Marike,

ktorá mi vedela vždy pripomenúť, že život ponúka aj iné potešenia, než vysedávanie za terminálom počítača.



Lektori: RNDr. Olga Štěpánková, CSc.
Ing. Marián Mach

VYSOKÁ ŠKOLA TECHNICKÁ V KOŠICIACH
ELEKTROTECHNICKÁ FAKULTA

---

OKRESNÁ STANICA MLADÝCH TECHNIKOV
ROŽŇAVA

Július C S O N T Ó

# PROLOG V PRÍKLADOCH

PRE SINCLAIR ZX SPECTRUM

# OBSAH

# PREDSLOV

Predkladaný text bol pôvodne určený poslucháčom Elektrotechnickej fakulty ako návod na cvičenia z predmetu Umelá inteligencia. Bol však koncipovaný tak, aby mohol slúžiť aj ako samostatná učebnica jazyka PROLOG. Vo výuke uvedeného predmetu sa osvedčil PROLOG v úlohe výkladového jazyka. S výhodou sa využíva jeho deklaratívnosť, t.j. potreba formulovať iba ciele, ktoré sa majú splniť bez nutnosti programovať spôsob ako toho dosiahnuť. Prostredníctvom PROLOGu sa prostriedky logiky dostávajú priamo do rúk programátora, pričom tieto prostriedky nie sú iba nástrojom ale aj účinnou metódou. Vysoká forma abstrakcie sa tu spája s formalizmom orientovaným na ľudský spôsob uvažovania a chápania. Tieto skutočnosti spolu s publicitou, ktorú PROLOG získal v roku 1981, keď ho Japonci vybrali za nosný jazyk počítačov 5.generácie vzbudzujú nádej že sa tento jazyk stane v blízkej budúcnosti vyhľadávaným programovacím prostriedkom.

Prvých päť kapitol obsahuje neformálny výklad základov jazyka, ilustrovaný početnými príkladmi. Ďalších šesť kapitol predstavuje rôzne aplikačné možnosti jazyka, pričom niektoré typické problémové okruhy (spracovanie prirodzeného jazyka, databázové systémy, atď.) boli z priestorových dôvodov vynechané. Prílohy obsahujú informácie potrebné pri praktickom používaní jazyka. Oproti pôvodnému textu bola doplnená PRÍLOHA 4, obsahujúca implementačne závislé informácie o verzii PROLOG-80 pre počítače ZX Spectrum.

Autor je zaviazaný vďakou Ing.P.Novákovi za cenné rady a podnety pri príprave textu, RNDr.O.Štepánkovej,CSc. a Ing.M.Machovi za starostlivé prečítanie rukopisu a za mnohé námety, ktoré boli v konečnej verzii rukopisu zohľadnené. Vďaka autora patrí aj V.Javorčíkovi, poslucháčovi 4.ročníka EF, za napísanie konverzného programu, pomocou ktorého bolo možné zaručiť, že v texte sa nachádzajú priamo zdrojové texty fungujúcich programov.

V Košiciach, 15.7.1988 Autor

# 1. DIALÓG SO SYSTÉMOM PROLOG

## 1.1. ZÁKLADNÉ POJMY

Programovanie v PROLOG-u pozostáva z dvoch krokov:

A. Vytváranie databázy obsahujúcej informácie o objektoch reálneho sveta a o vzťahoch medzi nimi. Databáza je vlastne prologovský "program". Jednotlivé informácie sú reprezentované tzv. klauzulami. Klauzuly v databáze môžu byť dvojakého typu:

A1. Fakty – vyjadrujú základné informácie o vlastnostiach objektov a o vzťahoch medzi objektami

A2. Pravidlá – všeobecné konštatovania o objektoch a o vzťahoch medzi nimi

Skupina klauzúl, vzťahujúcich sa na jeden typ informácie predstavuje definíciu predikátu.

B. Formulácia otázok o objektoch a vzťahoch medzi nimi. Prologovský "výpočet" je vlastne dialóg, v ktorom používateľ kladie systému otázky a systém na tieto otázky odpovedá na základe informácií, obsiahnutých v databáze. Systém má pritom schopnosť odvodzovať niektoré vzťahy, ktoré explicitne v databáze obsiahnuté nie sú.

## 1.2. REPREZENTÁCIA FAKTOV

Fakt je tvorený štruktúrou

```
funktor(arg1, arg2, ... , argN).
```

kde funktor reprezentuje vlastnosť, resp. vzťah a kde argumenty predstavujú jednotlivé objekty. Každý fakt musí byť ukončený bodkou, za ktorou musí nasledovať aspoň jedna medzera, alebo znak nového riadku.

Príklad 1.1.

Zapíšte v jazyku PROLOG nasledujúce výroky: "Dana je dievča" a "Jana ľúbi pivo".

Riešenie:

Prvý výrok obsahuje informáciu o jednej vlastnosti (je dievča) objektu (Dana). Druhý výrok vyjadruje vzťah (ľúbi) dvoch objektov (Jana a pivo):

```
dievca(dana).        lubi(jana,pivo).
```

Atómy (odpovedajúce približne textovým konštantám procedurálnych jazykov) začínajú v PROLOGu malými písmenami. Považujeme ich za štruktúry s nulovým počtom argumentov.

- *** -

Argumentami štruktúry môžu byť ľubovoľné objekty jazyka, teda aj štruktúry. To umožňuje vytvárať v zásade ľubovoľne zložité, hierarchicky členené štruktúry.

Príklad 1.2.

Vyjadrite v PROLOGu výrok "Peter ľúbi plzenské pivo".

Riešenie:

Základom výroku je binárny vzťah "Peter ľúbi pivo", ktorý je doplnený vlastosťou "pivo je plzenské". Pôvodný výrok je potom možné reprezentovať pomocou hierarchickej štruktúry (pre ujasnenie hierarchie je výhodné štruktúru zobraziť pomocou stromu):

```
                          lubi
                           / \
                          /   \
lubi(peter,pivo(plzenske))  peter pivo
                                 |
                                 |
                              plzenske
```

## 1.3. OPERÁTOROVÝ ZÁPIS ŠTRUKTÚRY

V prípade jedného, alebo dvoch argumentov je možný aj čitateľnejší, tzv. operátorový zápis štruktúry, ovšem za predpokladu, že funktor je deklarovaný ako operátor. Pri deklarácii sa zadávajú tri príznaky, charakterizujúce každý operátor:

1. precedencia - celé číslo, udávajúce, v akom poradí sa majú aplikovať operátory, keď toto poradie nie je explicitne vyjadrené zátvorkami (čím je toto číslo väčšie, tým slabšia je väzba operátora s operandami (napr. keď precedencie ± a * sú 31 a 21, potom a*b+c*d = (a*b)+(c*d) )

2. pozícia - atóm, udávajúci polohu operátora (f) vzhľadom k operandom (x,y); operátor teda môže byť:
   - fx - prefixový
   - xf - postfixový
   - xfx - infixný, ktorý nemá zmysel reťaziť
   - xfy - vpravo asociatívny infixný, napr. a.b.c = a.(b.c)
   - yfx - vľavo asociatívny infixný, napr. a/b/c = (a/b)/c \= a/(b/c)

3. meno - atóm, zhodný s funktorom štruktúry

Vlastná deklarácia sa robí pomocou štandardného predikátu op s vyššie uvedenými tromi argumentami.

Príklad 1.3.

Navrhnite operátorový zápis výrokov z príkladov 1.1 a 1.2.

Riešenie:

Deklarujme nasledujúce operátory:

op(6,xf,pivo)  op(8,xfx,lubi)  op(8,xf,je_dievca)

Potom nasledujúce dvojice zápisov sú rovnocenné:

dievca(dana)  dana je_dievca

lubi(jana,vino)  jana lubi vino

lubi(peter,pivo(plzenske))  peter lubi plzenske pivo

## 1.4. DIALÓG POUŽÍVATEĽA SO SYSTÉMOM

Po zavedení systému PROLOG sa objaví hlavička a nápovedný atóm 2- , ktorá vždy znamená, že systém očakáva používateľovu otázku (musí byť ukončená vždy bodkou a stlačením klávesy <NL>).

**Príklad 1.4.**

Zistite, či 3 je väčšie, ako 2 a či 5 je rovné 4.

Riešenie:
Dialóg so systémom bude nasledovný:

```
2- 3>2.<NL>
yes

2- 5=4.<NL>
no
```

Z predošlého dialógu je zrejmé, že systém "pozná" niektoré základné vzťahy z reálneho sveta vo forme tzv. štandardných predikátov, napr. $\equiv$ , $\geq$ , atď. Pritom komunikácia so systémom prebieha bez akýchkoľvek explicitných príkazov čítania a výpisu.

- *** -

"Svet" štandardných predikátov je ale veľmi obmedzený. Keď používateľ chce riešiť úlohy v zložitejšom "svete", musí uložiť do databázy PROLOG-u nové vzťahy, charakterizujúce tento "svet". Príslušné vzťahy sa zadávajú vo forme klauzúl. Do databázy ich môžeme načítať (v prologovskom slangu sa hovorí "nakonzultovať") z nejakého súboru. Okrem súborov na vonkajších pamätiach pozná PROLOG ešte fiktívny súbor user - v prípade vstupu je to klávesnica a v prípade výstupu obrazovka.

Zatiaľ zvolíme najjednoduchší zápis klauzúl do databázy priamo z klávesnice. Koniec zápisu signalizujeme implementačne závislou klávesou pre znak konca súboru <EOF> :

```
?- [user].<NL>
lubi(peter,mara).  lubi(jana,pivo).<NL>
lubi(dana,vino).  lubi(peter,pivo).<NL>
dievca(dana). dievca(jana). dievca(mara).<NL>
<EOF>
user consulted
yes
```

V prípade porušenia syntaxe systém hlási chybu, zlú klauzulu nezaradí a číta ďalšiu klauzulu. Napríklad časť predošlého dialógu by mohla byť nasledovná:

```
?- [user].<NL>
. . .
lubi().<NL>                          chybná klauzula
[Syntax error - subterm missing]     chybové hlásenie
lubi()^.                             označenie miesta chyby
lubi(peter,pivo).<NL>                správna klauzula
. . .
<EOF>
user consulted
yes
```

V obidvoch prípadoch je výsledkom dialógu správne uloženie všetkých klauzúl do databázy.

Príklad 1.5.

Zistite, či Mara a Viera sú dievčatá.

Riešenie:

Odpoveď zistíme v priebehu dialógu (pre objasnenie uvádzame ešte naviac v apostrofoch prepisy otázok používateľa a v úvodzovkách význam odpovedí systému):

```
?- dievca(mara).<NL>        'je Mara dievča ?'
yes                         "áno"

?- dievca(viera).<NL>       'je Viera dievča ?'
no                          "nemám informáciu o tom, či
                             Viera je dievča"
```

## 1.5. POUŽITIE PREMENNÝCH PRI KLADENÍ OTÁZOK

V dialógu je možné klásť aj otázky, v ktorých sa vyskytujú opytovacie zámená. Týmto zámenám odpovedajú pri formálnom prepise premenné ( musia začínať veľkým písmenom alebo podčiarnikom). Keď sa v otázke vyskytuje premenná, potom systém vo svojej odpovedi oznámi aj objekt, na ktorý sa táto premenná naviazala. Po tomto ozname sa očakáva pokyn používateľa, ktorý môže byť:

1. stlačenie klávesy <NL> (nepožaduje sa hľadanie prípadného ďalšieho riešenia)

2. bodkočiarka a <NL> (požaduje sa hľadanie ďalšieho riešenia)

Príklad 1.6.

Zistite kto všetko je dievča, koho ľúbi Peter, kto ľúbi pivo a kto ľúbi koho.

Riešenie:

```
?- dievca(D).<NL>          'ktoré dievčatá poznáš ?'
D = dana ;<NL>             "Dana" 'a ešte ?'
D = jana ;<NL>             "Jana" 'a ešte ?'
D = mara ;<NL>             "Mara" 'a ešte ?'
no                         "ďalšie dievča nepoznám"

?- lubi(peter,Koho).<NL>   'koho ľúbi Peter'
Koho =  mara ;<NL>         "Maru" 'a ešte ?'
Koho =  pivo ;<NL>         "pivo" 'a ešte ?'
no                         " už nikoho viac"

?- lubi(Kto,pivo).<NL>     'Kto ľúbi pivo ?'
Kto = jana ;<NL>           "Jana" 'a ešte?'
Kto = peter ;<NL>          "Peter" 'a ešte ?'
no                         "už nikto viac"
```

| | |
|---|---|
| ?- lubi(Kto,Koho).<NL> | 'Kto ľúbi Koho ?' |
| Kto = peter , | "Peter ľúbi |
| Koho = mara ;<NL> | Maru" 'a ešte ?' |
| Kto = jana , | "Jana ľúbi |
| Koho = pivo <NL> | pivo" 'to stačí ' |
| yes | "áno, zodpovedal som tvoju |
| | otázku" |

## 1.6. FORMULÁCIA ZLOŽENÝCH OTÁZOK

Uvedené jednoduché otázky možno spájať pomocou dvoch spojek: spojke a v PROLOGu odpovedá čiarka a spojke alebo bodkočiarka.

Príklad 1.7.

Zistite, či sú pravdivé výroky "Peter ľúbi pivo a víno" a "Peter ľúbi pivo, alebo víno".

Riešenie:

| | |
|---|---|
| ?- lubi(peter,pivo), | |
| lubi(peter,vino).<NL> | 'ľúbi Peter aj pivo aj víno ?' |
| no | "nie" |
| ?-lubi(peter,pivo); | |
| lubi(peter,vino).<NL> | 'ľúbi Peter pivo, alebo víno ?' |
| yes | "ano" |

- *** -

Položenie otázky v PROLOG-u (chápanej ako cieľ, ktorý sa má splniť) je do istej miery analogické volaniu procedúry v iných programovacích jazykoch. Ako je však vidieť z dialógov, v jazyku PROLOG nie je jednoznačne určené, ktorý parameter "procedúry" je vstupný ( t.j. v okamžiku volania definovaný) a ktorý výstupný. To umožňuje využívať ten istý predikát na rôzne účely: dievca môže testovať, či niekto je dievča, alebo môže generovať všetky dievčatá. Predikát lubi možno využívať štvorako (príklad 1.6).

Je ovšem potrebné si uvedomiť, že predikáty so zhodným menom, ale s rozdielnym počtom argumentov sa pokladajú v PROLOGu za

odlišné, takže nasladujúce fakty definujú dva rôzne predikáty:

lubi(peter,kovac,pivo).  lubi(kovac,pivo).

## 1.7. POUŽÍVANIE PRAVIDIEL

Ako už bolo v úvode spomenuté, všeobecné konštatovania o objektoch a o vzťahoch medzi nimi sa v PROLOGu reprezentujú pomocou pravidiel. Jedná sa o výroky typu "keď T, potom H", reprezentované štruktúrami s infixným operátorom :- :

hlava :- telo.

Hlavu tvorí štruktúra, reprezentujúca nadradený cieľ H (funktor udáva meno predikátu, definovaného týmto pravidlom). Telo je štruktúra, reprezentujúca podmienku I pre splnenie nadradeného cieľa. V tele pravidla sú jednotlivé dielčie štruktúry (reprezentujúce príslušné subciele) spojené operátormi konjunkcie ( a - čiarka) a disjunkcie ( alebo - bodkočiarka), pričom je možné použiť aj negáciu ( nie - not ). Telo pravidla však môže obsahovať niekedy iba jeden subcieľ. Pravidlo musí byť ukončené bodkou, za ktorou nasleduje znak nového riadku, alebo aspoň jedna medzera.

Príklad 1.8.

Zapíšte v PROLOGu: "Peter ľúbi dievčatá, ktoré ľúbia pivo".

Riešenie.

Vyjadríme zadaný výrok trochu formálnejšie: "keď D je dievča a D ľúbi pivo, potom Peter ľúbi D" čo sa už dá prepísať do PROLOGu nasledovne:

lubi(peter,D) :- dievca(D), lubi(D,pivo).

Premenná D slúži k odovzdávaniu informácií medzi jednotlivými subcieľmi tela pravidla, ako aj medzi telom a hlavou pravidla. Premenná je ovšem lokálna v rámci pravidla - medzi pravidlami nemozno pomocou nej prenášať žiadne informácie.

## 1.8. NÁVRH JEDNODUCHEJ DATABÁZY

**Príklad 1.9.**

Navrhnite prologovskú databázu, reprezentujúcu rodinné vzťahy naznačené stromom na obr. 1.1.

```
                    uza--+--karol
                         |
                   +-----+-----+
                   |     |     |
          jano--+--jana  dana  milan-----eva
                |
            +---+---+
            |       |
          peter    anna--+--jozo
                         |
                       viera
```

obr. 1.1

Riešenie:

Zaveďme predikát rodicia s tromi argumentami: Matka, Otec, Dieťa. Tento predikát je možné definovať v súlade so zadaným stromom pomocou nasledujúcich faktov:

```
rodicia(zuza,karol,jana    ).   rodicia(zuza,karol,dana ).
rodicia(zuza,karol,milan   ).   rodicia(jana,jano ,peter).
rodicia(jana,jano ,anna    ).   rodicia(anna,jozo ,viera).
rodicia(eva ,milan,bezdetni).
```

Bolo potrebné zaviesť špeciálne označenie bezdetni , aby sme mohli pomocou predikátu rodicia reprezentovať aj bezdetné manželské páry.

**Príklad 1.10.**

Pomocou databázy z predošlého príkladu zistite, kto je Petrov dedo a pokúste sa zovšeobecniť vzťah "dedo - vnúča".

Riešenie:

Petrovho deda zistíme v priebehu dialógu:

```
?- (rodicia(Rodic,_,peter); rodicia(_,Rodic,peter)),
   rodicia(_,Dedo,Rodic).
Rodic = jana, Dedo = karol ;<NL>
no
```

Prvý riadok otázky znamená "Rodic je matka alebo otec Petra" (disjunkciu je potrebné uzavrieť do zátvoriek, nakoľko bodkočiarka slabšie viaže operandy, ako čiarka nasledujúcej konjunkcie). A súčasne (konjunkcia) s touto prvou požiadavkou má platiť, že "Dedo je otcom Rodica" (druhý riadok otázky). Vo svojej odpovedi systém oznámil riešenie, ktoré našiel. Bodkočiarka a <NL> vyjadrujú používateľov pokyn, aby sa systém pokúsil nájsť ďalšie riešenie. Tento pokus bol neúspešný, čo systém oznámil lakonickým no . Podčiarnikmi sme označili v otázke anonymné premenné (premenné, na ktorých "nezáleží", ktoré sa vyskytujú v otázke iba raz).

Vypisovanie podobných dlhých otázok je nepohodlné. Naviac je možné zložitejšie otázky s výhodou hierarchicky členiť na podotázky. Je účelné zaviesť pomocné predikáty - vlastná podotázka tvorí telo pravidla, definujúceho ten ktorý predikát. V našom príklade naznačený rozklad vedie na tieto štyri pravidlá:

```
   otec(Otec, Dieta) :- rodicia(_,Otec, Dieta), Dieta\=bezdetni.
  matka(Matka,Dieta) :- rodicia(Matka,_,Dieta), Dieta\=bezdetni.
  rodic(Rodic,Dieta) :- otec(Rodic,Dieta) ; matka(Rodic,Dieta).
   dedo(Dedo, Vnuca) :- otec(Dedo,R), rodic(R,Vnuca).
```

Uvedené pravidlá sú "čitateľnejšie", ako pôvodná otázka a naviac tri prvé - pomocné - pravidlá sa budú hodiť pri definovaní ďalších predikátov, charakterizujúcich iné rodinné vzťahy. Aj tu sme použili anonymné premenné všade tam, kde neprenášajú informáciu medzi jednotlivými časťami pravidla (prvý argument subcieľa rodicia v tele prvého pravidla sme síce mohli označiť premennou Matka , ale táto premenná nikde inde v uvažovanom pravidle nevystupuje - táto premenná by ovšem nemala vôbec nič spoločné s premennou rovnakého mena v ďalšom pravidle).

Bez testov na konci prvých dvoch pravidiel by systém dával nezmyselné odpovede, napr.:

```
?- otec(milan,Dieta).
Dieta = bezdetni <NL>      /* nechceme hľadať ďalšie riešenia*/
yes
```

Odpoveď na pôvodnú otázku teraz získame jednoduchšie:

```
?- dedo(Dedo,peter).
Dedo = karol <NL>          /* nechceme hľadať ďalšie riešenia */
yes
```

<u>Príklad 1.11.</u>

Definujte predikáty pre vyhľadanie bratov a sestier v rodine z príkladu 1.9.

Riešenie:

Podobne ako v predošlom príklade definujeme brata a sestru s využitím pomocných predikátov <u>surodenci</u> , <u>muz</u> a <u>zena</u> :

```
surodenci(Sur1,Sur2)        :- rodicia(Matka,Otec,Sur1),
                               rodicia(Matka,Otec,Sur2),
                               Sur1\=Sur2.
muz(Otec)                   :- rodicia(_,Otec,_).
zena(Matka)                 :- rodicia(Matka,_,_).
sestra(Sestra,Surodenec)    :- surodenci(Sestra,Surodenec),
                               zena(Sestra).
brat(Brat,Surodenec)        :- surodenci(Brat,Surodenec),
                               muz(Brat).
```

Uvedené predikáty dávajú v niektorých špeciálnych prípadoch chybné výsledky:

```
?- brat(peter,anna).
no
?- zena(viera).
no
```

Pre odstránenie uvedeného nedostatku je potrebné pôvodnú databázu doplniť o ďalšie fakty, vďaka ktorým aj slobodní chlapci budú považovaní za mužov a slobodné dievčatá za ženy:

```
rodicia(viera, slob,  bezdetni).
rodicia(slob,  peter, bezdetni).
rodicia(dana,  slob,  bezdetni).
```

Je ovšem potrebné doplniť testy do tela niektorých pravidiel:

```
muz(Otec)        :- rodicia(_,Otec,_),  Otec  \= slob.
zena(Matka)      :- rodicia(Matka,_,_), Matka \= slob.
```

Príklad 1.12.

Navrhnite definície predikátov pre niektoré ďalšie vzťahy.

Riešenie:

Použijeme už definované pomocné predikáty a zavedieme ešte dva ďalšie - manzelia a druhe_koleno :

```
manzelia(Ona,On)            :- rodicia(Ona,On,_),
                               Ona\=slob, On\=slob.
svokra(Svokra,ZatNevesta)   :- matka(Svokra,Partner),
                               (manzelia(Partner,   ZatNevesta) ;
                                manzelia(ZatNevesta,Partner)    ).
teta(Teta,NeterSynovec)     :- sestra(Teta,Rodic),
                               rodic(Rodic,NeterSynovec).
druhe_koleno(SB1,SB2)       :- rodic(RA,SB1), rodic(RB,SB2),
                               surodenci(RA,RB).
sesternica(Sesternica,DK)   :- druhe_koleno(Sesternica,DK),
                               zena(Sesternica).
svagrina(Svagrina,S)        :- (manzelia(Svagrina,X),brat(X,S)) ;
                                (sestra(Svagrina,Y)            ,
                                 (manzelia(Y,S);manzelia(S,Y) ) ).
```

- *** -

Niektoré z uvedených predikátov sa chovajú pri generovaní alternatív trochu nekorektne. Napríklad na otázku muz(X) systém odpovie, že Karol je "trojnásobný" muž, atď. Súvisí to s neefektívnou formou reprezentácie rodičovských vzťahov. V príklade 2.3 bude tento nedostatok odstránený.

# 2. REKURZIE V PROLOGU

## 2.1. POJEM REKURZIE

O rekurzívnej definícii hovoríme, keď na definovanie nejakého objektu použijeme jednoduchší variant toho istého objektu. Použitie jednoduchšieho variantu zaistí, že nevznikne nekonečná cyklická definícia, ale že celý postup konverguje k nejakému hraničnému prípadu, ktorý musí byť samostatne definovaný. Známym príkladom je rekurzívna definícia faktoriálu:

(n+1)! = (n+1)*n!  n=1,2,...(rekurzívna definícia)

1! = 1  (hraničný prípad)

V porovnaní s procedurálnymi jazykmi (z ktorých niektoré rekurzie ani nepripúšťajú, napr. FORTRAN), je v PROLOGu rekurzia používaná oveľa častejšie. Nie je snáď prehnané tvrdenie, že každý netriviálny prologovský program používa rekurziu. V podstate možno rozlíšiť dva spôsoby použitia rekurzie:

- rekurzívna definícia predikátov - rekurzívne pravidlá
- rekurzívna definícia údajových štruktúr - zoznamov.

## 2.2. REKURZÍVNE PRAVIDLÁ

Pri rekurzívnej definícii predikátov sa v tele pravidla vyskytuje ten istý predikát ako v hlave, t.j. hlavný cieľ môže byť splnený iba vtedy, keď jeho jednoduchšia "kópia" ako subcieľ bola splnená.

Príklad 2.1.

Navrhnite predikát pre vyhľadanie potomkov v databáze rodiny z kapitoly 1.8.

Riešenie:

Je síce možná nerekurzívna definícia:

```
potomok(Dieta,Rodic)        :- rodic(Rodic,     Dieta).
potomok(Vnuca,Staryrodic)   :- rodic(Staryrodic,Rodic),
                               rodic(Rodic,     Vnuca).
potomok(Pravnuca,Prarodic):- rodic(Prarodic,Staryrodic),
                               rodic(Staryrodic,Rodic),
                               rodic(Rodic,     Pravnuca).

. . .                       /* ďalšie generácie */
```

ale oveľa elegantnejšia je rekurzívna definícia:

```
potomok(Dieta,  Rodic)      :- rodic(Rodic,     Dieta).
potomok(Potomok,Prapredok)  :- rodic(Prapredok,Predok),
                               potomok(Potomok,Predok).
```

Prvá klauzula ošetruje hraničnú situáciu - medzi prvým a druhým argumentom je rozdiel jednej generácie. Druhá klauzula predpokladá väčší generačný rozdiel - jej použitím sa rekurzívne "uberá" vždy jedna generáciu tak dlho, kým nenastane spomínaná hraničná situácia. Schematicky je tento postup znázornený na obr. 2.1.

```
|<--- potomok(Potomok,Prapredok) :-        ---------------->|
|                                                           |
|     rodic(Prapredok,Predok)  ,  potomok(Potomok,Predok). |
|<---------------------------->|<------------------------->|
|                              |                            |
| Prapredok    --------->    Predok  -->  ...  -->  Potomok |
```

obr. 2.1

V konkrétnom prípade napr. pre <u>potomok(viera,karol)</u> vyzerá naznačený postup takto:

| RODIČ | | DIEŤA | | | | | |
|---|---|---|---|---|---|---|---|
| karol | --> | jana | --> | anna | --> | viera | 1.rekurzívny krok |
| jana | --> | anna | --> | viera | | | 2.rekurzívny krok |
| anna | --> | viera | | | | | hraničná situácia |

Použitie predikátu je znovu mnohostranné:

```
?- potomok(zuza,  jana).          /* Je Zuza Janin potomok ? */
no

?- potomok(Kto,karol).            /* Kto sú potomkovia Karola? */
Kto = jana ;<NL>
Kto = dana ;<NL>
Kto = milan ;<NL>
Kto = peter ;<NL>
Kto = anna ;<NL>
Kto = viera ;<NL>
no

?- potomok(milan,Kto).            /* Kto sú predkovia Milana ? */
Kto = karol ;<NL>
Kto = zuza ;<NL>
no

?- potomok(Kto,Komu).             /* Kto je komu potomkom ? */
Kto = jana ,Komu = karol ;<NL>
Kto = jana ,Komu = zuza <NL>      /* stačia dve riešenia */
yes
```

- *** -

Použitie rekurzívnych pravidiel je v PROLOGu bežné a prirodzené, ale mnohým začiatočníkom a programátorom, odchovaným na jazykoch typu FORTRAN spôsobujú rekurzie značné ťažkosti. Najčastejšími chybami sú:

a. cyklická definícia:

```
otec(O,S) :-  syn(S,O).
syn(S,O) :- otec(O,S).
```

b. nesprávne poradie faktov a pravidiel v databáze (nestačí totiž uviesť všetky relevantné informácie o danej úlohe, je potrebné mať na zreteli aj spôsob práce PROLOGu, najmä postup pri prehľadávaní databázy a pravidlá pre viazanie premenných, viď kapitola 3 ); keď sa napríklad prehodí poradie klauzúl v

rekurzívnej definícii predikátu potomok (riešenie príkladu 2.1), systém sa pri pokuse o zodpovedanie otázky dostane do nekonečnej slučky

c. nesprávne poradie subcieľov v tele rekurzívneho pravidla; napríklad keď v tele rekurzívneho pravidla definujúceho predikát potomok zmením poradie subcieľov na potomok(Potomok,Predok), rodic(PraPredok,Predok) systém správne zodpovie otázky s kladnou odpoveďou, ale zacyklí sa pri hľadaní zápornej odpovede

## 2.3. REKURZÍVNE ÚDAJOVÉ ŠTRUKTÚRY - ZOZNAMY

Zoznam je usporiadaná postupnosť prvkov, ktorá môže mať ľubovoľnú dĺžku, pričom prvkom môže byť akýkoľvek term. "Usporiadaná" znamená, že poradie prvkov je významné. Zoznam sa hodí na zoskupenie informácií, ktoré nejak spolu súvisia, ale ich počet sa môže prípad od prípadu líšiť (t.j. nemožno pre tieto účely použiť bežný predikát).

