:

- Az 1, 3, 4, 6 számokból a négy alpművelet felhasználásával állítsuk elő a 24 számértéket!
- Mind a négy számot fel kell használni, tetszőleges sorrendben.
- Tetszőleges alpműveletek használhatók, tetszőleges zárójelezzel.

- Már van egy predikátumunk (`negylevelu/5`), amely adott számokból tetszőleges fát épít.

- Definiáljunk egy predikátumot, amely egy fának megfelelő aritmetikai kifejezéseket készít!

```
% fa_kif(Fa, Kif): Kif a Fa fával azonos alakú, azonos számokból álló
% aritmetikai kifejezés, amelyben a négy alpművelet fordulhat elő.
fa_kif(leaf(V), V).
fa_kif(node(L,R), Exp) :-
    fa_kif(L, E1),
    fa_kif(R, E2),
    alap4(E1, E2, Exp).

% alap4(X, Y, Kif): Kif az X és Y kifejezésekből a négy alpművelet egyikével áll elő.
alap4(X, Y, X+Y). % alap4(X, Y, X-Y).
alap4(X, Y, X*Y). % alap4(X, Y, X/Y).

| ?- fa_kif(node(leaf(1),node(leaf(2),leaf(3))), Kif).
Kif = 1+(2+3) ? ; Kif = 1-(2+3) ? ; Kif = 1*(2+3) ? ; Kif = 1/(2+3) ? ;
(...)
Kif = 1+2/3 ? ; Kif = 1-2/3 ? ; Kif = 1*(2/3) ? ; Kif = 1/(2/3) ? ; no
```

Példa: adott értékű kifejezés előállítása (folyt.)

- Korábban elkészített predikátumok:

- adott számokból álló fákat felsoroló `negylevelu/5`
- adott fával azonos szerkezetű aritmetikai kifejezéseket felsoroló `fa_kif/2`

- Ezeket építve könnyen megírható a feladvány megoldására használható predikátum:

```
% Kif egy a négy alpművelettel az X, Y, Z, U számokból
% felépített kifejezés, amelynek értéke Ertek.
negylevelu_erteke(X, Y, Z, U, Ertek, Kif) :-
    negylevelu(X, Y, Z, U, Fa),
    fa_kif(Fa, Kif),
    Kif =:= Ertek.
```

```
| ?- negylevelu_erteke(1,3,4,6,24,Kif).
...
```

- Megjegyzések

- Az aritmetikai eljárásokban a változók nem csak számokra, hanem tömör aritmetikai kifejezésekre is be lehetnek helyettesítve.
- A `negylevelu_erteke` eljárás utolsó hívása helyett **nem** lenne jó: `Ertek is Kif. Miért?`