Príklad 2.2.

Navrhnite "komprimovanú" formu definície predikátu lubi z kapitoly 1.4.

Riešenie:
Je nevhodné sústrediť informácie o tom, kto koho/čo ľúbi nasledovne:

```
lubi(peter,mara,pivo).
lubi(jana,pivo).
lubi(mara).      /* mara nelubi nikoho/nic */
```

nakoľko kvôli rozdielnemu počtu argumentov sa jedná vlastne o tri naprosto rozdielne predikáty. Je ovšem možný zápis pomocou zoznamov:

```
ma_rad(peter,[mara,pivo]).
ma_rad(jana,[pivo]).
ma_rad(mara,[]).
```

pričom posledný fakt reprezentuje výrok "Mara nemá rada nič/ni-

koho". Použili sme úmyselne iné meno predikátu, aby bolo možné zavedením nového pravidla:

```
lubi(Kto,Koho):- ma_rad(Kto, ZoznamLasok),
                 member(Koho,ZoznamLasok).
```

dosiahnuť podobné správanie systému, ako v príklade 1.6 (zmení sa poradie odpovedí systému na niektoré otázky). Neštandardný predikát member odpovedá výroku "prvý argument je prvkom zoznamu v druhom argumente", a nakoľko pracuje s rekurzívnou údajovou štruktúrou, jeho definícia je tiež rekurzívna (viď príklad 2.6).

Príklad 2.3

Navrhnite definíciu predikátu rodina s tromi argumentami (Matka, Otec, ZoznamDeti) tak, aby reprezentoval strom rodiny z príkladu 1.9.

Riešenie:

```
rodina( zuza,  karol, [jana,dana,milan] ).
rodina( jana,  jano,  [peter,anna]      ).
rodina( anna,  jozo,  [viera]           ).
rodina( eva,   milan, []                ).
rodina( viera, slob,  []                ).
rodina( slob,  peter, []                ).
rodina( dana,  slob,  []                ).
```

Aj pre takto zkódovaný rodinný strom je možné použiť väčšinu pôvodných predikátov z príkladov 1.10 - 1.12 a 2.1, keď predefinujeme predikát rodicia zo skupiny faktov na pravidlo

```
rodicia(Matka,Otec,Dieta) :- rodina(Matka,Otec,ZoznamDeti),
                             member(Dieta,ZoznamDeti).
```

Niekoľko málo ďalších zmien vyplýva zo skutočnosti, že bezdetné rodiny sú charakterizované prázdnym zoznamom v treťom argumente (atóm bezdetni je už zbytočný). Predikát member je automaticky nesplnený pre prázdny zoznam, vďaka čomu možno vypustiť pôvodné explicitné testy na bezdetnosť:

```
otec(Otec,  Dieta)     :- rodicia(  _  , Otec, Dieta).
matka(Matka, Dieta)    :- rodicia(Matka,  _  , Dieta).
```

Naviac je vhodné v definícii predikátov muz, zena, manzelia nahradiť subcieľ rodicia subcieľom rodina - odstránia sa tým "násobné" odpovede (viď poznámku za príkladom 1.12).

- *** -

Aj v predošlom príklade bol potrebný predikát member . Skôr než sa ho pokúsime definovať (príklad 2.6), je potrebné trochu bližšie pojednať o rôznych formách zápisu zoznamov.

Doteraz sme používali tzv. zoznamovú notáciu, v ktorej sa prvky zoznamu, oddelené čiarkami, uvedú v danom poradí medzi hranatými zátvorkami. V skutočnosti predstavuje neprázdny zoznam bežnú binárnu štruktúru, ktorej funktorom je bodka (je definovaný ako vpravo asociatívny infixný operátor). Prvým operandom je hlava zoznamu a druhým telo zoznamu. Hlava je prvý prvok zoznamu a telo je obvykle zoznam (z pôvodného zoznamu vznikne oddelením hlavy). Uvedená bodková notácia je v porovnaní so zoznamovou neprehľadnejšia. V ďalšom uvedieme príklady troch zoznamov (zápisy v jednom riadku sú rovnocenné):

| | | | |
|---|---|---|---|
| [a] | .(a,[]) | a.[] | a.[] |
| [a,b,c] | .(a,.(b,.(c,[]))) | a.(b.(c.[])) | a.b.c.[] |
| ??? | .(a,.(b,.(c,d ))) | a.(b.(c.d )) | a.b.c.d |

Posledný zoznam nekončí prázdnym zoznamom a zatiaľ ho nevieme zapísať v zoznamovej notácii. Aj zoznamová notácia však umožňuje oddeliť Hlavu zoznamu od Tela pomocou zápisu [Hlava|Telo] .

Príklad 2.4.

Rozložte rôzne zoznamy na hlavu a na telo.

Riešenie:

| ZOZNAM | HLAVA | TELO |
|---|---|---|
| [a] | a | [] |
| [a,b,c,d] | a | [b,c,d] |

| | | |
|---|---|---|
| [[a,b],c,d] | [a,b] | [c,d] |
| [a,[b,c]] | a | [[b,c]] |
| [a\|b] | a | b |
| [a,b,[c\|d]] | a | [b,[c\|d]] |
| [] | nedá sa rozložiť (neúspech) | |

Zoznam v predposlednom riadku odpovedá bodkovej notácii a.b.c.d. (Posledná bodka nie je súčasť zápisu, označuje koniec vety).

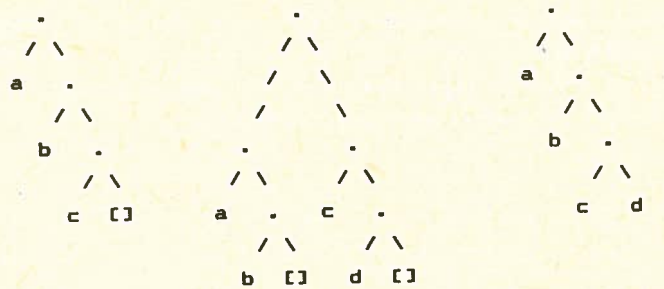
- *** -

Zložitejšie zoznamy je účelné názorne zobraziť binárnym stromom.

Príklad 2.5.

Zobrazte zoznamy [a,b,c], [[a,b],c,d] a [a,b,[c|d]] binárnym stromom.

Riešenie:




[a,b,c] [[a,b],c,d] [a,b,[c|d]]

obr. 2.2

## 2.4 REKURZÍVNE PRAVIDLÁ PRE PRÁCU SO ZOZNAMAMI

Väčšina prologovských programov využíva rekurzívne údajové štruktúry. Pre prácu s nimi je potrebné navrhnúť niektoré rekurzívne definované predikáty.

**Príklad 2.6.**

Navrhnite definíciu predikátu member , ktorého prvý argument je prvkom zoznamu, tvoriaceho druhý argument.

Riešenie:

X patrí do zoznamu, keď

a./ je jeho prvým prvkom --> member(X,[X|_]).

b./ je prvkom tela zonamu --> member(X,[_|T]):- member (X,T).

Prvá klauzula reprezentuje jednu hraničnú situáciu, druhou je prípad, keď zoznam v druhom argumente je prázdny a nedá sa rozložiť na hlavu a na telo (neúspech). Druhá klauzula je vlastné rekurzívne pravidlo, konvergujúce pri postupnom skracovaní testovaného zoznamu k jednej z uvedených hraničných situácií. Pre označenie údajov, na ktorých "nezáleží", boli použité anonymné premenné (podčiarniky). Predikát member možno použiť rôznymi spôsobmi:

```
?- member(c,[a,b,c,d]).     /* je c prvkom zoznamu [a,b,c,d] ? */
yes

?- member(q,[a,b,c]).       /* je q prvkom zoznamu  [a,b,c] ? */
no

?- member(X,[a,b,c]).       /* ktoré sú prvky zoznamu [a,b,c] ? */
X = a ;<NL>
X = b ;<NL>
X = c ;<NL>
no                          /* ďalšie prvky neexistujú */

?- member(a,X).             /* aké zoznamy obsahujú prvok "a" ? */
X = [a|_22] ;<NL>             /* zoznam s "a" ako prvým prvkom  */
X = [_21,a|_32] ;<NL>         /* zoznam s "a" ako druhým prvkom */
X = [_21,_31,a|_42] <NL>      /* zoznam s "a" ako tretím prvkom */
yes
```

Čísla s podčiarnikom sú vnútorné mená premenných. _21, _31 predstavujú bližšie nedefinované prvky zoznamu a _22, _32, _42 reprezentujú bližšie neurčené zoznamy. V príklade 5.3 bude podrobnejšie pojednané o tom, ako PROLOG k týmto odpovediam dospel.

Príklad 2.7.

Navrhnite predikát pre vytvorenie zoznamu L spojením zoznamov L1 a L2 tak, aby hlavou výsledného zoznamu bola hlava zoznamu L1 .

Riešenie:

Práve požiadavka spoločnej hlavy zoznamov L1 a L ponúka riešenie úlohy: Po oddelení hlavy od obidvoch zoznamov L1 a L musia byť opäť hlavy obidvoch zoznamov zhodné (tj. pôvodne v poradí druhé prvky zoznamov L1 a L ). To znamená rekurzívne volanie predikátu concat pre postupne stále kratšie zoznamy L1 a L . Tým je súčasne zaručená konvergencia k hraničnej situácii, ktorá nastane, keď sa zoznam L1 skráti až na prázdny zoznam. Celý proces možno schématicky znázorniť, viď obr. 2.3.

```
  L1      L2        H    L1    L2                H   L2
<-----|<-----|     |-|<----|<-----|              |-|<-----|
                                        . . .
<------------|     |-|<-----------|              |-|<-----|
     L             H        L                    H    L
```

obr. 2.3

Naznačenú rekurziu realizuje pravidlo:

```
concat([H|L1], L2 , [H|L]) :- concat(L1, L2, L).
```

a ošetrenie hraničnej situácie ďalšie pravidlo:

```
concat([], L2, L ) :- L2 = L.
```

Posledné pravidlo môže byť nahradené faktom a musí byť umiestnené pred rekurzívnym pravidlom, takže výsledné riešenie má tvar:

```
concat([], L, L).
concat([H|L1], L2, [H|L]) :- concat(L1, L2, L).
```

V ďalšom uvedieme niektoré z možných spôsobov použitia navrhnutého predikátu. Keď sú v odpovedi systému neviazané premenné, budú označované vhodným menom (napr. H1, H2, T, A, B, ... );

obor platnosti tohoto mena je spracovanie jednej otázky (i keď sa pritom generuje viacej alternatív). V dialógu s konkrétnym prologovským systémom je označovanie neviazaných premenných v odpovediach implementačne závislé (zvyčajne podčiarnik nasledovaný prirodzeným číslom, ktoré jednoznačne identifikuje príslušnú premennú):

```
?- concat([a,b,c], [d,e], [f,g]).
no

?- concat([a,b,c],[d,e],X).
X = [a,b,c,d,e] ;<NL>
no

?- concat(X,Y,[a,b,c]).
X = []          Y = [a,b,c] ;<NL>
X = [a],        Y = [b,c] ;<NL>
X = [a,b]       Y = [c] ;<NL>
X = [a,b,c],    Y = [] ;<NL>
no

?- concat(X,[a,b],Y).
X = [],           Y = [a,b] ;<NL>
X = [H1],         Y = [H1,a,b] ;<NL>
X = [H1,H2],      Y = [H1,H2,a,b] ;<NL>
X = [H1,H2,H3],   Y = [H1,H2,H3,a,b] <NL>
yes

?- concat([a,b],X,Y).
X = T, Y = [a,b|T] ;<NL>
no

?- concat(X,Y,Z).
X = [],           Y = T, Z=T ;<NL>
X = [H1],         Y = T, Z=[H1|T] ;<NL>
X = [H1,H2],      Y = T, Z=[H1,H2|T] ;<NL>
X = [H1,H2,H3]    Y = T, Z=[H1,H2,H3|T] <NL>
yes
```

```
?- concat([a,b],X,[a,b,c]).
X = [c] ;<NL>
no

?- concat(X,[c,d],[a,b,c]).
no
```

Príklad 2.8.

Použite predikát concat k definovaniu ďalších "rodinných" predikátov pre vyhľadanie:

a./ zoznamu mladších a starších súrodencov ZM, ZS dieťaťa D
b./ najstaršieho súrodenca SS rodiča R
c./ najbližšieho staršieho súrodenca NS dieťaťa D

za predpokladu, že deti sa po narodení zaradia vždy na začiatok zoznamu detí (viď kapitola 1.8).

Riešenie:

```
a./  surodenci(ZM,ZS,D) :- rodina(_,_,ZozDeti),
                           concat(ZM,[D|ZS],ZozDeti).

b./  najstarsi(SS,R)    :- (rodina(R,_,ZD) ; rodina(_,R,ZD)),
                           R \= slob, concat(_,[SS],ZD).

c./  starsi(NS,D)       :- rodina(_,_,ZozDeti),
                           concat(_,[D,NS|_],ZozDeti).
```

Príklad 2.9.

Nakoľko zoznamy sú v PROLOGu jednosmerné, často je potrebné získať k zoznamu L zoznam R s prvkami v obrátenom poradí. Navrhnite predikát, realizujúci naznačenú činnosť.

Riešenie:

Pri riešení možno postupovať nasledovne: oddelíme hlavu H zoznamu L , obrátime poradie prvkov tela T tohoto zoznamu (rekurzia !) a za takto získaný zoznam X pripojíme prvok H , čo je možné znázorniť graficky (obr.2.4).

```
     L            H      T                 H
<---------|      |-|<-------|             |-|
|--------->      |-||------->    . . .    |-|
     R            H      X                 H

                  |---------->             |->
                       R                   R
```

obr. 2.4

Zoznamy I , X sa postupne skracujú až na prázdne zoznamy, rekurzia teda smeruje k hraničnému stavu, reprezentovanému faktom, že obrátením prázdneho zoznamu sa získa prázdny zoznam. Tento fakt spolu s rekurzívnym pravidlom pre vyššie uvedený postup dáva riešenie úlohy :

```
rev([],[]).
rev([H|T],R) :- rev(T,X), concat(X,[H],R).
```

Dialóg so systémom pre rôzne spôsoby použitia navrhnutého predikátu je nasledovný :

```
?- rev([a,b,c],[c,b]).
no

?- rev([a,b,c],X).
X = [c,b,a] ;<NL>
no

?- rev(Y,[a,b,c]).
Y = [c,b,a] ;<NL>
- nekonečná slučka -

?- rev(X,Y).
X = [],        Y = [] ;<NL>
X = [A],       Y = [A] ;<NL>
X = [A,B],     Y = [B,A] ;<NL>
X = [A,B,C],   Y = [C,B,A] <NL>
yes
```

Keď chceme, aby sa navrhnutý predikát pri navracaní nezacyklil v tom prípade, keď je prvý argument neviazaná premenná, potom je potrebné definovať tento predikát pomocou troch klauzúl :

```
rrev([],[]).
rrev([H|T],R) :- nonvar(R), concat(X,[H],R),  rrev(T,X).
rrev([H|T],R) :- var(R),    rrev(T,X),        concat(X,[H],R).
```

V predošlom dialogu sa zmení iba to, že namiesto nekonečného cyklu systém správne odpovie no .

Je ovšem možné definovať aj efektívnejší predikát pre obrátenie poradia prvkov v zozname:

```
obrat(L,R) :- rev0(L,[],R).

rev0([],J,J).
rev0([H|A],B,R) :- rev0(A,[H|B],R).
```

Predikát rev0 v každom rekurzívnom kroku odoberá hlavu zoznamu z prvého argumentu a pripája ho na začiatok budovaného zoznamu v druhom argumente. Proces začína prázdnym zoznamom v druhom argumente a končí, keď sa stáva prázdnym zoznam v prvom argumente. Vtedy sa výsledok prenáša do tretieho argumentu a jeho prostredníctvom aj do nadradeného predikátu obrat :

| 1.arg | 2.arg | 3.arg |
|---|---|---|
| [a,b,c, ... ,x,y,z] | [] | R |
| [b,c, ... ,x,y,z] | [a] | R |
| [c, ... ,x,y,z] | [b,a] | R |
| [y,z] | [x, ... ,c,b,a] | R |
| [z] | [y,x, ... ,c,b,a] | R |
| [] | [z,y,x, ... ,c,b,a] | [z,y,x, ... ,c,b,a] |

# 3. ČINNOSŤ SYSTÉMU PROLOG

Doteraz sme opisovali činnosť systému PROLOG pri plnení zadaného cieľa iba intuitívne. Tento prístup síce vyhovuje pre jednoduché príklady, ale pre náročnejšie programy je potrebné dôkladnejšie pochopiť jednotlivé kroky, ktoré systém pri plnení cieľov vykonáva. Nasledujúce podkapitoly pojednávajú stručne a bez nároku na korektnosť z hľadiska matematickej logiky o týchto krokoch.

## 3.1. UNIFIKÁCIA A VIAZANIE PREMENNÝCH

Výber tých informácií z databázy, ktoré systém PROLOG potrebuje pri plnení zadaného cieľa, sa deje vyhľadaním prvej klauzuly, ktorá je s týmto cieľom unifikovateľná . Keď je touto klauzulou fakt, cieľ je okamžite splnený. Keď je ňou pravidlo, potom podmienkou splnenia cieľa je splnenie subcieľov v tele tohoto pravidla.

Na začiatku sú všetky premenné neviazané , t.j. nezastupujú žiadny konkrétny objekt. K viazaniu môže dôjsť práve v procese unifikácie.

Pravidlá pre unifikáciu sú nasledovné:

1. dve konštanty sú unifikovateľné iba keď sú zhodné

2. neviazaná premenná je unifikovateľná s každým objektom, pričom dochádza k viazaniu premennej na tento objekt (ktorý nemusí byť plne určený - môže obsahovať iné neviazané premenné); súčasne sú viazané na tento objekt všetky výskyty premennej v rozsahu jej platnosti

3. dve neviazané premenné sa po unifikácii stávajú združenými , čo znamená, že keď sa niektorá z nich v ďalšom naviaže na nejaký konkrétny objekt, aj druhá bude súčasne viazaná na ten istý objekt

4. dve štruktúry sú unifikovateľné, keď majú zhodný funktor a rovnaký počet argumentov, pričom všetky odpovedajúce si dvojice argumentov musia byť unifikovateľné

5. cieľ (resp. subcieľ) je unifikovateľný s faktom v databáze, keď sú reprezentované unifikovateľnými štruktúrami (atóm sa chápe ako štruktúra bez argumentov)

6. cieľ (resp. subcieľ) je unifikovateľný s pravidlom v databáze, keď cieľ a hlava pravidla sú tvorené unifikovateľnými štruktúrami.

Príklad 3.1.

Určite, či sú nasledujúce dvojice objektov unifikovateľné, alebo nie. Keď dochádza k viazaniu premenných, uveďte príslušné "priradenie":

| | | |
|---|---|---|
| a./ | alfa | alfa |
| b./ | 45 | 37 |
| c./ | dievca(jana) | dievca(X) |
| d./ | lubi(jano,Y) | lubi(X,pivo) |
| e./ | sum(2+3) | sum(5) |
| f./ | sum(2+3) | sum(X+Y) |
| g./ | sum(2+3) | sum(X) |
| h./ | lubi(jano,pivo(plzenske,12)) | lubi(jano,X) |
| i./ | lubi(jano,pivo(plzenske,X)) | lubi(Y,pivo(Z,12)) |
| j./ | a(alfa,b(X)) | a(U,b(U)) |
| k./ | lubi(jano,X) | lubi(Z,W) |

Riešenie:

a./ áno b./ nie c./ áno X = jana
d./ áno X = jano, Y = pivo e./ nie
f./ áno X = 2, Y = 3 g./ áno X = 2+3
h./ áno X = pivo(plzenske,12)
i./ áno X = 12, Y = jano, Z = plzenske
j./ áno X = alfa, U = alfa
k./ áno Z = jano, X a W sa stali združenými premennými

- *** -

Pre unifikáciu zoznamov platí v plnej miere bod 4., t.j. dva zoznamy sú unifikovateľné, keď sú unifikovateľné ich hlavy aj telá.

Príklad 3.2.

Určite, či sú uvedené dvojice zoznamov unifikovateľné a keď áno, uveďte aj príslušné viazania premenných:

| | | |
|---|---|---|
| a./ | [X,Y,Z] | [eva,jana,pavla] |
| b./ | [X\|[Y\|Z]] | [eva,jana,pavla] |
| c./ | [X\|W] | [eva,jana,pavla] |
| d./ | [X\|W] | [eva] |
| e./ | [X\|eva] | [eva,jana] |
| f./ | [X\|W] | [] |
| g./ | [X\|Z,W] | [eva,jana,pavla] |
| h./ | [X,Z\|W] | [eva,jana,pavla] |
| i./ | [X\|[Y\|Z]] | [eva\|[jana\|pavla]] |

Riešenie:

| | | |
|---|---|---|
| a/. | áno | X = eva, Y = jana, Z = pavla |
| b/. | áno | X = eva, Y = jana, Z = [pavla] |
| c/. | áno | X = eva, W = [jana,pavla] |
| d./ | áno | X = eva, W = [] |
| e./ | nie | f./ nie g./ chybná syntax prvého zoznamu |
| h./ | áno | X = eva, Y = jana, W = [pavla] |
| i/. | áno | X = eva, Y = jana, Z = pavla |

Prázdny zoznam nemožno rozdeliť na hlavu a telo (prípad f./). V prípade i./ máme zvláštny prípad, keď zoznam nekončí prázdnym zoznamom (viď analogické prípady v príkladoch 2.4 a 2.5).

Príklad 3.3.

Predpokladajte, že v databáze sú uložené klauzuly

```
dievca(jana).        dievca(mara).
lubi(jana,pivo).     lubi(jano,D) :- dievca(D), lubi(D,pivo).
```

Popíšte postup unifikácie a odpoveď systému pri splnení cieľov:

a./ ?- dievca(mara).
b./ ?- lubi(jana,X).
c./ ?- lubi(jano,X).

Riešenie:

a./ Cieľ je unifikovateľný s druhým faktom definície predikátu dievca a systém odpovie yes .

b./ Cieľ je unifikovateľný s prvým faktom definície predikátu lubi , pri unifikácii dochádza k viazaniu premennej X na atóm pivo a systém odpovie výpisom tohoto viazania a yes .

c./ Cieľ je unifikovateľný s druhou klauzulou definície predikátu lubi , pri unifikacii sa premenné X a D stávajú združenými. Podmienkou splnenia hlavného cieľa je splnenie oboch subcieľov z tela vybraného pravidla. Subcieľ dievca(X) je unifikovateľný s faktom dievca(jana) , pritom sa premenná X viaže na atóm jana (na všetkých troch miestach jej výskytu v klauzule !!). Nakoľko premenné X a D sú združené, súčasne dôjde aj k viazaniu D na jana . Druhý subcieľ lubi(jana,pivo) je unifikovatelný s faktom v databáze. Hlavný cieľ je splnený, systém vypíše viazanie premennej D a informáciu yes . Keď sa pri pokuse o splnenie cieľa hľadá unifikovateľná klauzula, databáza PROLOGu sa prehľadáva od začiatku.

## 3.2. NAVRACANIE

Doteraz uvedené poznatky o chovaní systému PROLOG môžme stručne zhrnúť v dvoch bodoch:

- pri prehľadávaní databázy sa postupuje zhora nadol (k nájdenej unifikovateľnej klauzule položí systém značku)

- v prípade konjunkcie cieľov, resp. subcieľov sa postupuje zľava doprava ; splnenie (sub)cieľa sa "hlási" susedovi sprava

a keď už žiadny taký nie je, tak nadradenému cieľu; v prípade, že nadradený cieľ je používateľom zadaný hlavný cieľ, potom systém oznámi viazanie premenných v hlavnom cieli a odpovie yes (keď sa v tele pravidla vyskytne dvakrát ten istý predikát, v oboch prípadoch sa databáza prehľadáva od začiatku, viď príklad 5.2).

V prípade neúspešného pokusu o splnenie niektorého subcieľa (nenašla sa unifikovateľná klauzula v databáze) sa systém "vracia naspäť k ľavému susedovi (už predtým úspešne splnenému) a pokúša sa ho opätovne splniť. Predtým sa však zrušia všetky viazania premenných, ktoré prebehli pri predošlej úspešnej unifikácii a hľadanie ďalšej unifikovateľnej klauzuly začína tentokrát o značky (t.j. za klauzulou, ktorá bola naposledy unifikovaná s subcieľom, ktorý sa pokúšame opätovne splniť). Proces "hlásenia" neúspechu ľavému susedovi sa označuje ako navracanie. Navracanie spolu s nasledujúcim pokusom o opätovné splnenie ľavého suseda predstavuje hľadanie novej alternatívy riešenia.

Keď je pokus o opätovné splnenie neúspešný, pokračuje navracaní ďalej doľava. V prípade neúspechu prvého subcieľa konjunkcii (ten už nemá ľavého suseda) sa neúspech "hlási" nadradenému cieľu. Keď je ním používateľom zadaný hlavný cieľ, potom systém odpovie no.

V prípade úspešného pokusu o opätovné splnenie sa pokráčuje znova pravým susedom (subcieľom), ktorý sa pokúšame splniť a nie opätovne splniť t.j. databáza sa prehľadáva od začiatku. Mohlo by sa zdať, že tento pokus je už nezmyselný, veď podobný pokus v predošlej etape už skončil neúspechom. Musíme si však uvedomiť, že tentokrát sa pokus robí "inou cestou", nakoľko dielčie riešenie ľavých susedov predstavuje inú alternatívu, než tomu bolo pri neúspešnom pokuse: prejaví sa to tým, že ľaví susedia odovzdávajú k novému pokusu spravidla odlišne viazané premenné, než pri predošlom pokuse.

Navracanie môže vyvolať aj sám používateľ zadaním bodkočiarky po kladnej odpovedi systému. Inicializuje tým hľadanie inej alternatívy riešenia.

Príklad 3.4.

Majme v databáze klauzuly

```
dievca(jana).          dievca(mara).
lubi(mara,pivo).       lubi(jano,D) :- dievca(D), lubi(D,pivo).
lubi(jano,vino).
```

Popíšte chovanie PROLOGu pri otázke

```
?- lubi(jano,X).
```

Riešenie:

Systém nájde pravidlo ako prvú klauzulu unifikovateľnú s hlavným cieľom (položí sem značku L1 ) a pokúsi sa postupne splniť obidva subciele. Subcieľ dievca(D) splní unifikáciou s faktom dievca(jana) (položí sem značku D1 ), ale neuspeje pri pokuse o splnenie druhého subcieľa lubi(jana,pivo) . Vyvolá sa navracanie a pokus o opätovné splnenie prvého subcieľa začína od značky D1 . Nová alternatíva je dievca(mara) , pre ktorú je druhý subcieľ splnený a tým je splnený aj nadradený (a súčasne hlavný) cieľ lubi(jano,mara) a systém odpovie X = mara . Zadaním bodkočiarky vyžiada používateľ ďalšie riešenie: systém začína prehľadávať databázu od značky L1 a nájde ďalšiu unifikovateľnú klauzulu lubi(jano,vino) a položí sem značku L2 . Tým sa úspešne skončil pokus o opätovné splnenie hlavného cieľa a systém odpovie X = vino . Ďalšou bodkočiarkou možno vyvolať znovue navracanie, ale nakoľko inej unifikovateľnej klauzuly za značkou L2 už nieto, systém odpovie no (t.j. neexistuje žiadne ďalšie riešenie).

## 3.3. DISJUNKCIA SUBCIEĽOV

Doteraz sme predpokladali, že používateľom zadaný hlavný cieľ ako aj telá pravidiel, ktoré sa pri pokuse o jeho splnenie použijú sú tvorené konjunkciou subcieľov (výroky so spojkou a ). Neraz sa však vyskytnú aj výroky so spojkou alebo , teda disjunkcie subcieľov. Pokiaľ sa jedná o alternatívne podmienky platnosti výroku, reprezentovaného hlavou pravidla, je

možné vyhnúť sa explicitnému zápisu disjunkcie pomocou bodkočiarky tak, že každý dielčí výrok zapíšeme pomocou samostatného pravidla. Tak napríklad predikát rodic z príkladu 1.10 môžeme definovať aj nasledovne:

```
rodic(Rodic,Dieta) :-  otec(Rodic,Dieta).
rodic(Rodic,Dieta) :- matka(Rodic,Dieta).
```

V záujme čitateľnosti programu je vhodné sa vyhýbať častému použitiu disjunkcií v zápise pravidiel všade tam, kde je to možné.

V prípade kombinovaných výrokov je potrebné zohľadniť skutočnosť, že operátor konjunkcie silnejšie viaže operandy než operátor disjunkcie a v prípade potreby použiť zátvorky (viď príklady 1.10 a 1.12).

Príklad 3.5.

Navrhnite definíciu predikátov svokor a rodic (viď príklady 1.11 a 1.12).

Riešenie:

```
svokor(Svokor,ZatNevesta) :-
  otec(Svokor,Partner),
  (manzelia(Partner,ZatNevesta) ; manzelia(ZatNevesta,Partner)).

rodic(Rodic,Dieta) :-
  (otec(Rodic,Dieta) ; matka(Rodic,Dieta)), Rodic \= slob.
```

Bez použitia disjunkcie je možné prepísať tieto definície takto

```
svokor(Svokor,ZatNevesta) :-
            otec(Svokor,Partner), manzelia(Partner,ZatNevesta).
svokor(Svokor,ZatNevesta) :-
            otec(Svokor,Partner), manzelia(ZatNevesta,Partner).

rodic(Rodic,Dieta) :- otec(Rodic,Dieta),  Rodic \= slob.
rodic(Rodic,Dieta) :- matka(Rodic,Dieta), Rodic \= slob.
```

Uvedené dva spôsoby zápisu vystihujú plne požadované relácie, nie sú však úplne ekvivalentné z hľadiska činnosti PROLOGu: v prípade nesplnenia subcieľa manzelia(Partner,ZatNevesta) sa systém pokúsi v prvom prípade splniť druhý subciel disjunkcie, kdežto v druhom prípade sa vyvolá navracanie k subcieľu otec .

- *** -

V niektorých prípadoch nie je možný "rozklad" prvého typu "zmiešanej" definície (disjunkcia je za "spoločným" subcieľom) na dve pravidlá naznačeným spôsobom. Jedná sa obvykle o vetvenie programu a cykly (viď ďalší príklad a príklad 4.8), často v kombinácii s predikátmi pre vstup údajov.

Príklad 3.6.

Navrhnite predikát, ktorý prečíta term a keď je to atóm, alebo číslo, potom vypíše príslušnú informáciu. V opačnom prípade bude nesplnený.

Riešenie:

```
test :- read(X),
        (integer(X), write(cislo) ; atom(X), write(atom)).
```

Rozpis na dve klauzuly:

```
test :- read(X), integer(X), write(cislo).
test :- read(X), atom(X),    write(atom).
```

nebude fungovať korektne pri načítaní iného termu než čísla (systém v druhom pravidle načíta ďalší term). Je však možné vyhnúť sa použitiu operátora disjunkcie zavedením pomocného predikátu:

```
test :- read(X), pom(X).

pom(X) :- integer(X), write(cislo).
pom(X) :- atom(X),    write(atom).
```

## 3.4. NEGÁCIA SUBCIEĽOV

Prologovská negácia nie je negáciou v pravom slova zmysle, tak ako ju používame v matematickej logike: subcieľ not(G) je splnený, keď cieľ G neuspeje. Pri programovaní sa táto skutočnosť prejaví najmarkantnejšie v tom, že negáciu nemožno použiť bezprostredne na generovanie alternatív.

Príklad 3.7.

Majme v databáze uložené fakty

```
lubi(peter,jana).     lubi(fero,jana).     lubi(jozo,jana).
lubi(peter,maria).    lubi(fero,maria).    lubi(jozo,maria).
lubi(peter,pavla).                         lubi(jozo,pavla).
lubi(peter,pivo).     lubi(fero,pivo).

dievca(jana).         dievca(pavla).       dievca(maria).
chlapec(peter).       chlapec(fero).       chlapec(jozo).
```

Formulujte prologovské ciele, vyjadrujúce nasledovné otázky:

a./ Koho nemá rád Fero ?

b./ Kto nemá rád pivo ?

c./ Kto nemá rád žiadne dievča ?

d./ Kto má rád všetky dievčatá ?

Riešenie:

a./ Začiatočník by pravdepodobne navrhol príslušný cieľ v tvare

```
?- not lubi(fero,Koho).
no
```

Systém našiel unifikovateľnú klauzulu lubi(fero,jana) a tým je hlavný cieľ nesplnený. Správne riešenie musí obsahovať najprv nenegovaný "generujúci" subcieľ a až za ním môže byť test, obsahujúci negáciu:

```
?- dievca(D), not lubi(fero,D).
D = pavla ;
no
```

b./ Analogicky s prípadom a./ nemožno formulovať hlavný cieľ púhou negáciou predikátu lubi(Kto,pivo) , ale takto

```
?- (chlapec(Kto) ; dievca(Kto)), not lubi(Kto,pivo).
```

Otázku sme koncipovali trochu všeobecnejšie aj pre prípad, keby sa v databáze nachádzali nejaké údaje o "pivárkach"

c./ Aj v tu zlyháva intuitívna "začiatočnícka" formulácia:

```
?- chlapec(CH), dievca(D), not lubi(CH,D).
CH = fero, D = pavla ;
no
```

Tej totiž prislúcha otázka "ktorý chlapec nemá rád aspoň jedno dievča ?". Správne zadaný cieľ pre pôvodnú otázku je:

```
?- chlapec(CH), not(( dievca(D), lubi(CH,D) )).
no
```

Dvojité zátvorky sú potrebné kvôli tomu, aby systém správne interpretoval čiarku medzi subcieľmi dievca a lubi ako operátor konjunkcie a nie ako oddeľovač parametrov neexistujúceho binárneho predikátu not .

d./ Otázkam tohoto typu prislúcha v matematickej logike kvantifikátor všeobecnosti, ktorý v PROLOGu nemožno priamo vyjadriť (prologovské zápisy vyjadrujú výroky s existenčným kvantifikátorom). Preformulujme preto pôvodnú otázku na ekvivalentný, ale existenčne kvantifikovaný tvar: "pre ktorého chlapca neexistuje také dievča, ktoré by tento chlapec nemal rád". Prepis do PROLOGu je potom nasledovný

```
?- chlapec(CH), not(( dievca(D), not lubi(CH,D) )).
CH = peter ;
CH = jozo ;
no
```

Premenné v negovanom subcieli (v našom príklade D ) ostávajú aj po splnení subcieľa not neviazané.

## 3.5. RIADENIE NAVRACANIA

Niekedy je potrebné zmeniť štandardný mechanizmus navracania (dôvody uvedieme pozdejšie). Hlavným prostriedkom riadenia navracania je symbol rezu , označovaný výkričníkom. Je to cieľ, ktorý je pri pokuse o splnenie okamžite splnený, ale pokus o jeho opätovné splnenie je vždy neúspešný. Naviac znemožní pokusy o opätovné splnenie subcieľov, ležiacice naľavo od neho. Spôsobí tým okamžité nesplnenie nadradeného cieľa, t.j. nedovolí ani hľadať v databáze nejakú ďalšiu klauzulu pre tento cieľ. Všetky uvedené činnosti sa realizujú odstránením príslušných znaciek z databázy (viď kapitolu 3.2).

Spôsob práce systému pri vyhodnocovaní symbolu rezu možno názorne demonštrovať na jednoduchom príklade:

```
hc :- x, y, nc, z, w.

nc :- a, b, c, !, d, e, f.
nc :- g, h.
```

Pri vyhodnocovaní subcieľov a, b, c pracuje systém tak, ako bolo uvedené v kapitolách 3.1 a 3.2. Po splnení subcieľa c je cieľ ! okamžite splnený a systém "normálne" vyhodnocuje subciele d, e, f . Keď je ale subcieľ d neúspešný, potom pokus o opätovné splnenie subcieľa ! je neúspešný a nie je možný ani pokus o opätovné splnenie subcieľa c ani o unifikáciu nadradeného cieľa nc s ďalším pravidlom a systém sa pokúša opätovne splniť subcieľ y . Druhé pravidlo pre nc ovšem nie je zbytočné, pri neúspechu konjunkcie a, b, c (t.j pred dosiahnutím symbolu rezu) sa systém pokúša použiť aj toto pravidlo.

"Zákaz" hľadania ďalších alternatív sa vzťahuje aj na použitie symbolu rezu v zápisoch, využívajúcich operátor disjunkcie. Napríklad aj pri použití pravidla

```
nc :- a, b, c, !, d, e, f ; g, h.
```

by sa systém choval vyššie uvedeným spôsobom.

Ďalšími prostriedkami pre riadenie navracania sú štandardné predikáty

repeat - je vždy splnený a je nekonečne krát opätovne splniteľný (používa sa na programovanie cyklov, viď kapitola 4)

true - je vždy splnený ale nie je opätovne splniteľný (príklad 3.12 ilustruje jeho použitie)

fail - je vždy nesplnený (používa sa na realizáciu cyklov a v kombinácii so symbolom rezu na realizáciu negácie pomocou neúspechu, viď príklady 3.8 a 3.9)

Použitia symbolu rezu možno rozdeliť do dvoch skupín:

- zelený rez , ktorý nemá vplyv na deklaratívnu sémantiku programu, teda po jeho odstránení sa získajú rovnaké výsledky

- červený rez , ktorý ovplyvňuje deklaratívnu sémantiku programu, teda program dáva odlišné výsledky

Zatiaľ čo zelený rez sa používa výhradne pre zvýšenie efektivity programu, červený rez má okrem tohoto použitia ešte aj ďalšiu, veľmi dôležitú funkciu: umožňuje použiť v PROLOGu predikát not , teda tzv. negáciu pomocou neúspechu (viď príklady 3.8 - 3.9).

Možno vysloviť niekoľko všeobecných zásad o použití rezu:

použitie rezu je zbytočné

- keď sa v tele pravidla pred ním nenachádza žiaden opätovne splniteľný subcieľ

- pred subcieľom repeat , ktorý je nekonečne krát opätovne splniteľný a tak nie je možné cez neho prejsť sprava doľava pri navracaní

<u>použitie rezu je možné (zelený rez)</u> všade tam, kde ďalšie riešenie už neexistuje (resp. existuje, ale nás už nezaujíma) a PROLOG by sa napriek tomu pokúšal unifikovať ďalšie klauzuly, prípadne splniť ďalšie ciele; pritom je potrebné rozlíšiť dva prípady:

- podľa argumentov v hlave pravidla je možné unifikáciou rozdeliť klauzuly na disjunktné skupiny (väčšinou je v takejto skupine jediná klauzula); rez položíme na začiatok tela posledného pravidla každej skupiny; dobrým príkladom je definícia predikátov <u>klas2</u> a <u>klas3</u> v príklade 3.10 (táto optimalizácia je zbytočná v tých verziách PROLOGu, ktoré majú indexáciu klauzúl, napr. DEC-10 PROLOG, QUINTUS PROLOG)

- výber danej klauzuly bol okrem unifikácie v hlave pravidla potvrdený aj splnením "testovacieho" subcieľa v tele klauzuly ; rez zaradíme <u>za</u> tento test (túto optimalizáciu už indexácia klauzúl nevie urobiť); príkladom sú niektoré formy zápisu vetvenia IF-THEN-ELSE, viď kapitolu 4.2

<u>použitie rezu je nevyhnutné (červený rez)</u> v týchto prípadoch:

- keď použijeme znalosť, vyplývajúcu z neúspechu predošlej klauzuly pre vynechanie testu v tele niektorej nasledujúcej klauzuly; rez umiestnime <u>za</u> testom

- pre zastavenie cyklu <u>generuj&testuj</u> po nájdení riešenia (keď vieme, že existuje viacero riešení a nás ďalšie riešenia už nezaujímajú); rez bude <u>za</u> testom

- v definícii negácie pomocou neúspechu (príklad 3.8) a vo všetkých predikátoch, ktoré používajú negáciu pomocou neúspechu a pritom nevolajú štandardný predikát <u>not</u> (príklad 3.9)

- pre zastavenie cyklu s <u>repeat</u> ; rez umiestnime za podmienku ukončenia cyklu (každé <u>repeat</u> musí mať svoj rez, inak sa program dostane do nekonečnej slučky)

- na konci akumulačného cyklu, aby sme zabránili generovaniu chybných výsledkov (jedná sa o medzivýsledky jednotlivých krokov cyklu, viď niektoré príklady v kapitole 4.6 a obr. 5.11 a 5.12)

- v spojení s disjunkciou (o tomto prípade pojednáme bližšie na konci podkapitoly)

Príklad 3.8.

Definujte predikát not tak, aby realizoval negáciu pomocou neúspechu, t.j. funkciu, opisovanú v kapitole 3.4.

Riešenie:

```
not Goal :- Goal, !, fail.
not Goal.
```

Príklad 3.9.

Definujte predikát, testujúci nepríslušnosť prvku k zoznamu, t.j. negáciu predikátu member (príklad 2.6) ovšem bez použitia štandardného predikátu not .

Riešenie:

```
not_member(H,[H|_]) :- !, fail.
not_member(H,[_|T]) :- not_member(H,T).
not_member(_, []  ).
```

Príklad 3.10.

Navrhnite predikát pre klasifikáciu termu do niekoľkých disjunktných skupín (predikát má teda vždy jediné riešenie).

Riešenie:

```
klasi(X,premenna) :- var(X).
klasi(X,  nil   ) :- X == [].
klasi(X, atom   ) :- atom(X), X \= [].
klasi(X,iny_term) :- nonvar(X), not atom(X).
```

V druhej klauzule nebolo možné použiť operátor unifikácie, nakoľko neviazaná premenná X by bola unifikovateľná s atómom [] a získali by sme druhé, nesprávne riešenie. Použitím zelených rezov možno tento program zefektívniť:

```
klas2(X,premenna) :- var(X), !.
klas2(X,  nil   ) :- X == [], !.
klas2(X, atom   ) :- atom(X), X \= [], !.
klas2(X,iny_term) :- nonvar(X), not atom(X).
```

Analýzou programu však zistíme, že v prvej klauzule sme ošetrili prípad, keď X je neviazaná premenná a v ďalších klauzulách už môžme využiť túto informáciu. Zelený rez sa však zmení na červený. Na základe rovnakej úvahy môžme vynechať i v telách ďalších pravidiel negácie testov, vystupujúcich v predchádzajúcich klauzulách. Tým sa všetky rezy zmenia na červené a program bude mať tvar:

```
klas3(X,premenna) :- var(X), !.
klas3([], nil   ) :- !.
klas3(X, atom   ) :- atom(X), !.
klas3(X,iny_term).
```

- *** -

Často potrebujeme posunúť vplyv rezu o jednu úroveň, t.j. u nejakého (vo všeobecnosti nedeterministického) subcieľa použiť iba jedno riešenie a neohroziť pritom výber ďalších klauzúl pre predikát, ktorý tento subcieľ volá. K tomuto účelu je výhodné definovať predikát

```
rez(X) :- X, !.
```

ktorý možno volať napr. s argumentom <u>retract</u> a pod. Inokedy naopak potrebujeme zamedziť vplyv rezu v argumente <u>call</u> na plnenie cieľa s <u>call</u> - plnenie X musíme zase odsunúť o jednu úroveň:

```
ries(X) :- X.
```

pričom sa jedná o prípady, keď X je konjunkcia subcieľov, ktorá môže obsahovať aj symbol rezu.

V súvislosti s použitím rezu pri riadení navracania je dobré mať na pamäti niektoré skutočnosti:

- zo štandardných predikátov iba <u>clause</u>, <u>retract</u>, <u>deny</u>, <u>repeat</u>, <u>call</u> a <u>subgoal of</u> sú opätovne splniteľné

- keď je nejaký predikát definovaný jediným pravidlom a v jeho tele sa vyskytuje opätovne splniteľný subcieľ, potom tento predikát môže byť tiež znovusplniteľný (má preto zmysel použiť rez aj pri definícii predikátu s jedinou klauzulou, ale až niekde za opätovne splniteľným subcieľom)

- podobne má zmysel umiestniť symbol rezu v tele posledného pravidla definície nejakého predikátu iba za opätovne splniteľný subcieľ (ak sa tam taký vôbec vyskytuje)

- aj keď je rez zelený pre pôvodne zamýšľaný charakter argumentov (vstupné/výstupné), pre inú kombináciu argumentov sa stane obvykle červeným

- keď sa v tele pravidla vyskytuje viacero opätovne splniteľných subcieľov, má zmysel použiť rez za každým z nich

- je síce možné programovať bez rezu, ale musí sa používať not, v ktorom jeden rez ostáva (viď príklad 3.9)

- nemožno zapísať niekoľko vnorených cyklov s repeat , každé nové repeat zastaví predošlý cyklus tejto klauzuly

- unifikácia nie je opätovne splniteľná

Na záver uvedieme niekoľko typických obratov s použitím disjunkcie a predikátov fail a true:

- cyklus riadený podmienkou (viď kapitola 4.4)

- vetvenie IF-THEN-ELSE uprostred tela pravidla (viď kap. 4.2)

- vytvorenie reverzibilnej operácie z operácie, ktorá je ireverzibilná:

```
p1(X) :- akcia(X).
p1(X) :- inverzna_akcia(X), !, fail.
```

predikáty assertb a retractb z kapitoly 9.2 ilustrujú tento prípad ; keď inverzna_akcia nie je opätovne splniteľná, potom je rez v druhom pravidle zbytočný

Priklad 3.11.

Navrhnite definíciu predikátu geti , ktorý funguje podobne ako štandardný predikát get , ale pri navracaní uloží prečítanú informáciu do databázy, odkiaľ ju pri novom vyvolaní geti "prečíta" znova.

Riešenie:

Je potrebné spojiť obe pravidlá pre geti , aby sa hodnota X získaná pomocou subcieľa daj preniesla do argumentu subcieľa assertz :

```
geti(X) :- daj(X), (true ; assertz(unget(X)), fail).

daj(X) :- retract(unget(X)), !.
daj(X) :- get(X).
```

- *** -

V prípade, že je potrebné zachovať nejaké informácie, ktoré by sa mohli zničiť pri plnení ďalších subcieľov, potom použijeme nasledovnú definíciu:

```
p2(X) :- kopiruj(X,Y), akcia(X),
         (true; inverzna_akcia(Y), fail).

kopiruj(X,Y) :- asserta(kopia(X)) ,retract(kopia(Y)).
```

Vo všeobecnosti "akcia" zničí X a preto je účelné použiť predikát kopiruj - týka sa vstupno/výstupných operácií a manipulácií s databázou.

Priklad 3.12.

Navrhnite predikát zvýšujúci stav počítadla v glob. premennej.

Riešenie:

```
pridaj(X) :- kopiruj(X,Y), set_gvar(suma, suma + X),
             (true ; set_gvar(suma,suma-Y), fail).
```

# 4. PROGRAMOVANIE RIADIACICH ŠTRUKTÚR

## 4.1. RIADIACE ŠTRUKTÚRY

Zástanci rýdzo logického programovania zdôrazňujú neprocedurálnu podstatu PROLOGu a skutočnosť, že v tomto jazyku nie sú potrebné príkazy typu GO TO, IF THEN ELSE, WHILE, FOR, atď. Programátori - praktici však využívajú s výhodou aj procedurálny prístup k programovaniu v PROLOGu.

Typickým príkladom je programovanie vetvení a cyklov, pre ktoré PROLOG nemá explicitne definované prostriedky. Prevažnú väčšinu úloh je však možné formulovať pomocou jazyka logiky, čo umožní napísať program bez zjavného použitia uvedených riadiacich štruktúr. Pre mnohé úlohy je ovšem tento prístup násilný a najmä programátori s "procedurálnou minulosťou" ho pociťujú ako veľmi neprirodzený.

Preto v ďalšom pojednáme o rôznych spôsoboch programovania vetvení a cyklov v PROLOGu. Je však potrebné si uvedomiť, že obvykle dospejeme k rovnakému výslednému programu, nezávisle na tom, či bol zostavený "logickým", alebo "procedurálnym" prístupom. Rozdiel je iba v myšlienkových postupoch pri zostavovaní programu, prípadne v slovnej interpretácii jednotlivých pravidiel.

## 4.2. ROZHODOVANIE, ANALÓGIA IF THEN ELSE ŠTRUKTÚR

Najprirodzenejším spôsobom možno naprogramovať vetvenie pomocou dvoch pravidiel, pričom v tele prvého je test IF a "činnosti" vetvy THEN a v tele druhého pravidla sú "činnosti" vetvy ELSE.

Príklad 4.1.

Navrhnite predikát, ktorý otestuje, či X leží medzi hodnotami D a H .

**Riešenie:**

```
test(X,D,H) :- X>D, X<H, write(v_intervale).
test(X,D,H) :-           write(mimo_interval).
```

Bolo by možné zlúčiť obe pravidlá do jediného:

```
test(X,D,H) :- X>D, X<H, write(v_intervale) ;
                         write(mimo_interval).
```

ale kvôli lepšej prehľadnosti programu sa odporúča vyhýbať sa disjunkcii, pokiaľ to je možné.

Obidve uvedené riešenia majú jeden nedostatok: keď X leží v predpísanom intervale a z nejakého dôvodu sa vyvolá navracanie, potom predikát "test" po prvej správnej alternatíve generuje aj druhú, chybnú. Uvedený nedostatok možno odstrániť doplnením negácie testu pred "činnosti" druhej vetvy:

```
test(X,D,H) :- X>D, X<H,          write(v_intervale).
test(X,D,H) :- not((X>D, X<H)), write(mimo_interval).
```

Dvojité zátvorky bolo potrebné použiť, aby systém nechápal prvý subcieľ ako predikát not s dvomi argumentami, ale s jedným, ktorý reprezentuje konjunkciu. Negácii je možné sa vyhnúť použitím komplementárneho testu:

```
test(X,D,H) :- X>D, X<H,       write(v_intervale).
test(X,D,H) :- (X=<D; X>=H), write(mimo_interval).
```

kde =< a >= sú štandardné predikáty pre "menší alebo rovný" a "väčší alebo rovný".

Iný spôsob, ako sa vyhnúť chybnej druhej alternatíve pri navracaní, je použitie symbolu rezu:

```
test(X,D,H) :- X>D, X<H, !,  write(v_intervale)  ;
                             write(mimo_interval).
```

- *** -

Použitie symbolu rezu, podobne ako použitie disjunkcie zneprehľadňuje programy. V záujme čitateľnosti sa odporúča používať symbol rezu iba v odôvodnených prípadoch.

Príklad 4.2.

Navrhnite predikát pre určenie cifier C a D súčtu dvoch jednociferných čísel A a B .

Riešenie:

Použitie disjunktívneho pravidla so symbolom rezu

```
sucet(A,B,C,D) :- X is A+B, (X<10,!,C=0,D=X; C=1,D is X-10).
```

sa javí efektívnejšie, nakoľko pri použití dvoch pravidiel by sa subcieľ X is A+B opakoval dvakrát. Štandardný predikát is viaže prvý argument na hodnotu aritmetického výrazu v druhom argumente. Keď chceme viazať premennú iba na hodnotu čísla, možno pre tento účel použiť aj ≡ aj is .

Je však možná aj definícia pomocou dvoch pravidiel, v ktorých sa vyhneme práve týmto typom viazaní premenných C a D (táto úprava zaistí výrazné zrýchlenie výpočtu oproti pôvodnému variantu):

```
sucet(A,B,0,X) :- X is A+B, X<10.
sucet(A,B,1,D) :- X is A+B, X>9, D is X-10.
```

Existuje aj tretia definícia predikátu súčet, ktorá ešte viac urýchli výpočet:

```
sucet(A,B,C,D) :- S is A+B, C is S/10, D is S mod 10.
```

V prípade, že argumenty A a B načítavame pomocou štandardného predikátu read , musíme v každom prípade použiť jediné pravidlo (raz načítané číslo sa ešte raz načítať nedá):

```
sucet(C,D) :- read(A), read(B), X is A+B,
              (X<10, !, C=0, D=X ; C=1, D is X-10).
```

Je ovšem možné využiť už definovaný predikát sucet so štyrmi argumentami (majme na pamäti, že predikáty so zhodným menom, ale s odlišným počtom argumentov sú chápané ako rozdielne):

```
sucet(C,D) :- read(A), read(B), sucet(A,B,C,D).
```

Použitie symbolu rezu v hlavnom pravidle znemožňuje pri navracaní opätovné splnenie tohoto typu predikátov (t.j. nemožno ich použiť ako generátor alternatívnych riešení).

Príklad 4.3.

Upravte definíciu predikátu sucet tak, aby umožnil generovať pri navracaní všetky možné súčty jednociferných čísel.

Riešenie:

Predpokladajme, že cifry sú uložené v databáze vo forme faktov (bolo by možné tieto cifry generovať elegantnejšie, rekurzívne - ovšem výpočet by bol menej efektívny):

```
cifra(0).  cifra(1).  cifra(2).  . . .  cifra(9).
```

Aby sme umožnili generovanie alternatív, použijeme pri definícii dve pravidlá bez symbolu rezu:

```
sucet(A,B,0,X) :- cifra(A), cifra(B), X is A+B, X<10.
sucet(A,B,1,D) :- cifra(A), cifra(B), X is A+B, X>9, D is X-10.
```

Je možné aj iné riešenie s využitím pomocného predikátu:

```
sucet(A,B,C,D) :- cifra(A), cifra(B), X is A+B, pom(X,C,D).

pom(X,C,D) :- X<10, !, C=0, D=X ; C=1, D is X-10.
```

Symbol rezu v pomocnom predikáte nezabráni generovaniu alternatív v nadradenom pravidle.

Príklad 4.4

Navrhnite operátory if , then , else tak, aby umožnili zápis vetvenia v tvare obvyklom v procedurálnych jazykoch.

Riešenie:

Deklarujme operátory nasledujúcim spôsobom:

```
op(108,fx,if), op(106,xfx,then), op(104,xfx,else)
```

a definujme ich pravidlom

```
if Podmienka then Cinnosti else Cinnost2 :-
                    Podmienka, !, Cinnosti ; Cinnost2.
```

Príkladom použitia takejto definície je predikát, testujúci, či načítaný term je celé číslo, alebo iný objekt jazyka PROLOG:

```
test :- read(X), if integer(X) then write(cele_cislo)
                                else write(iny_term).
```

alebo predikát pre výpočet absolútnej hodnoty rozdielu dvoch čísel:

```
abs(A,B,V) :- if A<B then V is B-A else V is A-B.
```

<u>Príklad 4.5.</u>

Upravte a doplňte predikáty z predošlého príkladu tak, aby umožnili zápis štruktúr typu IF THEN ELSIF THEN ELSIF THEN . . . ELSE.

Riešenie:

Deklarácie operátorov doplníme o <u>elsif</u> a zmeníme typ operátora <u>then</u> z <u>xfx</u> na <u>xfy</u> (na rozdiel od predošlého príkladu už budeme reťaziť tento operátor):

```
?- op(108,fx, if  ), op(106,xfy,then),
   op(104,xfx,else), op(104,xfx,elsif).
```

Pre ujasnenie navrhnutých deklarácií je výhodné zobraziť niektoré typické štruktúry binárnym stromom (obr. 4.1).

```
        if                                  if
        |                                   |
        |                                  then
       then                                 /\
        /\                                 / then
       /  \                               /   /\
      /   else                           /   /  \
     /     /\                           / elsif else
    /     /  \                         /   /\    /\
   p1    c1   c                       p1  c1 p2 c2 c
```

if p1 then c1 else c  if p1 then c1 elsif p2 then c2 else c

if(then(p1,else(c1,c)))

if(then(p1,then(elsif(c1,p2),else(c2,c))))

obr. 4.1

Príslušné predikáty je potom možné definovať pravidlami

```
if P1 then C1 else  C          :- P1, !, C1 ; C.
if P1 then C1 elsif P2 then X :- P1, !, C1 ; if P2 then X.
```

Prvé pravidlo ošetruje hraničný prípad (viď ľavý binárny strom) a druhé je rekurzívne.

Použitie novonavrhnutých predikátov demonštrujme na príklade podrobnejšieho testovania načítaného termu:

```
test1 :- read(X), if     integer(X) then write(cele_cislo)
                 elsif  atom(X)     then write(atom)
                                    else write(struktura).
```

<u>Príklad 4.6.</u>

Navrhnite predikát <u>case</u> realizujúci prepínač, analogický CASE príkazu v jazykoch pascalovského typu.

Riešenie:

Doplníme deklarácie operátorov z príkladu 4.5 o niektoré ďalšie:

```
op(108,fx,case), op(108,xfx,of), op(106,xfy,@), op(104,xfx,:)
```

a príslušné predikáty definujeme rekurzívne, pričom pri ošetrení hraničného prípadu využijeme predikáty z príkladu 4.5:

```
case X of X:C @  _   :- !, C.
case X of  _  @  Y   :- case X of Y.
case X of X:C else _ :- !, C.
case X of  _  else C :-    C.
```

Použite nových predikátov ilustrujme znovu na príklade testovania načítavaných termov:

```
test2 :- read(X), case X of
          a:(write(pismeno),tab(1),write(X)) @
          b:(write(pismeno),tab(1),write(X)) @
          7:(write(cislica),tab(1),write(X)) @
       g(Y):(write(struktura_s_argumentom),tab(1),write(Y))
       else  write(ine).
```

pričom "skupinu prikazov" je potrebné uzavrieť do zátvoriek, aby bola skutočne chápaná ako konjunkcia.

## 4.3. PROGRAMOVANIE CYKLOV

Už základnú formu dialógu používateľa s PROLOGom možno chápať ako cyklus. Počet opakovaní zadáva používateľ počtom bodkočiarok, ktorými vyvoláva hľadanie ďalších alternatív. Keď ho zaujímajú všetky alternatívy, môže nahradiť ťažkopádne zadávanie bodkočiarok použitím štandardného predikátu <u>fail</u>, ktorý sa preto niekedy označuje ako "programová bodkočiarka". V tomto prípade sa ovšem výstup hľadaných informácií musí explicitne naprogramovať, nakoľko <u>fail</u> na konci tela pravidla vyvolá stále neúspech a na systémovom výstupe sa okrem koncového "no" žiadna informácia neobjaví.

<u>Príklad 4.7.</u>

V databáze nech sú uložené fakty:

a(5). a(33). a(21). a(end). a(77). a(33).

Navrhnite predikát, ktorý by vypísal na terminál postupne argumenty týchto faktov.

Riešenie:

Zohľadňujúc skutočnosť, že štandardné predikáty pre výstup nie sú opätovne splniteľné, navrhneme predikát

```
vypis :- a(X), nl, write(X), fail.
```

Keď je potrebné vykonať nejaké činnosti pred vlastným cyklom (prípadne po ňom), je výhodné použiť pre všetky tieto funkcie jediný predikát (prechod na ďalšie pravidlá zaistí <u>fail</u> na konci predchádzajúcich pravidiel, subciele pred <u>fail</u> v prvom pravidle ovšem nesmú byť opätovne splniteľné):

```
vypis1 :- write('hladane argumenty su :'), fail.
vypis1 :- a(X), nl, write(X), fail.
vypis1 :- nl, write('to je vsetko').
```

Cyklus sa nemusí vzťahovať na celú klauzulu - prvé dve pravidlá by sme mohli nahradiť jediným:

```
vypis1 :- write('hladane argumenty su :'),
          a(X), nl, write(X), fail.
```

## <u>4.4. CYKLUS RIADENÝ PODMIENKOU</u>

Jedná sa o cyklus, ktorého ukončenie sa zaistí vhodným testom, teda vlastne vetvením programu.

Príklad 4.8.

Upravte predikát <u>vypis</u> z príkladu 4.7 tak, aby vypisoval iba argumenty faktov nachádzajúcich sa pred faktom <u>a(end)</u> .

Riešenie:

Použijeme postup analogický metódam z kapitoly 4.2:

```
vypis_zac :- a(X), (X=end, ! ; nl, write(X), fail).
```

Symbol rezu zabráni výpisu zvyšku databázy v prípade navracania.

- *** -

Keď sa alternatívy spracovávané v cykle získavajú prostredníctvom predikátov vstupu (nie sú opätovne splniteľné), potom je v kombinácii s fail (resp. s nesplnením niektorého subcieľa napravo od subcieľa pre vstup) potrebné použiť ešte štandardný predikát repeat (je nekonečnekrát opätovne splniteľný).

Príklad 4.9.

Navrhnite predikát pre vstup čísla, menšieho ako 1000.

Riešenie:

V tele predikátu umiestnime test na predpísanú hodnotu. Nesplnenie testu vyvolá opakovaný výpis nápovedného textu a načítanie nového údaja. Cyklus sa opakuje až do zadania správneho vstupného údaja:

```
vstup(X) :- repeat, write('Zadaj cislo mensie ako 1000 : ')
                read(X), X<1000, !.
```

Príklad 4.10.

Navrhnite predikáty, ktoré by umožnili zapísať cyklus riadený podmienkou v tvare obvyklom u procedurálnych jazykov.

Riešenie:

V PROLOGu sú časté prípady, keď sa vykoná nejaká činnosť a potom sa testuje, či bola pritom splnená podmienka ukončenia cyklu (typ REPEAT). Cykly s testom pred vykonaním činnosti (typ WHILE) sú menej časté. Nakoľko repeat je rezervované pre štandardný predikát, zvolíme iné kľúčové slovo:

```
?- op(100,fx,reiterate), op(96,xfx,until).
```

a definujeme príslušné predikáty nasledovne:

```
reiterate Cinnost until Podmienka :-
                repeat, Cinnost, (Podmienka, ! ; fail).
```

Potom napríklad pomocou nasledujúceho pravidla

```
vstup1(X) :-
  reiterate ( write('Zadaj cislo mensie ako 1000 : '), read(X) )
  until     X<1000.
```

možno realizovať vstup čísla aj s testom jeho hodnoty. Podobne zadaním cieľa

```
?- reiterate (pretekar(SC,Meno), write(Meno)) until SC=6.
```

môžeme vypísať mená všetkých pretekárov z databázy podľa príkladu 4.12. od začiatku až po pretekára so štartovným číslom 6 (vrátane). V tomto prípade sa zaisťuje opakovanie cyklu pomocou subcieľa pretekar na rozdiel od predikátu vstup1 , kde sa opakovanie zaisťuje predikátom repeat . Keď v databáze neexistuje pretekár s číslom na ktoré sa testuje, potom sa program dostane do nekonečnej slučky.

## 4.5.CYKLUS RIADENÝ PREMENNOU

Nakoľko predikát is nemožno opätovne splniť (a tobôž nie so zapamätaním medzivýsledku) je potrebné navrhnúť nejaký iný mechanizmus inkrementácie parametra cyklu.

Jednou z možností je použiť rekurziu, pričom parameter cyklu je potrebné prenášať cez jeden argument rekurzívne volaného predikátu. Nakoľko sa pri rekurzii "kópie" predikátu v každom kroku ukladajú do pamäti počítača, tento spôsob realizácie cyklu je pomerne neefektívny (najmä pri veľkom počte opakovaní).

Príklad 4.11.

Navrhnite predikát pre výpis N hviezdičiek a použite ho pre nakreslenie histogramu, pričom zobrazované číselné hodnoty sú uložené v databáze formou faktov

```
a(3).  a(9).  a(12).  a(5).  a(4).  a(7).
```

Riešenie:

```
star(0) :- !.
star(N) :- write(*), M is N-1, star(M).
```

Prvá klauzula uvedenej definície slúži iba k splneniu predikátu <u>star</u> , čo je ovšem dôležité pri jeho použití ako subcieľa v tele pravidla, cyklicky definujúceho nejaký iný predikát. Je tomu tak napríklad aj pri definíci. predikátu pre kreslenie histogramu:

```
histogram :- a(X), star(X), nl, fail.
histogram.
```

<u>Príklad 4.12.</u>

V databáze nech sú uložené fakty:

```
pretekar(3,kovac).   pretekar(5,juhas).   pretekar(2,rumcajs).
pretekar(6,malina).  pretekar(1,cecer).   pretekar(4,plavka).
```

Navrhnite predikát pre výpis mien pretekárov, ktorých štartovné čísla (prvý argument) sú <u>D</u> , <u>D+1</u> , <u>D+2</u> , ... , <u>H-1</u> , H .

Riešenie:

Na rozdiel od predošlého príkladu je teraz okrem hornej hranice premennou aj dolná hranica a musíme aj jej priradiť jeden argument rekurzívneho predikátu:

```
listina(D,H) :- D>H, !.
listina(I,H) :- pretekar(I,Meno), write(Meno), nl, J is I+1,
                listina(J,H).
```

Tento predikát funguje korektne vtedy, keď sú v databáze všetky štartovné čísla z intervalu <u><D,H></u> . Keď chceme zabezpečiť jeho správnu činosť aj v prípade, že niektoré z čísel bude chýbať, je potrebné upraviť druhú klauzulu v jeho definícii:

```
listina(I,H) :- (pretekar(I,Meno), write(Meno), nl ; true),
                J is I+1, listina(J,H).
```

- *** -

Použitie rekurzie je síce elegantný, ale pomerne neefektívny spôsob programovania cyklov riadených premennou. Iná možnosť je použiť špeciálny fakt v databáze pre uchovávanie parametra cyklu (počítadlo). V PROLOGu sú k dispozícii štandardné predikáty pre pridávanie klauzúl na začiatok databázy - asserta , resp. na jej koniec - assertz , ako aj predikát retract , ktorý odstraňuje klauzulu z databázy (k informáciám z odstraňovanej klauzuly je možný prístup pomocou viazania premenných).

Príklad 4.13.

Navrhnite nerekurzívnu definíciu predikátu pre výpis N hviezdičiek.

Riešenie:

Použijeme fakt poc(I) na uchovávanie parametra I cyklu. Zvlášť ošetríme prípad N=0 (prvá klauzula), potom vynulujeme počítadlo (druhá klauzula; subcieľ retractall(poc( )) odstraňuje z databázy všetky fakty s unárnym predikátom poc a je splnený aj v prípade, keď v databáze žiaden taký fakt neexistuje).

Vlastnú tlač potrebného počtu hviezdičiek realizuje tretia klauzula v procese navracania, vyvolávaného tak dlho, kým nie je splnená podmienka Y=N (predikáty is , write a asserta nie sú opätovne splniteľné, zatiaľ čo retract pri svojom opätovnom splnení viaže premennú X na hodnotu argumentu faktu poc , t.j. na okamžitú hodnotu parametra cyklu):

```
star1(0) :- !.
star1(N) :- retractall(poc(_)), assertz(poc(0)), fail.
star1(N) :- integer(N), retract(poc(X)), write(*), Y is X+1,
                        assertz(poc(Y)), Y=N, !.
```

Predikát fail na konci druhej klauzuly slúži k "automatickému prechodu" na tretiu klauzulu. Test integer(N) kontroluje, či argument predikátu star1 je skutočne číslo. Symbol rezu zabráni nekonečnému vypisovaniu hviezdičiek v prípade, keď sa z nejakých dôvodov vyvolá navracanie (test Y=N už nikdy viac nemôže byť splnený). Bolo nutné v rekurzívnom pravidle použiť predikát assertz , nakoľko pri použití asserta by sa fakt s novým stavom počitadla uložil pred smerník, od ktorého systém

začína hľadať pri pokuse o opätovné splnenie subcieľa retract - pri použití asserta by tento pokus bol neúspešný (funkcia dvojice predikátov assertz - retract je ovšem implementačne závislá a je vhodné ju preveriť na používanej verzii PROLOGu).

Niektoré implementácie PROLOGu ponúkajú špeciálny prostriedok, tzv. globálne premenné. Nezapadajú síce do koncepcie logického programovania, ale sú na druhej strane veľmi praktickým programátorským nástrojom. S výhodou možno použiť globálnu premennú ako parameter cyklu:

```
star2(N) :- integer(N), set_gvar(i,0),
            repeat, write(*), set_gvar(i,i+1), N is i, !.
```

Predikát set_gvar patrí medzi štandardné predikáty PROLOG-80. Priraďí globálnej premennej v prvom argumente hodnotu aritmetického výrazu z druhého argumentu. Meno globálnej premennej je atom, v našom príklade i. V teste na hornú hranicu cyklu N nemožno použiť operátor rovnosti ( i=N je pokus o unifikáciu premennej N s atómom i ).

Príklad 4.14.

Definujte vhodné operátory tak, aby umožnili zápis cyklu riadeného premennou v tvare obvyklom v procedurálnych jazykoch.

Riešenie:

Deklarujme operátory

```
?- op(104,fx,for), op(98,xfx,step), op(96,xfx,do),
   op(102,xfx,:=), op(100,xfx,to).
```

a definujme príslušné predikáty nasledovne:

```
for I:=Od to Do step Krok do Cinnost :-
      assertz(i(Od)), retract(i(I)),
      (I>Do, !, ; Cinnost, J is I+Krok, assertz(i(J)), fail).
```

Potom sa pri splnení cieľa

```
?- for I:=0 to 10 step 2 do (write(I),nl).
```

vypíšu čísla 0, 2, 4, 6, 8, 10 vždy na nový riadok.

Takto definované predikáty možno použiť iba vtedy, keď premenná Cinnost predstavuje cieľ, ktorý nemožno opätovne splniť. Keď totiž Cinnost je schopná generovať alternatívy, navracanie nedôjde až k subcieľu retract , ktorý má poskytnúť novú hodnotu parametra cyklu. Aby cyklus fungoval správne aj pre generujúcu Cinnost , je potrebné upraviť vyšie uvedenú definíciu na tvar:

```
for I:=Od to Do step Krok do Cinnost :-
                    set_gvar(I,Od), repeat, Cinnost,
                    (I>=Do, ! ; set_gvar(I,I+Krok), fail).
```

Po takto zmenenej definícii je možné výpis čísel od 0 do 10 s krokom 2 realizovať pomocou cieľa:

```
?- for i:=0 to 10 step 2 do (J is i, write(J), nl).
```

a výpis mien prvých troch pretekárov z databázy (viď príklad 4.12.) pomocou cieľa:

```
?- for i:=1 to 3 step 1 do
        (pretekar(Cislo,Meno), write(Meno), nl).
```

Takto realizované cykly nemožno ovšem vnárať do seba.

Použitie aritmetických operácií pri práci s počítadlom sa považuje niekedy za "neprologovské". Za predpokladu, že sa nepracuje s veľkým počtom opakovaní, je možné naznačený problém obísť definovaním predikátu nasledovnik pomocou faktov:

```
nasledovnik(0,1).  nasledovnik(1,2).  nasledovnik(2,3).  ...
```

a cykly organizovať nasledovne:

```
star1(0) :- !.
star1(X) :- nasledovnik(Y,X), write(*), star1(Y).
```

V porovnaní s cyklom, používajúcim globálnu premennú sa výpočet výrazne zrýchli. Je možné použiť aj "peanovskú" definíciu nasledovníka, podľa ktorej sú jednotlivé prvky reprezentované štruktúrami:

```
0   s(0)   s(s(0))   s(s(s(0)))   s(s(s(s(0))))   ...
0   1      2         3            4               ...
```

Výpočet sa ešte viac urýchli v porovnaní s "aritmetickým" počítadlom (pomocou is ) a naviac sa definícia "cyklických" predikátov ešte viac zjednoduší:

```
star2(0)    :- !.
star2(s(X)) :- write(*), star2(X).
```

Ovšem "volanie" takto definovaného predikátu je trochu nepohodlné. Napríklad výpis troch hviezdičiek sa musí vyžiadať pomocou subcieľa:

```
star2(s(s(s(0))))
```

naviac je tento spôsob náročnejší na pamäť (vnorená štruktúra namiesto celého čísla).

## 4.6. AKUMULAČNÉ CYKLY

Niekedy je potrebné odovzdávať medzivýsledky z jedného kroku cyklu do druhého (častý je prípad akumulácie medzivýsledkov). Pri realizácii tohoto typu cyklu v PROLOGu je potrebné vyhradiť pre tento účel ďalší argument predikátu (v cykloch s rekurziou), alebo fakt v databáze (v cykloch s navracaním).

**Príklad 4.15.**

Navrhnite predikát, ktorý opakovane vypisuje nápoveď > a číta číslo z klávesnice, pričom tieto čísla priebežne spočítava. Cyklus sa ukončí prečítaním nečíselného údaja.

Riešenie:

```
sum(V) :- write(>), read(X), integer(X), sum(W), V is W+X, !.
sum(0).
```

Rekurzívne volanie sum realizuje najprv postupné čítanie jednotlivých čísel. Po vstupe nečíselného údaja je prvá klauzula nesplnená, systém prejde na druhú, ktorá vynuluje počitadlo (argument predikátu). Súčasne sa splnil aj subcieľ sum na predošlej úrovni rekurzie a je možné pristúpiť k splneniu subcieľa is (premenná W je viazaná na hodnotu 0 a X na posledné načítané číslo). Pri postupnom vynáraní sa na vyššie úrovne rekurzie sa splnením subcieľa is realizuje pripočítavanie ďalších a ďalších čísel (v poradí opačnom než boli načítané). Pri splnení tohoto subcieľa na najvyššej úrovni sa premenná V viaže na hodnotu výsledného súčtu. Symbol rezu zabráni generovaniu ďalších alternatív pri navracaní (alternatívy by predstavovali medzisúčty v jednotlivých krokoch cyklu).

Je možné vytvárať súčet aj v priebehu vnárania sa do rekurzie v jednom argumente predikátu a výsledok na konci preniesť do druhého argumentu:

```
sum(DielciaSuma,Vysledok) :- write(>), read(X), integer(X),
                             NovaSuma is DielciaSuma + X,
                             sum(NovaSuma,Vysledok), !.
sum(Vysledok,Vysledok).
```

Pri volaní tohoto predikátu musí byť v prvom argumente nula a v druhom neviazaná premenná (resp. očakávaná hodnota súčtu).

Príklad 4.16.

Nech v databáze sú uložené fakty:

```
a(5).  a(7).  a(1).  a(4).  a(12).  a(23).  a(8).
```

Navrhnite predikát, ktorý určí súčet argumentov týchto faktov.

Riešenie:

Prirodzeným sa javí definovať požadovaný predikát analogicky ako v predošlom príklade:

```
sum1(V) :- a(X), sum1(W), V is W+X, !.
sum1(0).
```

Subcieľ a(X) ovšem pri svojom splnení viaže premennú X stále na hodnotu argumentu prvého faktu (jedná sa o splnenie, nie o opätovné splnenie tohoto subcieľa, t.j. databáza sa prehľadáva vždy od začiatku). Takto navrhnutý predikát by sa dostal do nekonečnej slučky. Uvedený nedostatok je možné odstrániť úpravou:

```
sum2(V) :- retract(a(X)), sum2(W), V is W+X, !.
sum2(0).
```

Toto riešenie je vhodné, keď fakty s predikátom a predstavujú medziprodukty, ktoré v ďalšom už nie sú potrebné (dokonca je vhodné od nich databázu očistiť). Keď je však potrebné zachovať pôvodný stav databázy, musíme zvoliť iné riešenie:

```
sum3(V) :- retract(a(X)), sum3(W), V is W+X, asserta(a(X)), !.
sum3(0).
```

Subcieľ asserta pri postupnom vynáraní sa na vyššie úrovne rekurzie ukladá fakty a(X) späť do databázy v pôvodnom poradí.

Použitie predikátu is v predošlých príkladoch vyžadovalo istú opatrnosť - bolo potrebné vždy zaistiť, aby v okamžiku volania is boli už všetky premenné aritmetického výrazu v druhom argumente viazané na celočíselné konštanty.

Príklad 4.17.

Navrhnite predikát pre určenie generačného rozdielu medzi členmi rodiny z kapitoly 1.8. Medzi rodičom a dieťaťom je rozdiel jednej generácie, atď.

Riešenie:

V podstate sa jedná o doplnenie rekurzívne definovaného predikátu potomok (viď príklad 2.1) o počítadlo rekurzívnych volaní v treťom argumente (inkrementáciu počítadla sme umiestnili až za subcieľ generacia , ktorý zabezbečí viazanie premennej G1 ):

```
generacia(Potomok,Predok, 1 ) :- rodic(Predok,Potomok).
generacia(Potomok,Predok,Gen) :- rodic(Predok,    X    ),
                                 generacia(Potomok,X,G1),
                                 Gen is G1+1.
```

Dialóg so systémom vyzerá nasledovne:

```
?- generacia(karol,peter,G).
no

?- generacia(peter,karol,G).
G = 2
yes

?- generacia(Potomok,jano,G).
Potomok = peter, G = 1 ;
Potomok = anna,  G = 1 ;
Potomok = viera, G = 2 ;
no
```

- *** -

Dôležitým príkladom akumulačných cyklov je budovanie zoznamov, alebo iných štruktúr. V zásade sa postupuje podobne, ako v doterajších príkladoch. Je ovšem potrebné uvážiť poradie prvkov v budovaných štruktúrach, nakoľko pri niektorých metódach sa zaraďujú prvky v opačnom poradí.

Príklad 4.18.

Predpokladajme, že v databáze sú uložené nasledujúce fakty:

```
p(a).   p(b).   p(c).
```

Navrhnite predikát, ktorý vytvorí zoznam, pozostávajúci z argumentov jednotlivých faktov.

Riešenie:

Predpokladáme, že príslušné fakty môžme v priebehu budovania zoznamu odstrániť z databázy (v prípade, že sa požaduje zachovanie pôvodného stavu databázy, použili by sme analogický postup ako v príklade 4.16):

```
collect([Hlava|Telo]) :- retract(p(Hlava)), collect(Telo), !.
collect([]).
```

Pre kumulovanie je možné použiť aj samostatný argument predikátu a výsledok preniesť do iného argumentu po ukončení kumulácie:

```
collect(Telo,Vysledok) :- retract(p(Hlava)),
                          collect([Hlava|Telo],Vysledok), !.
collect(Vysledok,Vysledok).
```

Ak by sme neboli v uvedených definíciách použili symboly rezu, prípadné navracanie by generovalo ako ďalšie riešenia dielčie zoznamy. Rozdiel v činnosti oboch predikátov ilustruje dialóg:

```
?- collect(Zoznam).
Zoznam = [a,b,c] ;
no
                          /* predpokladáme obnovenie databázy */
?- collect([],Zoznam).
Zoznam = [c,b,a] ;
no
```

Druhý z uvedených spôsobov je vhodné použiť nielen vtedy, keď z nejakého dôvodu potrebujeme opačné poradie prvkov, ale najmä v prípade, keď potrebujeme prvky pripájať pred nejaký iný zoznam:

```
?- collect([f,g],Zoznam).
Zoznam = [c,b,a,f,g] ;
no
```

## 4.7. CYKLICKÉ FORMY DIALÓGU

Obvykle používateľa zaujímajú všetky riešenia danej otázky a je mu na obtiaž vyžadovať tieto riešenia opakovaným zadávaním bodkočiarok. Naviac mu nemusí vyhovovať forma, v ktorej PROLOG vypisuje viazanie premenných pre jednotlivé riešenia. Príklad 4.7 naznačil možnosť ako navrhnúť predikát pre cyklický výpis riešení. Teraz sa pokúsime definovať niekoľko všeobecnejšie koncipovaných predikátov pre tieto účely.

Príklad 4.19.

Navrhnite predikát pre výpis všetkých riešení otázky Query .

Riešenie:

Vlastný cyklus bude realizovať predikát

```
all(Query) :- inic(1), Query, increment(N), nl, write(N),
                write(' > '), write(Query), page(N),
                fail.
all( _ ) :- nl, nl, write('no (more) answers').
```

Predikáty inic a increment slúžia k obsluhe počítadla (globálne premenné PROLOG-80 umožnia jednoduchšiu definíciu all ).

```
inic(Lower) :- retractall(rep(_)), asserta(rep(Lower)),!.

increment(N) :- retract(rep(N)), M is N + 1,
                asserta(rep(M)), !.
```

Predikát page slúži k pozastaveniu výpisu po zadanom počte riadkov (napr.15), aby sa pri väčšom počte riešení používateľ nemusel obávať, že mu niektoré výsledky uniknú pri výpise na obrazovku. Vo výpise sa pokračuje až po stlačení klávesy CR:

```
page(N) :- 0 is N mod 15, tab(20),
           write('CR ... pokracovanie'), get0(_).
```

Keď sú v databáze napríklad fakty:

```
pivo(saris,10).        pivo(bazant,12).    pivo(kozel,12).
pivo(staropramen,10).  pivo(prazdroj,12).
```

potom dialóg so systémom môže byť nasledovný:

```
?- all(pivo(_,_)).

1 > pivo(saris,10)
2 > pivo(bazant,12)
3 > pivo(kozel,12)
4 > pivo(staropramen,10)
5 > pivo(prazdroj,12)

no (more) answers
yes
```

Príklad 4.20.

Navrhnite predikát pre výpis všetkých termov, pre ktoré otázka Query má riešenie.

Riešenie:

Hľadaný predikát je obmenou riešenia predošlého príkladu:

```
all(Query,Term) :- inic(1), Query, increment(N), nl,
                   write(N), write(' > '),
                   write(Term), page(N), fail.
all( _ , _ ) :- nl, nl, write('no (more) answers'), fail.
```

Predikát fail na konci druhého pravidla spôsobí nesplnenie cieľa all. Zabráni sa tým výpisu "výsledkov" - všetkých premenných argumentov z otázky a ich vnútorných mien. Pre databázu z predošlého príkladu môžme viesť nasledujúci dialóg:

```
?- all(pivo(Znacka,_),Znacka).

1 > saris
2 > bazant
3 > kozel
4 > staropramen
5 > prazdroj

no (more) answers
no
```

Keď nás zaujímajú "úspešné" viazania viacerých argumentov z otázky, možno ich vypísať predikátom all tak, že z nich vytvoríme zoznam, napr.:

```
?- all(pivo(Znacka,Stupne),[Znacka,Stupne]).

1 > [saris,10]
2 > [bazant,12]
3 > [kozel,12]
4 > [staropramen,10]
5 > [prazdroj,12]

no (more) answers
no
```

Príklad 4.21.

Modifikujte riešenie predošlej úlohy tak, aby predikát kumuloval do zoznamu všetky termy, pre ktoré daná otázka má riešenie.

Riešenie:

Navrhneme infixné operátory

```
?- op(39,xfy,is_all), op(39,xfy,such_that).
```

pomocou ktorých možno realizovať požadovanú činnosť:

```
List is_all Term such_that Query :- all0(Query,Term),
                                    create(List), !.
```

Predikát all0 ukladá do databázy fakty o vyhovujúcich termoch a predikát create z nich na konci vytvorí zoznam:

```
all0(Query,Term) :- Query, assertz(term(Term)), fail.
all0( _ , _ ).

create([Term|List]) :- retract(term(Term)),
                       create(List).
create( [] ).
```

Pre databázu z príkladu 4.19 sa dostáva nasledujúci dialóg:

```
?- L is_all Brand such_that pivo(Brand,_).
L = [saris,bazant,kozel,staropramen,prazdroj], Brand = _15 ;
no
```

# 5. GRAFICKÁ REPREZENTÁCIA ČINNOSTI SYSTÉMU PROLOG

## 5.1. MODIFIKOVANÉ BLOKOVÉ MODELY

V prípade, že systém odpovedá inak, než očakávame, alebo keď chceme poznať detailne postup, ako systém odvodzuje svoju odpoveď, je vhodné použiť trasovanie. Nakoľko ladiaci výpis je pomerne neprehľadný, je výhodné trasovanú činnosť znázorniť graficky.

Pôvodné blokové modely [1] slúžia iba na sledovanie postupu plnenia cieľov. V ďalšom opíšeme modifikovanú metódu blokových modelov, ktorá umožní zároveň aj sledovanie viazania premenných.

V takomto modeli prislúcha každému cieľu (resp. subcieľu) jeden blok. V ľavom hornom rohu bloku je uvedený cieľ a pred ním v zátvorkách jeho poradové číslo (v zhode s ladiacim výpisom).

```
                    +-------------------+
                    | (N) cieľ(X,Y, ...) |
 pokus o splnenie   |                   | 1.úspech X=x1, Y=y1, ...
------>-------------+---------->--------+------------>----------+
        (N) Call    |                   | (N) Exit               |
                    |                   |                navracanie
                    |     Z1 -->        | 1.pokus o              V
                    |                   | opätovné splnenie      |
                    |              +--<--+-----------<----------+
                    |              |    | (N) Redo
                    |              V    |
                    |              |    | 2.úspech X=x2, Y=y2, ...
                    |              +-->-+----------->----------+
                    |                   | (N) Exit               |
                    |                   |                navracanie
                    |     Z2 -->        | 2.pokus o              V
                    |                   | opätovné splnenie      |
------<-------------+---------------+--<--+-----------<----------+
        (N) Fail    |               "   | (N) Redo
                    |               V   |
                    |               "   | 3.úspech X=x3, Y=y3, ...
                    |            +==>==+==========>==========
                    |     Z3 -->        | (N) Exit
                    +-------------------+
```

obr. 5.1

Pre prácu PROLOGu sú charakteristické štyri činnosti:

Call (GOAL) - - začína sa pokus o splnenie cieľa
Exit (PROVED) - cieľ bol splnený
Redo (RETRY) - pokus o inú alternatívu riešenia
Fail (FAILED) - pokus bol neúspešný

Pri každej činnosti je uvedené aj označenie, používané štandardne pri ladiacom výpise (v zátvorke sú uvedené označenia ktoré vypisuje PROLOG-80, verzie 3.X a RT-11 PROLOG pri použití bodov zastavenia). Uvedené štyri činnosti odpovedajú vstupom a výstupom bloku - bránam - podľa schémy na obr. 5.1.

V prípade úspešného pokusu o splnenie cieľa sú uvedené aj objekty, na ktoré boli naviazané jednotlivé premenné v argumentoch predikátu, reprezentujúceho vyšetrovaný cieľ. Navracanie je vyvolané neúspechom pri pokuse o splnenie niektorého z nasledujúcich subcieľov, alebo v prípade globálneho cieľa priamo používateľom (zadaním bodkočiarky po výpise nájdeného riešenia).

Uvedená forma reprezentácie zachováva smery, ktoré sleduje PROLOG pri svojej činnosti:

- v konjunkcii subcieľov zľava doprava pri pokuse o splnenie a sprava doľava pri navracaní

- zhora nadol pri pokuse o opätovné splenie cieľa, t.j. pri hľadaní ďalšej takej klauzuly v databáze PROLOGu, ktorá je unifikovateľná s vyšetrovaným cieľom (k nájdenej klauzule - od ktorej prípadné ďalšie prehľadávanie začína - možno položiť značku Zi --> )

Na obr. 5.1 je pri druhom pokuse o opätovné splnenie cieľa jednoduchou čiarou vyznačený prípad, keď ďalšie alternatívne riešenie neexistuje a dvojitou čiarou prípad, keď sa ešte našla aj tretia alternatíva (ale navracanie už nebolo vyvolané).

Príklad 5.1

Majme v databáze PROLOGu uložené informácie o tom, kto koho/čo ľúbi (viď príklady 1.4 a 1.8):

```
/* L1 */  lubi(peter,mara).
/* L2 */  lubi(jana,pivo).
/* L3 */  lubi(dana,vino).
/* L4 */  lubi(peter,X) :- dievca(X), lubi(X,pivo).
/* L5 */  lubi(peter,pivo).

/* D1 */  dievca(dana).
/* D2 */  dievca(jana).
/* D3 */  dievca(mara).
```

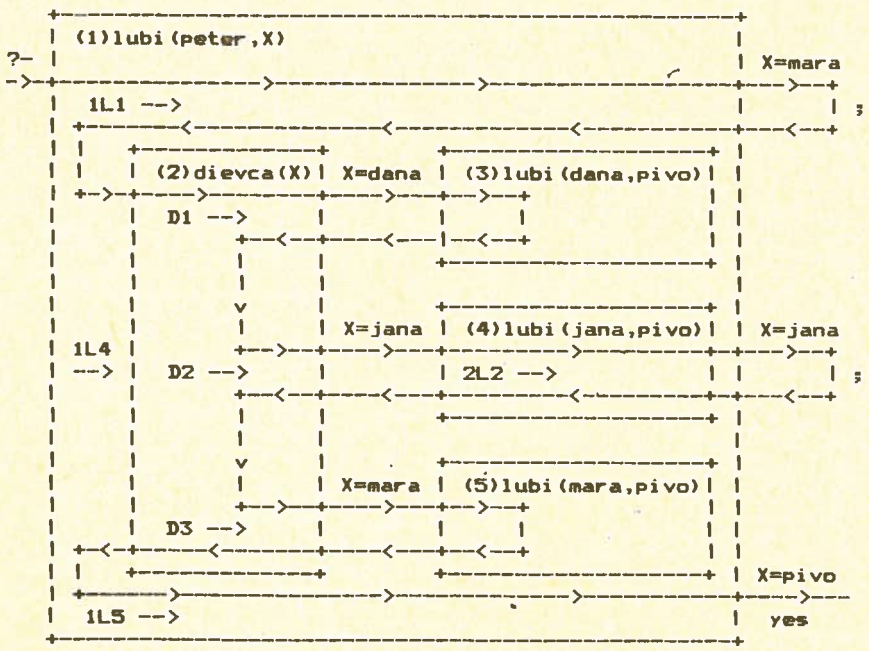
Nájdite odpoveď na otázku "koho/čo ľúbi Peter" a vyznačte podrobne činnosť systému PROLOG pri zodpovedaní tejto otázky.

Riešenie:

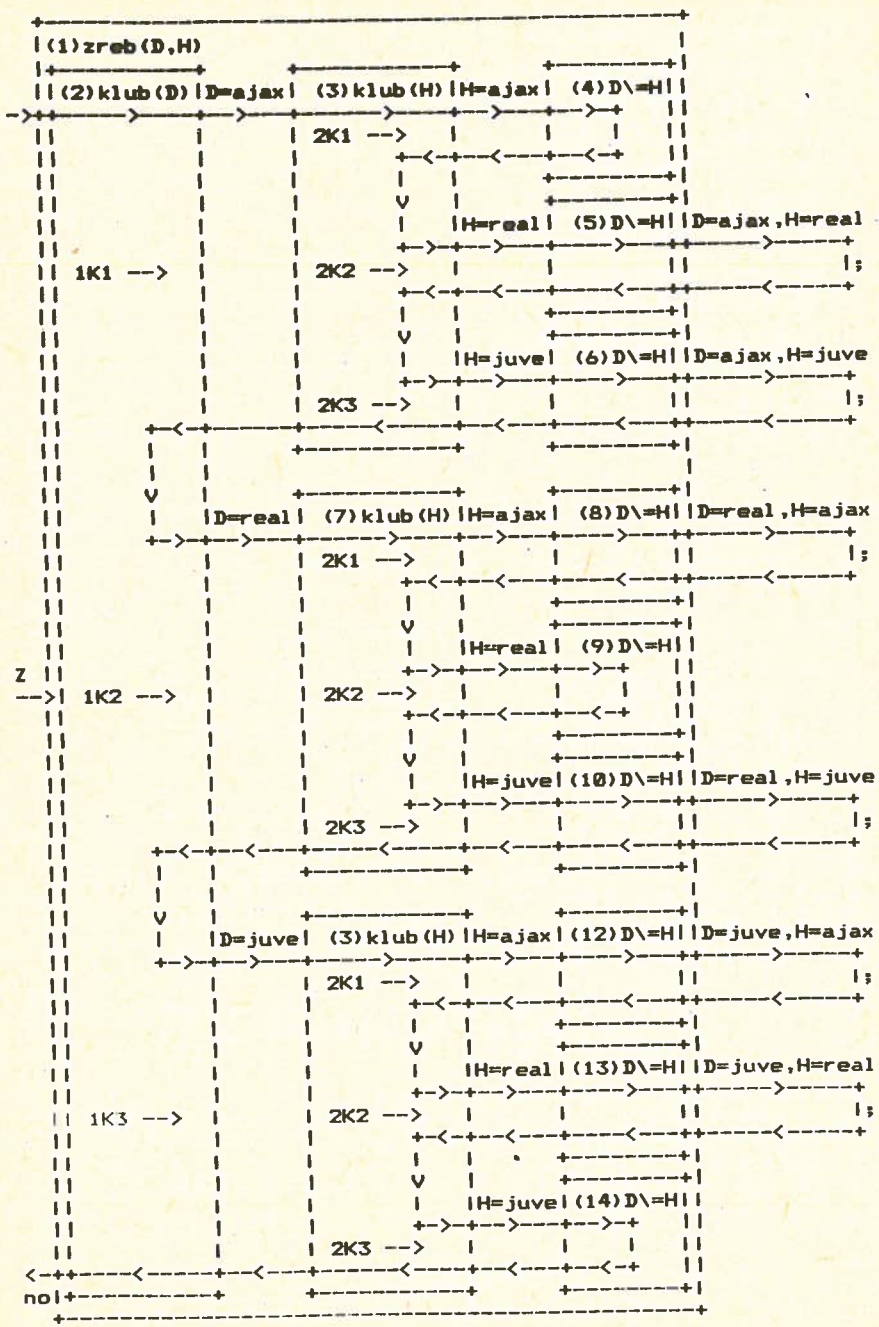
Vyvoláme nasledujúci dialóg (predpokladáme použitie verzie 4.1 PROLOG-80 s neovládanými bránami - pri štarte systému je štandardne navolený režim ovládaných brán, v ktorom systém po každom kroku čaká po výpise nápovede 2 na pokyn používateľa; tento režim možno zmeniť predikátom nonv ):

```
?- trace.
yes

?- lubi(peter,X).
(1) Call: lubi(peter,X)
(1) Exit: lubi(peter,mara)

X = mara ;
(1) Redo: lubi(peter,mara)
 (2) Call: dievca(X)
 (2) Exit: dievca(dana)
 (3) Call: lubi(dana,pivo)
 (3) Fail: lubi(dana,pivo)
 (2) Redo: dievca(dana)
 (2) Exit: dievca(jana)
 (4) Call: lubi(jana,pivo)
 (4) Exit: lubi(jana,pivo)
(1) Exit: lubi(peter,jana)

X = jana ;
```

```
(1) Redo: lubi(peter,jana)
 (4) Redo: lubi(jana,pivo)
 (4) Fail: lubi(jana,pivo)
 (2) Redo: dievca(jana)
 (2) Exit: dievca(mara)
 (5) Call: lubi(mara,pivo)
 (5) Fail: lubi(mara,pivo)
 (2) Redo: dievca(mara)
 (2) Fail: dievca(X)
(1) Exit: lubi(peter,pivo)

X = pivo
yes
```




obr. 5.2


(1)zreb(D,H)
(2)klub(D)
D=ajax
(3)klub(H)
H=ajax
(4)D\=H
2K1 -->
H=real
(5)D\=H
D=ajax,H=real
1K1 -->
2K2 -->
H=juve
(6)D\=H
D=ajax,H=juve
2K3 -->
D=real
(7)klub(H)
H=ajax
(8)D\=H
D=real,H=ajax
2K1 -->
H=real
(9)D\=H
Z
1K2 -->
2K2 -->
H=juve
(10)D\=H
D=real,H=juve
2K3 -->
D=juve
(3)klub(H)
H=ajax
(12)D\=H
D=juve,H=ajax
2K1 -->
H=real
(13)D\=H
D=juve,H=real
1K3 -->
2K2 -->
H=juve
(14)D\=H
2K3 -->
no

obr.5.3

Blokový model uvedeného riešenia je na obr. 5.2. Sú v ňom uvedené aj značky, prislúchajúce jednotlivým klauzulám z databázy. Keď sa vyskytne opakované volanie predikátu, potom číslo pred značkou označuje jej poradové číslo (pri rekurzívnom volaní je to vlastne hĺbka rekurzie). Na prvej úrovni sa v príklade na obr. 5.2 po zadaní bodkočiarky po výpise riešenia X = mara začne prehľadávať databáza od značky 1L1 , ale pri pokuse o splnenie subcieľa lubi v tele pravidla L4 sa prehľadáva databáza od začiatku (druhá úroveň, viď značku 2L2 ).

Príklad 5.2

Vysledujte podrobne činnosť PROLOGu pri zostavení hracieho plánu pohárovej súťaže s tromi účastníkmi:

```
/* K1 */  klub(ajax).
/* K2 */  klub(real).
/* K3 */  klub(juve).
```

pomocou pravidla:

```
/* Z */ zreb(Domaci,Hostia) :- klub(Domaci), klub(Hostia),
                               Domaci \= Hostia.
```

Riešenie.

Priebeh dialógu je patrný z obr.5.3. Tento príklad ešte názornejšie ilustruje skutočnosť, že PROLOG začína prehľadávať databázu od začiatku pri pokuse o splnenie cieľa a od poslednej značky pri pokuse o jeho opätovné splnenie.

Príklad 5.3.

Vyšetrite činnosť PROLOGu pri použití rekurzívne definovaného predikátu member (viď príklad 2.6):

```
/* P1 */   member(X,[X|_]).
/* P2 */   member(X,[_|T]):- member(X,T).
```

Riešenie.

Na obr 5.4 - 5.7 sú uvedené blokové modely činnosti systému pri riešení jednotlivých otázok:

A./ - je c prvkom zoznamu [a,b,c,d] ? - obr. 5.4

- je q prvkom zoznamu [a,b,c] ? - obr. 5.5

B./ - ktoré sú prvky zoznamu [a,b,c] ? - obr. 5.6

C./ - aké zoznamy obsahujú prvok a ? - obr. 5.7

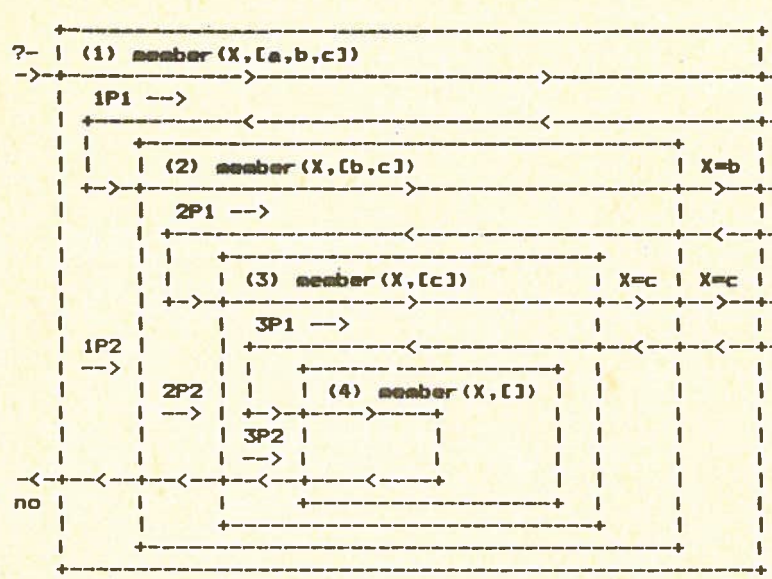
V obr. 5.7 sú uvedené kvôli lepšej čitateľnosti "symbolické" mená premenných, zatiaľ čo ladiaci výpis obsahuje ich vnútorné mená - číslo, pred ktorým je podčiarnik (každej premennej systém jednoznačne priradí takéto číslo).

```
        +-----------------------------------------+
        | (1) member(c,[a,b,c,d])                 |
        |     +-------------------------------+   |
        |     | (2) member(c,[b,c,d])         |   |
        |     |     +-----------------+       |   |
  ?-    |     |     | (3) member(c,[c,d])   |   |
---->---+-->--+-->--+-----------------+-->--+-->--+--->----
        | --> | --> |  3P1 -->          |   |   |
        | 1P2 | 2P2 +-------------------+   |   |   yes
        |     +-------------------------------+   |
        +-----------------------------------------+
```

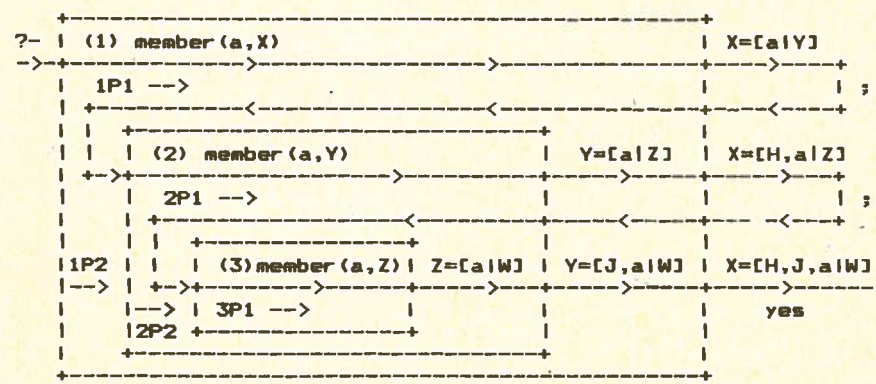
obr. 5.4

```
      +-------------------------------------------------+
      | (1) member(q,[a,b,c])                           |
      |     +-----------------------------------------+ |
      |     | (2) member(q,[b,c])                     | |
      |     |     +-------------------------------+   | |
      |     |     | (3) member(q,[c])             |   | |
      |     |     |     +-----------------+       |   | |
  ?-  |     |     |     | (4) member(q,[])      |   | |
-->---+-->--+-->--+-->--+-->--+-->--+           |   | |
      | 1P2 | 2P2 | 3P2 |       |               |   | |
      | --> | --> | --> |       |               |   | |
--<---+--<--+--<--+--<--+--<--+--<--+           |   | |
no    |     |     |     +-----------------+       |   | |
      |     |     +-------------------------------+   | |
      |     +-----------------------------------------+ |
      +-------------------------------------------------+
```

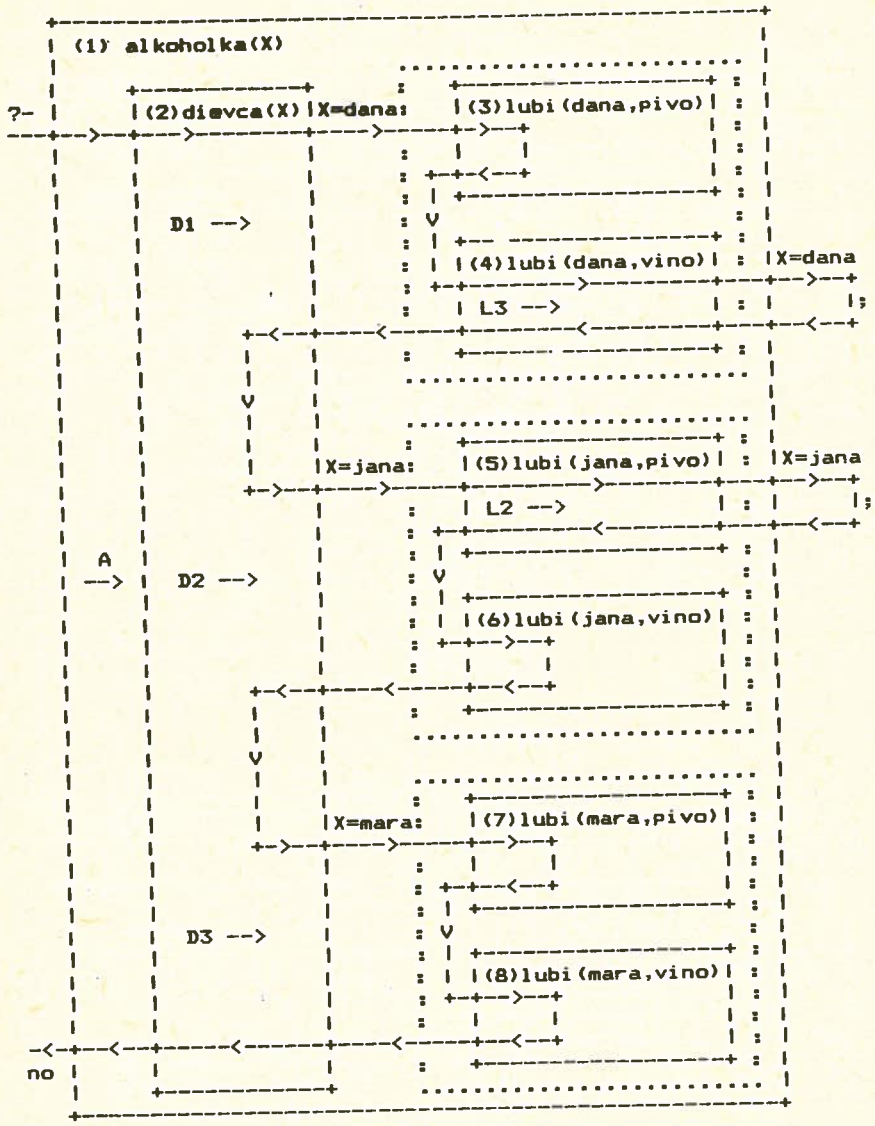
obr. 5.5


?-
(1) member(X,[a,b,c])
X=a
1P1 -->
;
(2) member(X,[b,c])
X=b
X=b
2P1 -->
;
(3) member(X,[c])
X=c
X=c
X=c
3P1 -->
;
1P2 -->
2P2 -->
(4) member(X,[])
3P2 -->
no

obr. 5.6


?-
(1) member(a,X)
X=[a|Y]
1P1 -->
;
(2) member(a,Y)
Y=[a|Z]
X=[H,a|Z]
2P1 -->
;
1P2 -->
(3)member(a,Z)
Z=[a|W]
Y=[J,a|W]
X=[H,J,a|W]
3P1 -->
2P2
yes

obr. 5.7


(1) alkoholka(X)
?-
(2)dievca(X)
X=dana:
(3)lubi(dana,pivo)
D1 -->
(4)lubi(dana,vino)
X=dana
L3 -->
X=jana:
(5)lubi(jana,pivo)
X=jana
L2 -->
A
-->
D2 -->
(6)lubi(jana,vino)
X=mara:
(7)lubi(mara,pivo)
D3 -->
(8)lubi(mara,vino)
no

obr. 5.8

## 5.2. DISJUNKCIA V BLOKOVÝCH MODELOCH

Modifikované blokové modely možno použiť aj v prípade, keď sa v tele niektorého pravidla vyskytne disjunkcia subcieľov. Tejto disjunkcii bude odpovedať fiktívny blok (budeme ho označovať bodkovanou čiarou.

Príklad 5.4.

Doplňme klauzuly z príkladu 5.1 o ďalšiu:

```
/* A */  alkoholka(X) :- dievca(X),
                         ( lubi(X,pivo) ; lubi(X,vino) ).
```

Zostavte blokový model pre otázku "ktoré dievčatá radi pijú".

Riešenie.

Príslušný blokový model je uvedený na obr. 5.8.

## 5.3. NEGÁCIA V BLOKOVÝCH MODELOCH

Modifikované blokové modely umožňujú aj reprezentáciu negovaných subcieľov. Štandardné trasovanie nepodáva informáciu o plnení subcieľov v argumente predikátu not . Preto by už značka ani správne v obrázku nemala byť uvedená, ale budeme ju v ďalšom zakresľovať, aby sme vyznačili klauzulu, ktorá spôsobi neúspech.

Situácia sa stáva zložitejšou, keď argumentom predikátu not je konjunkcia subcieľov. Keď chceme získať informáciu o činnosti PROLOGu pri pokuse o splnenie tejto konjunkcie, môžeme nahradiť predikát not predikátom nie :

```
nie(X) :- not(X).
```

Namiesto podrobného trasovania nastavíme tentokrát iba body zastavenia na všetky neštandardné predikáty (vrátane nie ) - zamedzíme tým "duplicitnému" výpisu predikátu not aj nie .

**Príklad 5.5.**

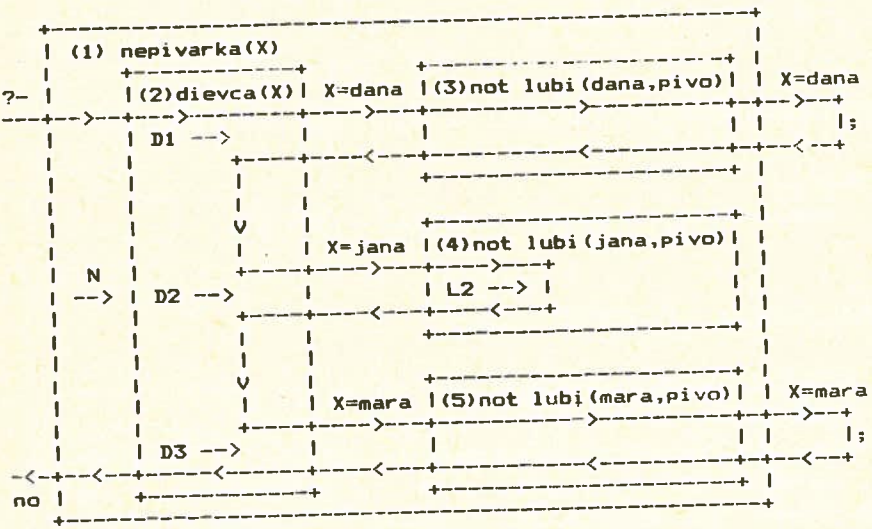
Doplňme predošlé klauzuly o ďalšiu:

```
/* N */  nepivarka(X):-dievca(X), not lubi(X,pivo).
```

Vyšetrite chovanie systému pri zodpovedani otázky "ktoré dievčatá neľúbia pivo".

**Riešenie.**

Priebeh dialógu a činnosť PROLOGu sú patrné z obr. 5.9.




obr. 5.9

**Príklad 5.6.**

Vyšetrime chovanie systému pri plnení cieľa mizogyn(X), opisujúceho takých chlapcov, ktorí neľúbia žiadne dievčatá. Predpokladáme, že v databáze sú naviac fakty, definujúce predikát chlapec (je to nutné, nakoľko predikát not nemožno použiť na generovanie alternatív, ale iba na ich testovanie):

```
mizogyn(X) :- chlapec(X),
              not(( dievca(Y), lubi(X,Y) )).

chlapec(peter).    chlapec(fero).
```

Riešenie:

Ladiaci výpis bude nasledovný:

```
(1) Call: mizogyn(X)
 (2) Call: chlapec(X)
 (2) Exit: chlapec(peter)
 (3) Call: not(dievca(Y),lubi(peter,Y))
 (3) Fail: not(dievca(Y),lubi(peter,Y))
 (2) Redo: chlapec(peter)
 (2) Exit: chlapec(fero)
 (3) Call: not(dievca(Y),lubi(fero,Y))
 (3) Exit: not(dievca(Y),lubi(fero,Y))
(1) Exit: mizogyn(fero)
```

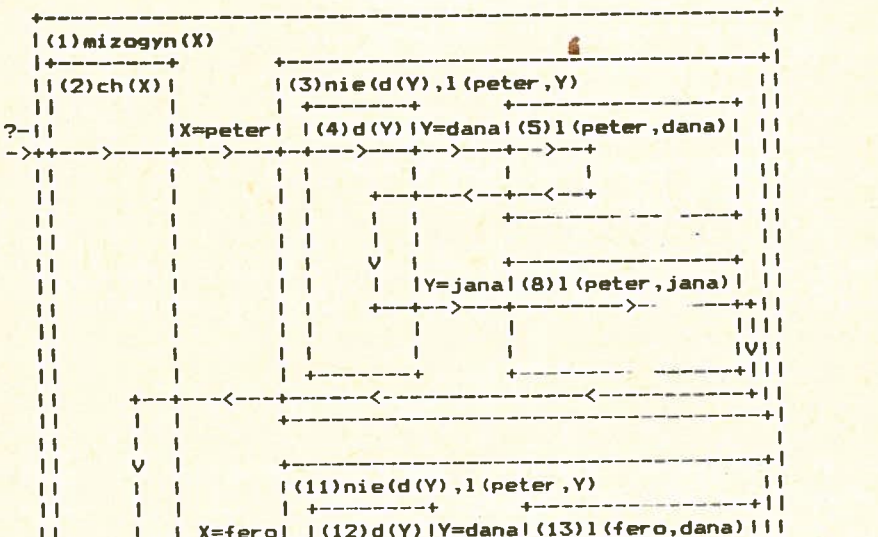
Keď chceme získať informáciu o činnosti PROLOGu pri pokuse o splnenie tejto konjunkcie, môžeme urobiť nasledujúcu úpravu:

```
mizogyn(X) :- chlapec(X),
              nie(( dievca(Y), lubi(X,Y) )).
```

Namiesto podrobného trasovania nastavíme tentokrát iba body zastavenia na predikáty chlapec , dievca , lubi , a nie (zamedzíme tým "duplicitnému" výpisu predikátu not aj nie ). Okrem toho nastavíme bránu Call na ovládanú, aby sme mohli preskočiť (pomocou funkcie skip ) podrobný výpis informácii o plnení subcieľa lubi . Začiatok ladiaceho výpisu bude nasledovný:

```
(1) Call: mizogyn(X) ?<CR>
 (2) Call: chlapec(X) ?<CR>
 (2) Exit: chlapec(peter)
 (3) Call: nie(dievca(Y),lubi(peter,Y)) ?<CR>
  (4) Call: dievca(Y) ?<CR>
  (4) Exit: dievca(dana)
  (5) Call: lubi(peter,dana) ?s
  (5) Fail: lubi(peter,dana)
  (4) Redo: dievca(dana)
  (4) Exit: dievca(jana)
  (8) Call: lubi(peter,jana) ?s
  (8) Exit: lubi(peter,jana)
 (3) Fail: nie(dievca(Y),lubi(peter,Y))      atď ...
```

V uvedenom výpise sme "preskočili" výpis informácií o blokoch 6 a 7 (odpovedajú subcieľom v tele pravidla L4 ). Príslušný blokový model je na obr.5.10 (aby sme zmenšili horizontálny rozmer obrázku, v záhlaví blokov sú uvedené iba počiatočné písmená mien predikátov chlapec , dievca , ľubi a nie sú zakreslené ani značky).




obr.5.10

## 5.4 REZ V BLOKOVÝCH MODELOCH

V blokových modeloch možno veľmi názorne sledovať vplyv symbolu rezu na činnosť systému PROLOG. Pri navracaní cez blok rezu sa pokračuje z jeho brány Fail priamo na bránu Fail nadradeného cieľa.

Príklad 5.7.

Vyšetrite činnosť PROLOGu pri použití predikátu pre budovanie zoznamu z argumentov faktov typu p(Arg) definovaného nasledovne (jedná sa vlastne o predikát collect z príkladu 4.18, ovšem bez symbolu rezu):

```
c([H|T]) :- retract(p(H)), c(T).
c([]).

p(a).  p(b).
```

Riešenie:

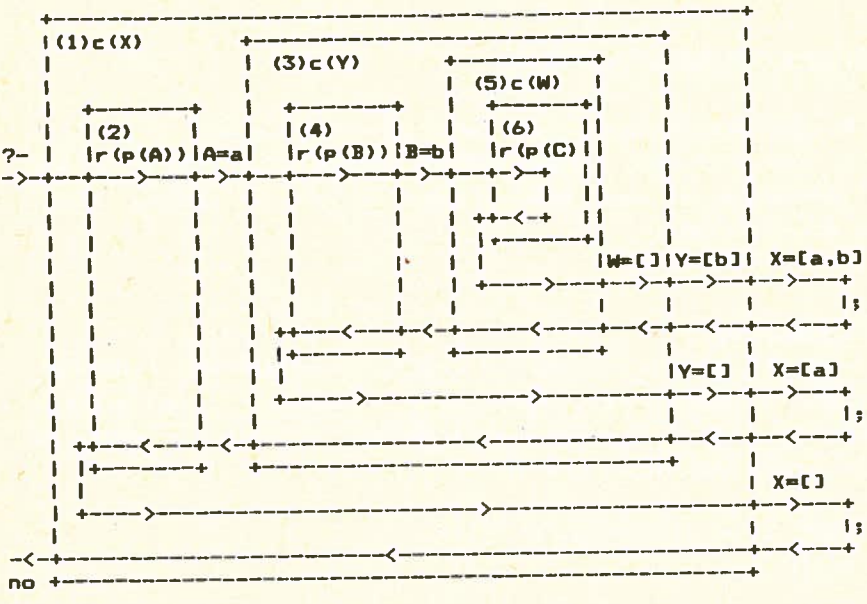
Hľadaný blokový model je na obr. 5.11. Ako je vidieť, pri navracaní sa zbytočne generujú ďalšie "alternatívy" - dielčie zoznamy, ktoré predstavujú medzivýsledky jednotlivých krokov rekurzie. Z dôvodov zmenšenia horizontálneho rozmeru obrázku je uvedené iba začiatočné písmeno mena predikátu retract .

Príklad 5.8.

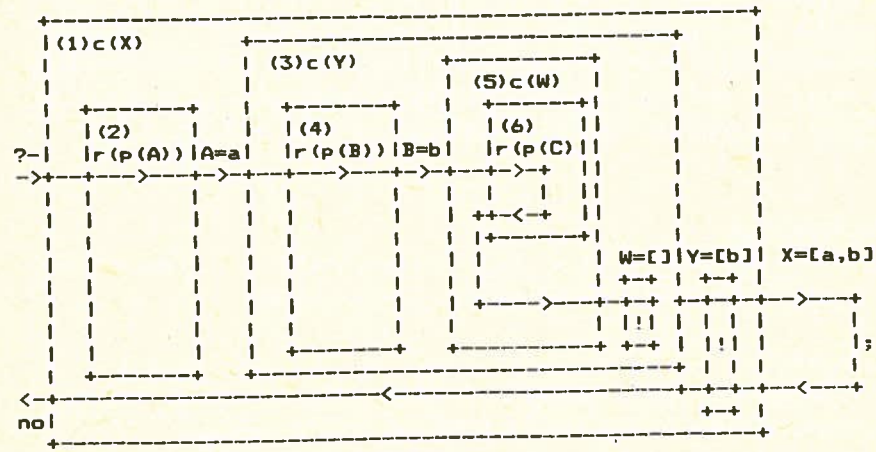
Vyšetrite vplyv symbolu rezu, umiestneného na konci prvej klauzuly v definícii predikátu c v príklade 5.7.

Riešenie.

Z blokového modelu na obr.5.12 je patrné, že sa použitím rezu odstránilo nežiadúce generovanie falošných výsledkov.




obr.5.11




obr.5.12

# 6. SYMBOLICKÉ MANIPULACIE

## 6.1. ARITMETIKY PRE ZÁPORNÉ ČÍSLA

Nakoľko štandardný PROLOG pracuje iba s prirodzenými číslami, je niekedy potrebné doplniť do systému predikáty pre prácu so zápornými číslami. Predovšetkým je potrebné definovať operátor unárne minus :

```
?- op(4,fx,¯).
```

Príklad 6.1.

Navrhnite predikát pre súčet dvoch celých čísiel (môžu byť kladné aj záporné). Predpokladáme, že sčítance sú konštanty, alebo premenné viazané na konštanty, t.j. nepožadujeme "generačnú" schopnosť.

Riešenie:

Pre každú zo štyroch možných kombinácii znamienok operandov potrebujeme samostatné pravidlo:

```
sum( A, B, V) :- integ2(A,B), V is A + B.
sum( A,¯B, V) :- integ2(A,B),
                 (A >= B, !,   V is A - B        ;
                               W is B - A, V = ¯W ).
sum(¯A, B, V) :- integer(B), sum(B,¯A,V).
sum(¯A,¯B,¯V) :-             sum(A, B,V).

integ2(X,Y) :- integer(X), integer(Y).
```

V treťom pravidle stačí testovať, či druhý argument je nezáporný, pri pokuse o splnenie subcieľa sum(B,¯A,V) pomocou druhého pravidla sa preverí nezápornosť A .

Test:

```
?- sum(¯5,3,X).
X = ¯2
yes
```

Príklad 6.2.

Navrhnite predikát pre násobenie celých čísiel.

Riešenie:

```
mul( A, B, V) :- integ2(A,B), V is A * B.
mul( A,~B,~V) :- integ2(A,B), V is A * B.
mul(~A, B, V) :- integer(B), mul(B,~A,V).
mul(~A,~B, V) :-             mul(A, B,V).
```

Pri použití takto navrhnutého predikátu vzniká jeden problém: pri násobení záporného čísla nulou je výsledkom záporná nula ~0. Uvedený nedostatok odstránime pridaním dvoch pravidiel pre násobenie nulou. V ostatných pravidlách je ovšem nutné upraviť test na typ operandov - nesmú byť ani nulové:

```
mul( 0, B, 0) :- integ(B).
mul( A, 0, 0) :- nonzero(A).
mul( A, B, V) :- cardinal(A,B), V is A * B.
mul( A,~B,~V) :- cardinal(A,B), V is A * B.
mul(~A, B, V) :- cardinal(A,B), mul(B,~A,V).
mul(~A,~B, V) :-                mul(A, B,V).

cardinal(X,Y) :- cardinal(X), cardinal(Y).

cardinal(X)   :- integer(X), X \= 0.

nonzero(X)    :- integ(X), X \= 0.

integ( X)     :- integer(X).
integ(~X)     :- integer(X).
```

Test:

```
?- mul( 3,~5,X).
X = 15
yes
```

## 6.2. ARITMETIKY PRE ČÍSLA V DESATINNOM TVARE

Predpokladajme, že čísla v desatinnom tvare budú zadané ako štruktúra, ktorej argumentami budú celá a desatinná časť čísla a funktorom bude infixný operátor dvojbodka - **:** , deklarovaný nasledovne:

```
?- op(2, xfx, :).
```

Desatinná časť bude maximálne dvojmiestna (kvôli jednoduchším aritmetikám). Jednomiestna desatinná časť je povolená, ale bude sa chápať odlišne od bežných zvyklostí, napr. zápis 32:1 sa bude interpretovať ako 32.01 a nie ako 32.10. Pre zápis záporných čísiel použijeme unárne mínus deklarované v kapitole 6.1.

Predikáty pre jednotlivé aritmetické operácie budeme koncipovať tak, že keď niektorý argument bude celočíselný, potom sa pred vykonaním vlastnej operácie pretransformuje na desatinné číslo s nulovou desatinnou časťou (pomocou predikátu split ). Je to síce výpočtovo náročnejšie riešenie, ale nie je potrebné zostavovať samostatné pravidlá pre rôzne kombinácie typov argumentov.

Priklad 6.3.

Navrhnite predikát pre sčítanie čísiel v desatinnom tvare.

Riešenie:

Požadovaný predikát definujeme štyrmi základnými pravidlami pre rôzne kombinácie znamienok operandov:

```
sum(A,B,V) :- split(A,IntA,FractA),
              split(B,IntB,FractB),
              integ2(IntA,IntB),
              F is FractA + FractB,
              Fract is F mod 100, Transf is F / 100,
              Int is IntA + IntB + Transf,
              split(V,Int,Fract), !.
```

```
sum(A,¯B,V) :-
   split(A,IntA,FractA), split(B,IntB,FractB),
   integ2(IntA,IntB),
   (FractA >= FractB, F is     FractA-FractB, Transf=0;
                      F is 100+FractA-FractB, Transf=1),
   sum(IntA,¯Transf,X), sum(X,¯IntB,Int),
   (F=0, V=Int                                          ;
         (integer(Int), V=Int:F                         ;
                        Fract is 100 - F,
                        (Int= ¯1, Int1=0            ;
                                  sum(1,Int,¯Int1) ),
                        V= ¯(Int1:Fract)                ) ),!.

sum(¯A, B, V) :- posit(A,B), sum(B,¯A,V).
sum(¯A,¯B,¯V) :- posit(A,B), sum(A, B,V).

split( C ,C,0) :- integer(C).
split(C:D,C,D).

posit(A:B) :- integ2(A,B).
posit( A ) :- integer(A).
posit(A,B) :- posit(A), posit(B).
```

V tele prvého pravidla pre definíciu predikátu sum sa najprv zistia celé a desatinné časti operandov a otestuje sa, či sú oba operandy kladné. Potom sa urobí zvlášť súčet desatinných častí a celých častí (keď je súčet desatinných častí väčší ako 100, prenesie sa jednotka do súčtu celých častí). Na konci sa vygeneruje výsledok (keď desatinná časť je nulová, výsledok je celé číslo).

Druhé pravidlo rieši vlastne odčítanie dvoch čísiel. Po úvodných krokoch (zhodných so začiatkom činnosti predošlého pravidla) sa odčítajú desatinné časti (v prípade potreby je prenos jednotky z vyššieho rádu). Potom sa odčítajú celé časti pri zohľadnení prenosu. Keď zlomková časť výsledku je nulová, výsledok je celočíselný. Pre kladnú celočíselnú časť sa výsledok zostaví priamo z celočíselnej a desatinnej časti. V prípade zápornej celočíselnej časti je výsledok záporný. Jeho desatinná časť sa určí ako doplnok získanej desatinnej časti E do stovky.

Celočíselná časť Int1 záporného výsledku sa získa pripočítaním jednotky k pôvodnej celej časti Int a zmenou znamienka. Zmenu znamienka realizujeme v treťom argumente subcieľa sum . Aby sme nedostali zápornú nulu, je potrebné prípad nulovej celočíselnej časti záporného výsledku ošetriť zvlášť.

Aby sme sa vyhli cyklickej definícii tohoto pravidla, musíme pred jeho definíciu doplniť ravidlo pre odčítanie dvoch celých čísiel (v porovnaní s kapitolou 6.1 je potrebné umiestniť symbol rezu na koniec pravidla, aby sa po jeho úspešnom použiti zabránilo prípadnému použitiu vyššie uvedeného pravidla pre prvý kladný a druhý záporný argument):

```
sum(A,¯B,V) :- integ2(A,B),
               (A >= B,  V is A - B          ;
                         W is B - A, V = ¯W ), !.
```

Test:

```
?- sum(¯2:22,3,X).
X = 0:78
yes

?- sum(¯2:22,1:22,X).
X = ¯1
yes
```

Príklad 6.4.

Navrhnite predikát pre násobenie čísiel v desatinnom tvare.

Riešenie:

```
mul( 0 , X ,0) :- number(X), !.
mul( X , 0 ,0) :- number(X), !.
mul( X ,0:0,0) :- number(X), !.
mul(0:0, X ,0) :- number(X), !.
mul( A , B ,V) :- split(A,IntA,FractA),
                  split(B,IntB,FractB),
                  integ2(IntA,IntB),
                  mulff(FractA,FractB,F),
                  mulif(IntA,FractB,I1:F1),
                  mulif(IntB,FractA,I2:F2),
```

```
                    Fract is F1 + F2 + F,
                    Transf is Fract / 100,
                    FractV is Fract mod 100,
                    IntV is IntA * IntB + I1 + I2 + Transf,
                    split(V,IntV,FractV),!.
mul( A, B, V)  :- posit(A,B), mul(A,B,VV),
                    (VV = 0, V = 0 ; V = ¯VV), !.
mul( A, B, V)  :- posit(A,B), mul( B,A,V).
mul(¯A,¯B, V)  :- posit(A,B), mul( A,B,V).

mulff(A,B, V ) :- W is A * B, V is (W + 50) / 100.

mulif(A,B,C:Z) :- W is A * B, C is W / 100, Z is W mod 100.

number( X  ) :- integ(X).
number( A:B) :- posit(A:B).
number( A:B) :- posit(A:B).
```

Podobne ako v prípade násobenia celých čísiel (kapitola 6.1) sme na začiatku uviedli štyri pravidlá pre násobenie nulou. Tým zabránime, aby pri násobení záporného čísla nulou bola výsledkom záporná nula.

Predikát mulff slúži k vynásobeniu desatinných časti operandov so zaokrúhlením na dve miesta. Pôvodne zvolené obmedzenie na dvojmiestnu desatinnú časť vyplýva práve z tejto etapy výpočtu - v prípade dlhších desatinných časti by bolo potrebné ošetriť pripady, v ktorých by pri násobení došlo k preplneniu.

Predikát mulif realizuje násobenie celej časti jedného operandu s desatinnou časťou druhého s následným prevodom výsledku do desatinného tvaru (použitie predikátu mul by viedlo k cyklickej definícii).

Test:

```
?- mul( 2:50,4,X).
X = ¯10
yes
```

```
?- mul(2:22,1:11,X).
X = 2:46
yes
```

Priklad 6.5.

Navrhnite predikát pre umocňovanie: základ je celé číslo, alebo číslo v desatinnom tvare a exponent je celé nezáporné číslo.

Riešenie:

Umocnenie zrealizujeme pomocou prislušného počtu rekurzivne volaných násobení:

```
pow(N,M,V) :- number(N), zardinal(M), M1 is M - 1,
              pow(N,M1,W), mul(N,W,V).
pow(N,0,1) :- number(N).
```

Test:

```
?- pow( 2:50,2,X).
X = 6:25
yes

?- pow(¯5,3,X).
X =  125
yes
```

## 6.3. ZJEDNODUŠOVANIE ARITMETICKÝCH VÝRAZOV

Priklad 6.6.

Navrhnite predikát pre zjednodušovanie aritmetických výrazov, ktorý by realizoval úpravy typu

5 + 3*a + 8 + b + a + 7 + 4*b -> 20 + 4*a + 5*b

Riešenie:

Pri riešení je účelné postupovať procedurálne:

A / vytvoríme zoznam L jednotlivých sčítancov pomocou predikátu adt decomp

B1/ celočíselné sčítance budeme kumulovať v druhom argumente predikátu adt_comp

B2/ sčítance tvorené atómom, resp štruktúrou číslo*atóm budeme kumulovať v databáze v tvare faktov elem(Atom,Pocet), generovaných predikátom adt_comp

B3/ výsledný výraz vytvoríme z výsledku, získaného v bode B1 a z faktov, generovaných v bode B2 pomocou predikátu adt_end

Hlavný predikát definujeme nasledovne

```
adt_simpl(I,O) :- retractall(elem(_,_)), adt_decomp(I,[],L),
                  adt_comp(L,0,O).
```

kde I je spracovávaný výraz a O je jeho upravený tvar.

Predikát podľa bodu A má rekurzívny charakter

```
adt_decomp(A+B,T,L) :- adt_decomp(A,[B|T],L).
adt_decomp(A,T,[A|T]).
```

kde prvý argument reprezentuje ešte nespracovanú časť aritmetického výrazu, v druhom sa buduje postupne zoznam sčítancov, ktorý sa po skončení rozkladu prenesie do tretieho argumentu ako výsledok.

Rekurzívny charakter má aj definicia predikátu podľa bodov B1, B2:

```
adt_comp([],S,V)        :- adt_end(S,V).
adt_comp([H|T],S,V)     :- integer(H), S1 is S+H,
                           adt_comp(T,S1,V).
adt_comp([N*A|T],S,V)   :- integer(N), add(A,N),
                           adt_comp(T,S,V).
adt_comp([H|T],S,V)     :- add(H,1),
                           adt_comp(T,S,V).
```

Prvý argument predstavuje postupne skracovaný zoznam sčítancov,

v druhom argumente sa kumuluje súčet celočíselných sčítancov a treti argument reprezentuje upravený výraz, generovaný predikátom adt_end . Predikát add realizuje kumuláciu ostatných sčítancov v databáze:

```
add(A,N) :- retract(elem(A,N1)), N2 is N+N1,
            assertz(elem(A,N2)).
add(A,N) :- asserta(elem(A,N )).
```

Prvé pravidlo testuje, či už predtým bol nájdený sčítanec typu A ( N1 je potom doterajši násobok) a v pripade uspešného testu zvýši stav počítadla na N2 . V opačnom pripade (prvý výskyt sčítanca typu A) sa príslušný fakt uloží do databázy.

Výsledný, upravený aritmetický výraz sa generuje nasledovne

```
adt_end(0,W) :- ea(X), adt_end(X,W).
adt_end(S,W) :- ea(X), adt_end(S+X,W).
adt_end(S,S).
```

V prvom argumente sa postupne buduje upravený výraz, ktorý sa na konci spracovania (keď v DB už nie je žiadny fakt elem ) prenesie do druhého argumentu ako výsledok. Prvé pravidlo slúži k zahájeniu budovania upraveného výrazu v pripade, že v pôvodnom výraze sa nevyskytli číselné sčítance (bez tohoto pravidla by výsledok v podobnom pripade začinal zbytočne 0+...).

Pomocný predikát ea okrem odoberania faktov elem z databázy upravuje aj sčítance výsledku v špeciálnych prípadoch (členy s násobkom 0 vynecháva a v pripade násobku 1 vynechá z výsledku koeficient 1 ):

```
ea(X) :- retract(elem(A,N)),
         (N=0, !, ea(X) ; N=1, !, X=A ; X=N*A).
```

Keď sčítance sú zložitejšie (napr. 2*a*b+4*a*b ), potom ostanú v pôvodnom tvare - je to dôsledok ľavej asociativity operátora násobenia: 2*a*b=(2*a)*b . Predefinovaním op(31,xfy,*) na pravú asociativitu možno síce dosiahnuť, že pre uvedené sčítance

sa získa úprava na 6*a*b , ale ako uvedieme v ďalšom je možné tento problém riešiť aj iným spôsobom. Zmena asociativity by v prípade zložitejších výrazov (najmä v kombinácii s operátorom delenia spôsobovala problémy), nakoľko ľavá asociativita sa pokladá obvykle za samozrejmú (20/10/2=1).

Príklad 6.7.

Upravte predikáty z predošlého prikladu tak, aby akceptovali aj reálne čísla a aj operátor odčítania. Návod: Doplňte definíciu predikátu adt_decomp tak, aby sa odčítanie pretransformovalo na pripočítanie člena so záporným znamienkom. Nahraďte predikáty pracujúce s prirodzenými číslami príslušnými predikátmi pre čísla v desatinnom tvare (viď kapitola 6.2).

Riešenie:

```
adt_decomp(A+B,   T,L) :- adt_decomp(A,[B|T],L).
adt_decomp(A-B,   T,L) :- posit(B), adt_decomp(A,[¯B|T],L).
adt_decomp(A-N*B,T,L) :- posit(N), adt_decomp(A,[ N*B|T],L).
adt_decomp(A-B,   T,L) :- adt_decomp(A,[¯1*B|T],L).
adt_decomp(A,     T,[A|T]).

adt_comp([],      S,V) :- adt_end(S,V).
adt_comp([H|T],   S,V) :- sum(S,H,S1), adt_comp(T,S1,V).
adt_comp([N*A|T],S,V) :- number(N), add(A,N),adt_comp(T,S,V).
adt_comp([H|T],   S,V) :- add(H,1), adt_comp(T,S,V).

add(A,N) :- retract(elem(A,N1)), sum(N1,N,N2),
            assertz(elem(A,N2)).
add(A,N) :- assertz(elem(A,N)).

adt_end(0,W) :- ea(X,Z),
                (Z= +, !, adt_end( X ,W) ; adt_end(  X,W)).
adt_end(S,W) :- ea(X,Z),
                (Z= +, !, adt_end(S+X,W) ; adt_end(S-X,W)).
adt_end(S,S).

ea(X,Z) :- retract(elem(A,M)),
           (M=0, !, ea(X,Z) ; (M= ¯N, !, Z= - ; M=N, Z= +),
                              (N=1, !, X=A ; X=N*A)        ).
```

Definície predikátov <u>sum</u> , <u>number</u> , <u>posit</u> sú uvedená v kapitole 6.2.

Test:

```
?- adt_simpl(3+a-b-2:55-3:25*a+5:00*b+3:25,X).
X = 3:70-2:25*a+4*b
yes

?- adt_simpl(3-4*a+b+3*a-3+a-c,X).
X = b-c
yes
```

<u>Príklad 6.8.</u>

Navrhnite predikát pre zjednodušovanie aritmetických výrazov, realizujúci úpravy typu:

```
5*a*3/a^4*b^2*4/b -> 60/a^3*b
```

Riešenie:

Deklarujeme operátor umocňovania

```
?- op(6,xfy,^).
```

Príslušné definičné klauzuly pre požadované úpravy sú

```
mul_simpl(I,O) :- retractall(elem(_,_)),
                  mul_decomp(I,[],L), mul_comp(L,1,O).

mul_decomp(A* ¯B, T,   L     ) :- mul_decomp(A,    [¯1,B|T],L).
mul_decomp(A*B,   T,   L     ) :- mul_decomp(A,       [B|T],L).
mul_decomp(A/B^N, T,   L     ) :- mul_decomp(A,   [B^ ¯N|T],L).
mul_decomp(A/ ¯B, T,   L     ) :- mul_decomp(A,[¯1,B^ ¯1|T],L).
mul_decomp(A/B,   T,   L     ) :- mul_decomp(A,   [B^ ¯1|T],L).
mul_decomp(¯A,    T,[¯1,A|T]).
mul_decomp(A,     T, [A|T]   ).

mul_comp(  [],    S,V) :- mul_end(S,V).
mul_comp( [H|T], S,V) :- number(H), mul(S,H,S1),
                         mul_comp(T,S1,V).
mul_comp([H^N|T],S,V) :- number(N),add(H,N), mul_comp(T,S,V).
```

```
mul_comp( [H|T], S,V) :-            add(H,1), mul_comp(T,S,'').

mul_end( 1,W) :- em(X,Z),
                 (Z= -, !, mul_end( 1/X,W) ; mul_end( X ,W)).
mul_end(¯1,W) :- em(X,Z),
                 (Z= -, !, mul_end(¯1/X,W) ; mul_end(¯X ,W)).
mul_end( S,W) :- em(X,Z),
                 (Z= +, !, mul_end( S*X,W) ; mul_end(S/X,W)).
mul_end( S,S).

em(X,Z) :- retract(elem(A,M)),
           (M=0, !, em(X,Z) ; (M= ¯N, !,Z= - ; M=N, Z= +),
                               (N=1, !, X=A ; X=A^N)        ).
```

Definícia predikátu <u>add</u> je uvedená v príklade 6.6 a prediká-tu <u>mul</u> v príklade 6.4.

Test:

```
?- mul_simpl(1:50*a/b*3/a^2:50*b,X).
X = 4:50/a^1:50
yes

?- mul_simpl(a^2:50/b^2:50/a^1:50*b^0:50,X).
X = a/b^2
yes

?- mul_simpl(¯a*a,X).
X = ¯(a^2).
yes

?- mul_simpl(¯a* ¯b,X).
X = a*b
yes
```

Bez prvého a šiesteho pravidla v definícii predikátu <u>mul decomp</u> by odpoveď systému na posledné dve otázky bola neúplná: <u>¯a*a</u> , resp. <u>¯a* ¯b</u> . Bez štvrtého pravidla by ostali neupravené výrazy typu <u>a^3/ ¯a</u> .

Príklad 6.9.

Upravte predikáty z príkladov 6.7 a 6.8 tak, aby realizovali úpravy typu:

5+3*a*5/b+a*b^2*3/b^3+5*4 -> 25+18*a/b

Riešenie:

V prvej etape zostrojíme zoznam sčítancov s využitím predikátu adt_decomp , na každý z nich aplikujeme mul_simpl a z takto upravených prvkov zostavíme výsledok pomocou adt_comp :

```
simpl(I,O) :- retractall(elem(_,_)), adt_decomp(I,[],LI),
              rewrite(LI,LO),        adt_comp(LO,0,O).

rewrite([],[]).
rewrite([HI|TI],[HO|TO]) :- mul_simpl(HI,HO), rewrite(TI,TO).
```

Uvedený návrh rieši vytýčenú úlohu iba čiastočne :

```
?- simpl(2*3*4+2*a*3+3*a*b+a+a*b+5*5,X).
X = 49+3*a*b+7*a+a*b
yes
```

Problém je v tom že, vzhľadom k ľavej asociativite operátora násobenia platí 3*a*b = (3*a)*b a tak tretie pravidlo v definícii predikátu adt_comp nespracuje členy tohoto typu. Je potrebné čiastočne upraviť predikát mul_end tak, aby najprv vytvoril rekurzívne štruktúru súčinov nenumerických prvkov a až pred ňu predradil príslušný číselný koeficient (napr. v našom príklade 3*(a*b) ):

```
*--b                    *--*--b
|                       |  |
*--a                    3, a
|
3

3*a*b                   3*(a*b)
```

Prípady, keď tento koeficient má hodnoty 0, ¯1, 1 , je potrebné ošetriť zvlášť:

```
mul_end(0,0) :- retractall(elem(_,_)).
mul_end(S,W) :- em(X,Z),
                (Z= -, !, mul_end(1/X,S,W) ; mul_end(X,S,W)).
mul_end(S,S).

mul_end(A,S,W) :- em(X,Z),
                (Z= +, !, mul_end(A*X,S,W) ; mul_end(A/X,S,W)).
mul_end(A, 1, A ).
mul_end(A,¯1,¯A ).
mul_end(A, S,S*A).
```

S ľavou asociativitou aditívnych a multiplikatívnych operátorov súvisí ešte jeden problém:

```
?- simpl(a*(b*a),X), simpl(a+(b+a),Y).
X = a*(b*a), Y=a+(b+a)
yes
```

Aby zjednodušenie prebehlo aj v týchto prípadoch, je potrebné doplniť po jednom pravidle pred definície príslušných predikátov:

```
mul_decomp(A*(B*C),T,L) :- mul_decomp(A*B*C,T,L).
adt_decomp(A+(B+C),T,L) :- adt_decomp(A+B+C,T,L).
```

Pre operátory odčítania a delenia by bolo potrebné doplniť ďalšie dve trojice pravidiel, môžu totiž vznikať rôzne kombinácie operandov, napr. A*(B/C), A/(B/C), A/(B*C) . Podobné problémy vznikajú aj pri riešení ďalších dielčich úloh úpravy zložitejších výrazov, preto sa zatiaľ obmedzíme iba na operátory sčítania a násobenia. V príkladoch 6.12 - 6.14 budú definované predikáty predspracovania východzieho výrazu a konečnej úpravy zjednodušeného výrazu, ktoré umožnia vylúčiť operátory odčítania a delenia z vlastného procesu zjednodušovania.

Príklad 6.10.

Navrhnite predikáty realizujúce zjednodušovanie aritmetických výazov typu:

3*exp(2*a+a)+exp((2+1)*a) -> 4*exp(3*a)

Riešenie:

Budeme postupne rozkladať štruktúru, reprezentujúcu východzí výraz a na každej úrovni rozkladu vykonáme možné zjednodušenia a postupne zase poskladáme štruktúru, predstavujúcu zjednodušený výraz (obr. 6.1).

```
+---------------exp
|               |
*--exp          *--a          *--exp
|   |           |      -->    |   |
3   +--a        +--1          4   *--a
    |           |                 |
    *--a        2                 3
    |
    2
```

obr. 6.1

V prípade unárnych funktorov použijeme pre rozklad a opätovné skladanie štandardný predikát =.. . V prípade operátorov sčítania a násobenia použijeme postup uvedený v príkladoch 6.6 a 6.8. Aj pre ne by bolo síce možno použiť predikát =.. , ale pri reťazení rovnakých operátorov by sa veľmi ťažko spracovávali výrazy so striedavým charakterom členov, napr. b*a^3*b*2*a*b^5 . Predbežne upustíme od spracovania operátorov odčítania a delenia (budú zohľadnené znova v príklade 6.12). Je teda možné z definície predikátov adt decomp a mul decomp vynechať príslušné pravidlá.

Hľadané riešenie má potom tento tvar:

```
simple(I,I) :- atom(I) ; number(I).
simple(I,O) :- I=..[Op,AI], simple(AI,AO),
               W=..[Op,AO], trans(W,O).
simple(I,O) :- functor(I,Op,2), proc(Op,I,O).

proc(+,I,O) :- adt_decomp(I,[],LI), rewr(LI,LO),
               adt_comp(LO,0,O).
proc(*,I,O) :- mul_decomp(I,[],LI), rewr(LI,LO),
               mul_comp(LO,1,O).
proc(^,I,I).

rewr([],[]).
rewr([HI|TI],[HO|TO]) :- simple(HI,HO), rewr(TI,TO).

trans(exp(0),1).
trans(exp(log(X)),X).
trans(X,X).
```

Test:

```
?- simple((3+2)*a*(4*3+2)*a^2+a^3,X).
X = 71*a^3
yes

?- simple(a*(a+a),X).
X = a*(2*a)
yes
```

Neúplná úprava výsledku v druhom príklade je spôsobená skutočnosťou, že pri plnení cieľa <u>proc(*,a*(a+a),O)</u> sa pri splnení subcieľov v tele príslušného pravidla získajú nasledovné viazania premenných:

```
mul_decomp(a*(a+a),[],LI)    ...   LI = [a,(a+a)]
rewr([a,(a+a)],LO)           ...   LO = [a,(2*a)]
mul_comp([a,(2*a)],1,O)      ...   O  = a*(2*a)
```

Riešenie naznačeného problému predefinovaním predikátu <u>rewr</u> bude uvedené v príklade 6.13.

Predikát trans ošetruje niektoré špeciálne prípady, napr.:

```
?- simple(exp(0*a)+exp(log(a*a)),X).
X = 1+a^2
yes
```

Príklad 6.11.

Doplňte definíciu predikátu proc tak, aby realizoval úpravy typu:

a^2^3 -> a^8 (a^3)^2 -> a^6

a definíciu predikatu trans tak, aby ošetril špeciálne prípady pre ďalšie funkcie (log, sin, cos) a pre umocňovanie.

Riešenie:

Posledné pravidlo definujúce predikát proc v príklade 6.10 je potrebné nahradiť trojicou predikátov

```
proc(^, A^B,      V  ) :- pow(A,B,V).
proc(^,(A^B)^C,   O  ) :- proc(*,B*C,W), simple(A^W,O).
proc(^, A^B,    AA^BB) :- simple(A,AA), simple(B,BB).
```

pričom sa využíva predikát pow pre číselné argumenty (viď príklad 6.5). Definíciu predikátu trans doplníme o fakty:

```
trans(log(1),0).                 trans(log(exp(X)),X).
trans(sin(0),0).                 trans(sin(¯X),¯sin(X)).
trans(sin(¯N*X),¯sin(N*X)).      trans(cos(¯N*X),cos(N*X)).
trans(cos(0),1).                 trans(cos(¯X),cos(X)).
trans(¯ ¯X,X).                   trans(1^N,1).
trans(X^1,X).                    trans(X^0,1).
```

pričom ale fakt trans(X,X) musí ostať na konci. Je potrebné doplniť do tela tretieho pravidla definície predikátu simple subcieľ trans :

```
simple(I,O) :- functor(I,Op,2), proc(Op,I,W), trans(W,O).
```

Test:

```
?- simple(¯sin(¯b)+a^log(cos(0)),X).
X = 1+sinb
yes
```

Príklad 6.12.

Navrhnite definíciu predikátu remove , ktorý odstráni zo spracovávaného výrazu operátory odčítania a delenia. Nech tento predikát umožní použiť vo východzom výraze pre odmocninu unárny predikát sqrt . Návod: Urobte postupný rozklad štruktúry reprezentujúci východzí výraz, na každej úrovni urobte požadovanú úpravu a potom vybudujte novú štruktúru výrazu bez zmienených operátorov (analogicky s druhým pravidlom pre simple v príklade 6.10).

Riešenie:

```
?-op(200,fx,?).

? X :- remove(X,Y), simple(Y,Z), write(Z).

remove(I,I) :- atom(I) ; number(I).
remove(I,O) :- I=..[Op,AI], remove(AI,AO),
               V=..[Op,AO], rem(V,O).
remove(I,O) :- I=..[Op,AI1,AI2], remove(AI1,AO1),
               remove(AI2,AO2),
               V=..[Op,AO1,AO2], rem(V,O).

rem(sqrt(X),X^0:50).
rem(A/B,A*B^ ¯1).
rem(A-B,A+ ¯1*B).
rem(X,X).
```

Test:

```
?- ?a^2/(5*a-4*a).
a
yes

?- ?a^(b+1)/a.
```

Nevhodná úprava výsledku v druhom prípade je spôsobená neúplnou definíciou predikátu mul_comp , ktorý kumuluje v databáze iba činitele so zhodným základom a s číselným exponentom, resp. bez exponentu (tretie a štvrté pravidlo pre mul_comp , príklad 6.8). Riešenie naznačeného problému bude uvedené v príklade 6.13.

Naviac je potrebné doplniť v definícii predikátu adt_comp pred posledné pravidlo ešte ďalšiu klauzulu:

```
adt_comp([¯H|T],S,V) :- add(H,¯1), adt_comp(T,S,V).
```

Bez neho by výrazy typu 3*a-a ostali neupravené.

Niektoré typy výrazov nie je možné zjednodušiť pomocou doteraz navrhnutých predikátov, napr.:

```
?- ?a/(a/b).
a/(a/b)
yes

?- ?a^2/(4*a-a).
a^2/(3*a)
yes
```

Po úpravách, uvedených v ďalšom príklade, bude možné uskutočniť aj tieto zjednodušenia.

Príklad 6.13.

Urobte potrebné zmeny v definícii nasledujúcich predikátov:

A./ Upravte predikát rewr tak, aby realizoval úpravy typu

a*(a+a) -> 2*a^2 namiesto a*(2*a) viď príklad 6.9

B./ Upravte predikát mul_comp a s ním súvisiace pomocné predikáty tak, aby realizoval úpravy typu

a^(b+1)/a -> a^b namiesto a^(1+b)/a viď príklad 6.12

C./ Upravte predikáty adt_end a mul_end tak, aby v upravenom výraze boli vždy členy s operátormi sčítania a násobenia pred členmi s operátormi odčítania a delenia.

Riešenie:

A./ Pôvodný predikát rewr , definovaný v príklade 6.10 premenujeme na adt_rewr a použijeme ho v tele pravidla s hlavou proc(+,I,O) , kdežto v tele pravidla s hlavou proc(*,I,O) použijeme subcieľ mul_rewr , pričom príslušný predikát je definovaný nasledovne:

```
mul_rewr([],[]).
mul_rewr([H|TI],LO) :- simple(H,SS), (SS= ~S ; SS=S),
                       mul_decomp(S,[],V), concat(V,TO,L),
                       (SS= ~S, LO=[~1|L] ; LO=L),
                       mul_rewr(TI,TO).

concat([],L,L).
concat([H|L1],L2,[H|L3]) :- concat(L1,L2,L3).
```

Prvý test SS= ~S zaručí, že v strednej časti tela druhého pravidla sa pracuje iba s činiteľom, pred ktorým nie je operátor unárne minus; keď pred pôvodným činiteľom bol tento operátor, potom sa do výsledného zoznamu pridá ~1 (viď druhý test). Keď je S štruktúra s násobením ako hlavným operátorom, potom subcieľ mul_decomp vytvorí zoznam V jeho činiteľov, ktorý sa pomocou predikátu concat pripojí k zoznamu TO . V opačnom prípade sa k zoznamu TO pripojí jediný prvok S .

B./ Pred posledné pravidlo v definícii predikátu mul_comp doplníme ďalšie pravidlo

```
mul_comp([H^E|T],S,V) :- addl(H,E), mul_comp(T,S,V).
```

a dodefinujeme nový pomocný predikát addl :

```
addl(H,E) :- retract(elem(1,H,T)), append(E,T,ET),
             assertz(elem(1,H,ET)).
addl(H,E) :- retract(elem(_,H,S)), append(E,[S],ES),
             assertz(elem(1,H,ES)).
```

```
addl(H,E) :- apend(E,[],EE), assertz(elem(1,H,EE)).

apend(A,B,V) :- adt_decomp(A,[],C), beg, adt_rewr(C,D), end,
                concat(D,B,V).

beg :- retract(elem(A,B,C)), assertz(ex(A,B,C)), beg.
beg.

end :- retract(ex(A,B,C)), assertz(elem(A,B,C)), end.
end.
```

Nové pravidlo pre mul_comp ošetrí prípad, keď exponent nie je číselná hodnota, ale výraz typu:

a ; 3*a ; 3*a+1-4*b

V tomto prípade predikát addl kumuluje v databáze prostredníctvom faktu elem(1,H,L) informáciu o súčine členov so základom H , pričom L je zoznam príslušných sčítancov. Napr. po spracovaní výrazu

x^a*y^(b-1)*x^(3*a+5)*y*x^~4

budú v databáze dva fakty

elem(1,x,[a,3*a,5,~4]). a elem(1,y,[b,~1,1]).

Predikát append realizuje rozklad exponentu na jednotlivé činitele, zjednodušuje ich pomocou adt_rewr a pripája ich pomocou concat k výslednému zoznamu. Predikáty beg a end zabezpečia, že fakty elem ktoré sú pred pokusom o splnenie subcieľa adt_rewr v databáze nebudú pri splňovaní tohoto subcieľa pozmenené.

Oproti doterajším úvahám predpokladáme, že fakty elem majú tri argumenty. Vyžaduje to ovšem úpravu predikátu add . Táto úprava musí zabezpečiť aj spoluprácu s predikátom addl . Uvedená úprava poslúži aj k realizácii posledného bodu zadania príkladu:

```
add(H,N) :- retract(elem(1,H,L)), assertz(elem(1,H,[N|L])).
add(H,N) :- retract(elem(_,H,S)), sum(S,N,S1),
            (posit(S1), !, Z= + ; Z= -), assertz(elem(Z,H,S1)).
add(H,N) :- (posit(N), !, Z= + ; Z= -), assertz(elem(Z,H,N )).
```

V prípade číselných exponentov sa v prvom argumente objavuje znamienko čísla v treťom argumente. Keď aspoň jeden z exponentov pre príslušný základ je nenumerický, prvý argument je 1 a tretí je zoznamom príslušných činiteľov. Pred konečným spracovaním výrazu pomocou mul_end je ovšem nutné pretransformovať fakty typu 1 na fakty typu +/- . Prvé pravidlo definície mul_comp sa preto zmení na tvar:

```
mul_comp([],S,V) :- pow_end, mul_end(S,V).

pow_end :- retract(elem(1,A,L)), beg, adt_comp(L,0,O), end,
           (O= ¯X, !, Z= - ; Z= +),
           assertz(elem(Z,A,O)), pow_end.
pow_end.
```

C./ Požadované poradie operátorov (najprv sčítanie a násobenie a až potom odčítanie a delenie) dosiahneme úpravou predikátov mul_end , adt_end , ea , em tak, aby spracovali najprv fakty typu elem(+, , ) :

```
adt_end(0,W) :- ea(X), adt_end( X, W).
adt_end(S,W) :- ea(X), adt_end(S+X,W).
adt_end(S,S).

ea(X) :- retract(elem(+,A,N)),
         (N=0, !, ea(X) ; N=1, !, X=A ; X=N*A).
ea(X) :- retract(elem(-,A,N)),
         (N= ¯1, !, X = ¯A ; X=N*A).

mul_end(0,0) :- retractall(elem(_,_,_)).
mul_end(S,W) :- em(X), mul_end(X,S,W).
mul_end(S,S).

mul_end(A, S, W ) :- em(X), mul_end(A*X,S,W).
mul_end(A, 1, A ).
```

```
mul_end(A,¯1,¯A ).
mul_end(A, S,S*A).

em(X) :- retract(elem(+,A,N)),
        (N=0, !, em(X) ; N=1, !, X=A ; X=A^N).
em(X) :- retract(elem(-,A,N)),  X=A^N.
```

Test:

```
?- ?a*(a+a)+a^(b-1)*a.
2*a^2+a^b
yes

?- ?a^2/b*(3*a-2*b+c).
a^2*(3*a+c+ ¯2*b)*b^ ¯1
yes

?- ?a^2/(b/a).
a^2*(b*a^ ¯1)^ ¯1
yes

?- ?a^2/(4*a-a).
a^2*(3*a)^ ¯1
yes

?- ?a^2/(3*a-4*a)^2.
a^2* ¯a^ ¯2
yes
```

Prvý príklad ilustruje splnenie prvých dvoch bodov zadania, druhý príklad splnenie tretieho bodu (konečnú úpravu t.j. náhradu mocnín s exponentom ¯1 operátorom delenia a náhradu kombinácie operátorov ± ¯ operátorom odčítania vyriešime v príklade 6.14). Neúplnú úpravu v treťom príklade dokončí pravidlo

```
proc(^,(A*B)^N,AA*BB) :- simple(A^N,AA), simple(B^N,BB).
```

ktoré zaradíme ako prvé do definície predikátu proc(^, , ) . Nedokončenú úpravu v štvrtom a piatom príklade vyriešime doplnením pravidiel

```
trans( A^N,     AA      ) :- pow(A,N,AA).        /* číselné A */
trans(¯X^N,     X^N     ) :- even(N).            /* párne N   */
trans(¯X^N,     ¯(X^N)  ) :- odd(N).             /* nepárne N */
trans((A*X)^N, AA*X^N ) :- pow(A,N,AA).          /* číselné A */
trans((X*Y)^N, X^N*Y^N) :- integ(N).             /* celé N    */
```

pred ostatné klauzuly pre trans . Definícia predikátu integ bola uvedená v príklade 6.1 a ďalšie pomocné predikáty sú definované nasledovne:

```
even( X) :- integer(X), 0 is X mod 2.
even(¯X) :- even(X).

odd( X) :- integer(X), 1 is X mod 2.
odd(¯X) :- odd(X).
```

Príklad 6.14.

Navrhnite predikáty pre konečnú úpravu aritmetických výrazov, pri ktorej je kombinácia operátorov + ¯ nahradená operátorom odčítania, násobenie zápornou mocninou je nahradené delením, mocnina s exponentom 0:50 je nahradená funktorom sqrt a "podiel" funktorov sin a cos je nahradený funktorom tan Návod: Navrhnite predikát clean , ktorý postupne rozoberá spracovávaný výraz a "čistí" jeho zložky a potom postupne zase skladá konečný tvar aritmetického výrazu (analogicky s predikátom remove v príklade 6.12). Vlastné pravidlá úpravy definujte pomocou predikátu change (analógia rem z príkladu 6.12). Tam kde je potrebné v jednom kroku realizovať náhradu pomocou operátora delenia a funktora sqrt použite ďalší pomocný predikát chsq .

Riešenie:

Doplníme definíciu predikátu 2 , z príkladu 6.12 na tvar:

```
? X :- remove(X,Y), simple(Y,Z), clean(Z,V), write(V).
```

a nadefinujeme nové predikáty

```
clean( X,      X) :- atom(X) ; integer(X).
clean(X*Y^ ¯N,W) :- clean(X,U), clean(Y^N,V), change(U/V,W).
clean( X,      Y) :- X=..[Op,AI], clean(AI,AO),
```

```
                        V=..[Op,AO], change(V,Y).
clean( X,       Y) :- X=..[Op,AI1,AI2], clean(AI1,AO1),
                                        clean(AI2,AO2),
                        V=..[Op,AO1,AO2], change(V,Y).

change( X+ ¯Y,              X-Y       ).
change( X+ ¯N*Y,            X-N*Y     ).
change(¯X+Y,                Y-X       ).
change( X^1,                X         ).
change( X^ ¯1,              1/X       ).
change( X^ ¯N,              1/Y       ) :- chsq(X^N,Y).
change( sin(X)/cos(X),      tan(X)    ).
change( sin(X)^N/cos(X)^N,  tan(X)^N  ).
change( cos(X)/sin(X),      cotan(X)  ).
change( X,                  Y         ) :- chsq(X,Y).

chsq(X^0:50,sqrt(X)).
chsq(X^N:50,sqrt(X)^M) :- integer(N), M is 2*N+1.
chsq(X,X).
```

Druhé pravidlo v definicii clean je potrebné kvôli tomu, aby sa výraz typu a*b^¯2 upravil na a/b^2 a nie na a*(1/b^2) ako by tomu bolo bez tohoto pravidla.

## 6.4. SYMBOLICKÉ DERIVOVANIE

### Príklad 6.15.

Navrhnite predikát realizujúci symbolické derivovanie zadanej funkcie podľa jednej premennej.

Riešenie:

Základom programu bude predikát deriv , ktorého prvým argumentom je derivovaná funkcia; v druhom argumente je "premenná", podľa ktorej sa derivuje a treti argument je hľadaná derivácia. Podobne ako v kapitole 6.3 musíme dôsledne rozlišovať medzi "premennou" funkcie (v PROLOGu je reprezentovaná atómom) a medzi prologovskou premennou. Základné predpisy pre derivovanie zapíšeme pomocou prologovských pravidiel:

a./ derivácia "premennej" podľa nej samej

```
deriv(X,X,1) :- !.
```

b./ derivácia konštanty

```
deriv(T,X,0) :- atom(T) ; integer(T).
```

c./ derivácia súčtu dvoch funkcií

```
deriv(U+V,X,A+B) :- deriv(U,X,A), deriv(V,X,B).
```

d./ derivácia záporne vzatej funkcie

```
deriv(¯T,X,¯R) :- deriv(T,X,R).
```

e./ derivácia funkcie, násobenej konštantou

```
deriv(K*U,X,V*K) :- number(K), deriv(U,X,V), !.
deriv(U*K,X,V*K) :- number(K), deriv(U,X,V), !.
```

f./ derivácia súčinu dvoch funkcií

```
deriv(U*V,X,U*B+V*A) :- deriv(U,X,A), deriv(V,X,B).
```

g./ derivácia celočíselnej mocniny funkcie

```
deriv(U^K,X,U^(K+ ¯1)*W*K) :- number(K),
                              deriv(U,X,W), !.
```

h./ derivácia funkcie, umocnenej na inú funkciu

```
deriv(U^V,X,log(U)*U^V*Z+U^(V+ ¯1)*V*W) :-
                              deriv(U,X,W), deriv(V,X,Z).
```

K uvedeným pravidlám pripojíme ďalšie, ktoré definujú derivácie niektorých elementárnych funkcií:

```
deriv(  exp(T),X,  exp(T)*R) :-                deriv(T,X,R).
deriv(  log(T),X,  T^(¯1)*R) :-                deriv(T,X,R).
deriv(  sin(T),X,  cos(T)*R) :-                deriv(T,X,R).
deriv(  cos(T),X, ¯sin(T)*R) :-                deriv(T,X,R).
deriv( sinh(T),X, cosh(T)*R) :-                deriv(T,X,R).
deriv( cosh(T),X, sinh(T)*R) :-                deriv(T,X,R).
deriv( atan(T),X, (1+T^2)^(¯1)*R) :-           deriv(T,X,R).
deriv( asin(T),X, (1+ ¯(T^2))^(¯0:50)*R) :- deriv(T,X,R).
deriv(  tan(X),X,W) :-       deriv(sin(T)*cos(T)^ ¯1,X,W).
```

Test:

```
?- deriv(log(sin(x)), x, D).
D = sin(x)^ ¯1*(cos(x)*1)
yes

?- deriv(log(x^4), x, D).
D = (x^4)^ ¯1*(x^(4+ ¯1)*1*4)
yes
```

Ako je vidieť, výsledky získané aplikáciou navrhnutých pravidiel sú veľmi neprehľadné a je potrebné ich ďalej upraviť. Pri zápise "derivačných" pravidiel sme rešpektovali niektoré zásady, ktoré zjednodušia túto úpravu:

- keď sa vo výsledku objavil súčin, potom sme členy, ktoré sú, alebo môžu byť konštantami, umiestnili na koniec

- namiesto odpočítania sme použili pripočítanie člena so záporným znamienkom (unárnym mínus ¯ )

- namiesto delenia sme použili násobenie členom, umocneným na mínus prvú

Pritom sme predpokladali, že derivovaná funkcia neobsahuje delenie, odčítanie (operátor "binárne mínus") a odmocninu (funktor sqrt ). Sledovali sme tým jednoduchosť uvedených "derivačných" pravidiel a tiež aj jednoduchosť ďalšej úpravy zderivovanej funkcie (viď kapitolu 6.3).

Príklad 6.16.

Navrhnite predikát, ktorý najprv upraví derivovanú funkciu podľa vyššie uvedených zásad, potom uskutoční vlastnú deriváciu a nakoniec túto deriváciu zjednoduší.

Riešenie:

Predspracovanie derivovanej funkcie (odstránenie operátorov delenia a odčítania, ako aj a funktora sqrt sa realizuje predikátom remove (viď príklad 6.12). Zjednodušenie zderivovanej funkcie má na starosti predikát simple (viď príklad 6.11) a jej úpravu na konečný tvar uskutoční predikát clean (viď príklad 6.14):

```
d(F,Var,DD) :- remove(F,F1), deriv(F1,Var,D),
               simple(D,SD), clean(SD,DD).
```

Aby sme získali predstavu o charaktere funkcií, ktoré vznikajú v jednotlivých etapách spracovania zderivovanej funkcie, uvedieme niekoľko príkladov. Význam jednotlivých premenných je:

```
F  - derivovaná funkcia
D  -  derivácia
SD - zjednodušená derivácia
DD - upravená derivácia

F  >  log(sin(x))
D  >  sin(x)^ ¯1*(cos(x)*1)
SD >  cos(x)*sin(x)^ ¯1
DD >  cotan(x)

F  >  log(x^4)
D  >  (x^4)^ ¯1*(x^(4+ ¯1)*1*4)
SD >  4*x^ ¯1
DD >  4/x

F  >  sqrt(1-x^2)
D  >  (1+ ¯1*x^2)^(0:50+ ¯1)*(0+x^(2+ ¯1)*1*2* ¯1)*0:50
SD >  ¯(x*(1+ ¯(x^2))^ ¯0:50)
DD >  ¯(x/sqrt(1-x^2))
```

Z uvedených príkladov je zrejmá potreba spracovávania pomerne zložitých štruktúr (najmä "medzivýsledkov"), čo je vzhľadom na rekurzívnu definíciu príslušných predikátov veľmi náročné na pamäť. Priebežné "čistenie" lokálneho zásobníka možno dosiahnuť symbolmi rezu na konci všetkých pravidiel, definujúcich predikáty remove, simple a clean . Ďalšiu úsporu pamäti "čistením" globálneho zásobníka zabezpečí úprava definície trojargumentového predikátu d s prenosom "medzivýsledkov" cez databázu:

```
d(F,_,_) :-                    remove(F,R),  asserta(a(R)), fail.
d(_,V,_) :- retract(a(R)), deriv(R,V,D), asserta(a(D)), fail.
d(_,_,_) :- retract(a(D)), simple(D,S),  asserta(a(S)), fail.
d(_,_,D) :- retract(a(S)), clean(S,D).
```

Pri tejto úprave je bezpodmienečne nutné vylúčiť možnosť opätovného splnenia predikátov remove, deriv, simple a clean , čo sa dosiahne vyššie uvedeným použitím symbolov rezu v príslušných pravidlách.

Kvôli prehľadnejšej komunikácii so systémom je účelné deklarovať operátory:

```
?- op(200, fx, d), op(202, xfx, %).
```

a definovať ich pravidlami:

```
d F % d Var :- atom(Var), d(F,Var,D), write(D).
d F         :- d F % d x.
```

Druhé pravidlo slúži pre ošetrenie najčastejších prípadov, keď sa zadaná funkcia derivuje podľa "premennej" x .

Test:

```
?- d sqrt(log(x)).
0:50*(1/sqrt(log(x))/x)
yes

?- d a^x*x^a.
a^(1+x)*x^(a-1)+log(a)*x^a*a^x
yes
```

```
?- d x*asin(log(x)).
1/sqrt(1-log(x)^2)+asin(log(x))
yes

?- d atan(a^2) % d a.
2*(a/(1+a^4))
yes
```

Príklad 6.17.

Upravte predikáty z predošlých príkladov tak, aby umožnili aj derivovanie podľa viacerých premenných. Predpokladajme, že premenné, podľa ktorých sa má derivovať, sú zadané v tvare zoznamu.

Riešenie:

Uvedenú úlohu možno riešiť prepisom pravidla, definujúceho predikát derivovania na rekurzívny tvar:

```
d D % d       []          :- write(D).
d F % d [Var|VarList] :- d(F,Var,D), d D % d VarList, !.
```

Uvedené pravidlá musia byť pred pravidlami pre definíciu tohoto predikátu z príkladu 6.16 ([] je totiž tiež atóm !!).

Pomocou takto navrhnutého predikátu možno realizovať aj N-té derivácie podľa jednej premennej - požadovanú premennú uvedieme v zozname N-krát.

Test:

```
?- d x/4^x.
1/4^x-x*log(4)/4^x
yes

?- d log(sin(z)) % d z.
cotan(z)
yes
```

```
?- d 1/(1-x) % d [x,x,x,x].
24/(1-x)^5
yes

?- d sqrt(x^2+y^2)^3 % d [x,y].
3*(x*y/sqrt(x^2+y^2))
yes
```

Najmä pri deriváciách vyšších rádov sa občas prejavia niektoré neriešené otázky úpravy aritmetických výrazov. Predovšetkým sa jedná o vytknutie spoločných členov pred zátvorku, uvedenie na spoločného menovateľa a následné krátenie celými výrazmi, atď. V prípade, že dôjde k vyčerpaniu dostupnej pamäte, je možné riešiť derivácie vyšších rádov nerekurzívne, s ukladaním medzivýsledkov do databázy.

# 7. METÓDA GENERUJ & TESTUJ

Mnohé úlohy sa riešia v dvoch etapách: najprv sa vygeneruje kandidát na riešenie a potom sa otestuje. V PROLOGu môžeme naznačený postup vyjadriť predikátom

```
riesenie(X) :- generuj(X), testuj(X).
```

Pritom interpret jazyka zabezpečuje pomocou mechanizmu navracania vlastné hľadanie riešenia. Postupuje pritom "naslepo" bez možnosti detekovať príčinu neúspechu testu a usmerniť generovanie ďalších alternatív tak, aby sa vopred vylúčili "nepersperktívni" kandidáti.

V ďalšom ukážeme na niekoľkých typických príkladoch ako je možné účelným skĺbením oboch etáp zefektívniť tento postup. Obvykle totiž riešením je zložitejší objekt, generovaný vo viacerých krokoch. Nebudeme s testovaním čakať až do chvíle, keď celý kandidát na riešenie je vygenerovaný, ale testujeme dielčiu časť kandidáta po každom kroku generovania.

<u>Príklad 7.1.</u>

Navrhnite program na riešenie algebrogramu

```
   S E N D
 + M O R E
 ----------
 M O N E Y
```

t.j nahraďte každé písmeno jednou číslicou tak, aby naznačený súčet bol správny (čísla nesmú začínať nevýznamnou nulou).

Riešenie:

Najjednoduchši variant bude generovať postupne všetky možné náhrady a vzápätí ich testovať:

```
algebrogram1([S,E,N,D],[M,O,R,E],[M,O,N,E,Y]) :-
    sub_permut([0,1,2,3,4,5,6,7,8,9,0],[S,E,N,D,M,O,R,Y]),
    M > 0, S > 0,
```

```
sucet_zoz([0,S,E,N,D],[0,M,O,R,E],[M,O,N,E,Y],0).

sub_permut(_,[]) :- !.
sub_permut(Vzor,[X|Zbytok]) :- vyber(X,Vzor,VzorBezX),
                               sub_permut(VzorBezX,Zbytok).

vyber(P, [P|T],    T     ).
vyber(P, [H|T], [H|TbezP]) :- vyber(P,T,TbezP).

sucet_zoz([],[],[],0).
sucet_zoz([H1|T1],[H2|T2],[HV|TV],PD) :-
                    sucet_zoz(T1,T2,TV,PS),
                    Suma is H1+H2+PS,
                    rozklad(Suma,HV,PD).

rozklad(S,C,PD) :- C is S mod 10, PD is S/10.
```

Je možné definovať predikát pre rozklad cifier aj dvoma inými spôsobmi:

```
rozklad1(S,S,0) :- S <  10.
rozklad1(S,C,1) :- S >= 10, C is S - 10.

rozklad2(S,C,PD) :- S <  10, C = S,       PD = 0;
                    S >= 10, C is S - 10, PD = 1.
```

Predikát <u>rozklad</u> pracuje asi o 30% rýchlejšie než <u>rozklad1</u> a takmer trikrát rýchlejšie, ako predikát <u>rozklad2</u> (tieto údaje platia pre PROLOG-80). Uvedené porovnanie má všeobecnejšiu platnosť: je výhodné realizovať čo najviac unifikácií priamo v hlave pravidla, je výhodné definovať predikát jediným pravidlom - ovšem bez disjunkcie v jej tele (najmä keď argumenty operátora "bodkočiarka" sú konjunkcie).

Predikát <u>sub_permut</u> generuje všetky možné substitúcie písmen daného algebrogramu za desiatkové cifry. Subcieľ <u>vyber</u> realizuje priraďovanie jednotlivých číslic z postupne sa zmenšujúceho zoznamu nepoužitých cifier príslušným písmenám (premenným).

Podmienky M > 0 a S > 0 vylučujú prípady, keď číslo začína nepodstatnou nulou. Predikát sucet zoz realizuje sčítanie dvoch čísiel, reprezentovaných zoznamom ich cifier (zoznamy cifier sčítancov v prípade potreby doplníme nevýznamnými nulami tak, aby boli rovnako dlhé, ako zoznam cifier výsledku). Štvrtý argument je prenos (pri volaní, t.j. v najvyššom ráde je prenos doľava PD nulový a v najnižšom ráde, t.j. v prvej definičnej klauzule je zase prenos sprava PS nulový):

```
                        <--          0 <--
...                 .        ...    []
                   H2               []
                  ------------------------
0 <--       PD <-- HV               []
```

Uvedený program pracuje veľmi neefektívne, nakoľko testovanie sa realizuje až po vygenerovaní všetkých cifier algebrogramu.

Príklad 7.2.

Upravte definíciu predikátu algebrogram1 tak, aby sa testy realizovali priebežne, vždy po vygenerovaní dvojice cifier jedného rádu. Neúspešnosť vyšetrovaného kandidáta na riešenie sa tak zistí skôr a vyvolá sa ihneď navracanie.

Riešenie:

```
algebrogram2([S,E,N,D],[M,O,R,E],[M,O,N,E,Y]) :-
                sucet(0 ,D,E,Y,P1),
                sucet(P1,N,R,E,P2),
                sucet(P2,E,O,N,P3),
                sucet(P3,S,M,O, M), M > 0, S > 0,
                dif([S,E,N,D,M,O,R,Y]).

sucet(PS,H1,H2,HV,PD) :- cifra(H1), cifra(H2),
                         Sum is H1 + H2 + PS,
                         rozklad(Sum,HV,PD).

dif([]).
dif([H|T]) :- not member(H,T), dif(T).

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

Predikát sucet generuje dvojicu spočítavaných cifier H1, H2 uvažovaného rádu, k ich súčtu pridá prenos sprava PS , zistí cifru HV výsledku a prenos vľavo PD . Predikát dif otestuje nakoniec, či sa tá istá cifra nepridelila viacerým písmenám.

Uvedený spôsob generovania a testovania cifier na nerovnosť je pomerne neefektívny. Nájdenie jediného možného riešenia

```
  9 5 6 7
+ 1 0 8 5
----------
 1 0 6 5 2
```

trvá až pol hodiny. Výrazné (asi 7 násobné) zrýchlenie sa dosiahne zaradením testov dif za každý subcieľ sucet :

```
algebrogram3([S,E,N,D],[M,O,R,E],[M,O,N,E,Y]) :-
            sucet(0 ,D,E,Y,P1), dif([D,E,Y]),
            sucet(P1,N,R,E,P2), dif([D,E,Y,N,R]),
            sucet(P2,E,O,N,P3), dif([D,E,Y,N,R,O]),
            sucet(P3,S,M,O, M), M > 0, S > 0,
            dif([S,E,N,D,M,O,R,Y]).
```

Elegantnejšie riešenie je uvedené v príklade 7.3.

Príklad 7.3.

Navrhnite úpravu predikátov z príkladu 7.2 tak, aby sa súčty jednotlivých rádov realizovali v cykle a aby generovanie cifier automaticky vylúčilo možnosť náhrady rôznych písmen tou istou číslicou (analogicky s príkladom 7.1).

Riešenie:

```
algebrogram4([S,E,N,D],[M,O,R,E],[M,O,N,E,Y]) :-
   sucet_zoz([0,S,E,N,D],[0,M,O,R,E],[M,O,N,E,Y],0,
            [0,1,2,3,4,5,6,7,8,9],_), M > 0, S > 0.

sucet_zoz([],[],[],0,C,C).
sucet_zoz([H1|T1],[H2|T2],[HV|TV],PD,ZPC,ZNC):-
```

```
                    sucet_zoz(T1,T2,TV,PS,ZPC,ZC),
                    sucet_cifier(H1,H2,HV,PS,PD,ZC,ZNC).

sucet_cifier(H1,H2,HV,PS,PD,ZPC,ZNC) :- zvol(H1,ZPC, Z1),
                                        zvol(H2, Z1, Z2),
                                        zvol(HV, Z2,ZNC),
                                        Suma is H1+H2+PS,
                                        sucet(Suma,HV,PD).

zvol(X,L,L) :- nonvar(X), !.
zvol(X,L,LbezX) :- vyber(X,L,LbezX).
```

V definícii predikátu sucet_zoz pribudli posedné dva argumenty

```
sucet_zoz(ZoznamCifierPrvehoScitanca,
          ZoznamCifierDruhehoScitanca,
          ZoznamCifierVysledku,
          PrenosDolava,
          ZoznamPouzitelnychCifier,
          ZoznamNepouzitychCifier).
```

Tri subciele zvol generujú tri cifry práve vyšetrovaného rádu, pričom sa tieto cifry odoberajú zo zoznamu nepoužitých cifier. Keď ovšem niektorá z cifier bola už v niektorom z predošlých rádov vygenerovaná (napr. E sa generuje v najnižšom ráde a v ďalších dvoch sa opäť používa), potom ostane tento zoznam nezmenený, viď prvá klauzula pre zvol. Vylúčenie čísiel, začínajúcich nepodstatnými nulami (podmienky M > 0 a S > 0 ) sa robí až po vygenerovaní celého riešenia, takže neprispeje k zefektívneniu výpočtu. V porovnaní s predikátom algebrogram2 sa výpočet zrýchli 1.5 krát.

Príklad 7.4.

Prelínanie generovania a testovania realizujte pomocou "zmrazovania" cieľov. Jedná sa o špeciálny prostriedok riadenia výpočtu, ktorý poskytujú niektoré implementácie PROLOGu (v PROLOG-80 realizuje túto funkciu predikát constrain , viď prílohu 1).

Riešenie:

```
algebrogram5([S,E,N,D],[M,O,R,E],[M,O,N,E,Y]) :-
    constrain(S,S>0), constrain(M,M>0),
    sucet_zoz([0,S,E,N,D],[0,M,O,R,E],[M,O,N,E,Y],0),
    sub_permut([1,2,3,4,5,6,7,8,9,0],[Y,D,E,N,R,O,S,M]).

sucet_zoz([],[],[],0).
sucet_zoz([H1|T1],[H2|T2],[HV|TV],PD):-
                    sucet_zoz(T1,T2,TV,PS),
                    sucet(H1,H2,HV,PS,PD).

sucet(A,B,P,P, 0 ) :- nonvar(B), !.
sucet(A,B,C,PS,PD) :- constrain(B, sucet0(A,B,C,PS,PD)).

sucet0(A,B,C,PS,PD) :- S is A+B+PS, C is S mod 10, PD is S/10.
```

Predikát sub permut slúži v tomto prípade ako generátor, pričom poradie generovaných premenných bolo potrebné zvoliť tak, aby vo chvíli "rozmrazenia" príslušného subcieľa sucet0 boli jeho prvé tri argumenty viazané (viazanie štvrtého argumentu - prenosu sprava - je zabezpečené poradím premenných v druhom argumente subcieľa sub permut). Podmienky M > 0 a S > 0 sme "zmrazili", takže systém ich preveruje okamžite po vygenerovaní cifier M , resp. S .

Tento postup už nie je úplne univerzálny, nakoľko v iných algebrogramoch by sa mohlo stať pri postupnom "rozmrazovaní" subcieľov sčítavania cifier jednotlivých rádov (od najnižšieho začínajúc), že sa naviaže niektorá premenná, na ktorú je "zmrazený" subcieľ súčtu cifier vyššieho rádu pre ktorý ešte neboli naviazané všetky sčítance predikátu is a program by kvôli tomu havaroval.

Príklad 7.5.

Navrhnite program pre riešenie nasledujúcej úlohy: Ivan, Peter a Jožo tipujú víťaza turnaja štyroch mužstiev - Sparta, Inter, Slovan, Bohemka. Ich tvrdenia sú:

Ivan : "Vyhrá Inter alebo Bohemka."
Peter : "Inter nevyhrá."
Jožo : "Nevyhrá ani Sparta, ani Bohemka."

Ktoré mužstvo vyhralo, keď sa ukázalo, že iba jeden z tipujúcich mal pravdu ?

Riešenie:

```
turnaj(Tipol,V) :-
     vyber(Tipol,[ivan,peter,jozo],Ostatni), vitaz(V),
     vyrok(Tipol,V), not(( member(N,Ostatni), vyrok(N,V) )).

vitaz(sparta). vitaz(inter). vitaz(slovan). vitaz(bohemka).

vyrok(ivan, V) :- V = inter ; V = bohemka.
vyrok(peter,V) :- V \= inter.
vyrok(jozo, V) :- V \= sparta, V \= bohemka.
```

Predikátom vyber generujeme úspešného tipujúceho (v treťom argumente je zoznam tých, čo sa mýlili) a predikátom vitaz generujeme možných víťazov. Potom testujeme platnosť výroku úspešného tipujúceho a nakoniec preveríme neplatnosť výrokov tých ostatných.

Test:

```
?- turnaj(Prorok,Vitaz).
Prorok = peter, Vitaz = sparta ;
no
```

Príklad 7.6.

Upravte definíciu predikátu turnaj pre prípad, že by sa výroky týkali aj mužstiev na ostatných miestach výslednej tabuľky:

Ivan : "Vyhrá Inter alebo Slovan a Sparta bude na jednom z posledných dvoch miest."
Peter : "Inter bude druhý a Sparta nebude tretia."
Jožo : "Na prvých dvoch miestach budú Inter a Slovan."

Riešenie:

```
turnaj1(Tipol,Tab) :-
                vyber(Tipol,[ivan,peter,jozo],Ostatni),
                permut([sparta,inter,slovan,bohemka],Tab),
                vyrok(Tipol,Tab),
                not(( member(N,Ostatni), vyrok(N,Tab) )).

vyrok(ivan,     [V,_|T]    )  :- (V = inter ; V = slovan),
                                  member(sparta,T).
vyrok(peter,[_,inter,T,_])  :- T \= sparta.
vyrok(jozo,     [P,D|_])    :- permut([inter,slovan],[P,D]).

permut([],[])    :- !.
permut(Z,[H|T]) :- vyber(H,Z,W), permut(W,T).
```

Predikát <u>permut</u> vytvára v druhom argumente pri navracaní všetky možné permutácie prvkov zoznamu z prvého argumentu. Druhým argumentom predikátu <u>vyrok</u> je v tomto prípade zoznam, reprezentujúci výslednú tabuľku turnaja.

Test:

```
?- turnaj1(Prorok,Tabulka).
Prorok = ivan, Tabulka = [inter,bohemka,sparta,slovan] ;
Prorok = ivan, Tabulka = [inter,bohemka,slovan,sparta] ;
Prorok = ivan, Tabulka = [slovan,bohemka,sparta,inter] ;
Prorok = ivan, Tabulka = [slovan,bohemka,inter,sparta] ;
Prorok = jozo, Tabulka = [sparta,inter,bohemka,slovan] ;
no
```

# 8. RIEŠENIE ÚLOH V STAVOVOM PRIESTORE

## 8.1. STAVOVÝ PRIESTOR A JEHO REPREZENTÁCIA GRAFOM

Jadrom každej intelektuálnej úlohy je účelný rozhodovací proces - riešenie úlohy (niekedy, najmä v robotike sa pre tento proces používa označenie plánovanie chovania ). Na základe vhodného modelu sveta ( svetom sa rozumie zvyčajne úzko špecializované prostredie, v ktorom sa hľadá riešenie úlohy) je potrebné usporiadať elementárne akcie do postupnosti tak, aby sa zo zadaného východzieho stavu dosiahol cieľový stav. Jednou z foriem reprezentácie modelu sveta je stavový priestor , definovaný ako dvojica:

$$P = ( S , F )$$

kde $S = \{si\}$ ... konečná množina stavov
$F = \{fi\}$ ... konečná množina akcií (akcia reprezentuje povolený prechod z jedného stavu do druhého)

a úloha v stavovom priestore P je potom dvojica

$$U = ( s0 , G )$$

kde $s0$ ... počiatočný stav ( $s0$ je prvkom množiny $S$ )
$G$ ... množina cieľových stavov ( $G$ je podmnožinou $S$ )

Riešenie úlohy je potom reprezentované lineárnym plánom , tvoreným postupnosťou akcií:

$$RF = (f1, f2, \ldots ,fn)$$

ku ktorej existuje postupnosť stavov

$$RS = (s1, s2, \ldots ,sn)$$

taká, že platí:

$$s1 = f1(s0)$$
$$s2 = f2(s1) = f2(f1(s0))$$
$$\ldots$$
$$sn = fn(sn-1) = fn( \ldots f2(f1(s0)) \ldots )$$

pričom sn je prvkom množiny G .

Názornú predstavu o štruktúre stavového priestoru možno získať pomocou grafu, ktorého uzly reprezentujú jednotlivé stavy a ktorého orientované hrany predstavujú príslušné akcie. Hľadanie riešenia úlohy je potom ekvivalentné hľadaniu cesty v grafe od uzla, reprezentujúceho východzí stav s0 do niektorého z uzlov, reprezentujúcich cieľové stavy sn (sú prvkami množiny G ).

## 8.2. HĽADANIE CESTY V GRAFE - PREHĽADÁVANIE DO HĽBKY

Zostavenie vhodného modelu sveta pre riešenú úlohu je silne závislé na konkrétnej úlohe a často býva nejednoznačné. Touto etapou riešenia sa budeme podrobne zaoberať v kapitolách 8.3-8.6 v rámci popisu jednotlivých špeciálnych úloh. Stavy budú v jednotlivých príkladoch reprezentované rôznymi objektami jazyka PROLOG (od čísiel až po zoznamy zoznamov). Zatiaľ predpokladajme, že v databáze je uložený popis grafu stavového priestoru vo forme faktov typu:

```
oh(From,To).
```

reprezentujúcich orientovanú hranu smerujúcu z uzla From do uzla To (orientovaná hrana reprezentuje operátor, ktorému nemusí prislúchať inverzný operátor, t.j. prechod zo stavu To do stavu From ).

Priklad 8.1.

Navrhnite predikát pre vyhľadanie cesty v grafe zadanom vyššie uvedeným spôsobom.

Riešenie.

Jedná sa o typickú rekurzívnu úlohu, analogickú hľadaniu potomka v príklade 2.1:

```
depth1(S0,SN) :- oh(S0,SN).
depth1(S0,SN) :- oh(S0, X), depth1(X,SN).
```

Toto riešenie má dva nedostatky:

- keď existuje v grafe cyklus, potom sa môžeme dostať do nekonečnej slučky (závisí to na tom, v akom poradi sú v databáze uložené fakty, opisujúce graf)

- výsledkom riešenia je iba prologovské yes (cesta existuje), alebo no (cesta neexistuje).

Oba problémy možno riešiť súčasne sledovaním uzlov "dielčej cesty", ktorú postupne rekurzívne vyhľadávame. Budú k tomu slúžiť fakty uzol(X) v databáze. Pre ich ukladanie musíme navrhnúť špeciálny predikát, nakoľko štandardné predikáty asserta a assertz pri navracani neobnovia pôvodný stav databázy. Súčasne môžeme sledovať, či nechceme zaradiť do databázy uzol, ktorý sa v doteraz nájdenej ceste už raz vyskytol a zamedziť tak zacykleniu programu:

```
store(X) :-  node(X), !, fail.
store(X) :-  assertz(node(X)).
store(X) :-  retract(node(X)), fail.
```

Vlastný predikát hľadania cesty musíme volať z nadradeného cieľa, ktorý slúži na čistenie databázy a na uloženie východzieho uzla grafu:

```
depth2(S0,SN) :- retractall(node(_)), store(S0), depth0(S0,SN).

depth0(S0,SN) :- oh(S0,SN), store(SN), path.
depth0(S0,SN) :- oh(S0, X), store( X), depth0(X,SN).
```

Predikát path realizuje výpis všetkých uzlov, nachádzajúcich sa na nájdenej ceste:

```
path :- node(X), write(X),tab(1), fail.
path.
```

Keď existuje viacej ciest v grafe medzi zadanými uzlami, nájdeme iba jednu z nich (poradie faktov <u>oh</u> v databáze určí, ktorá z ciest bude nájdená). Vyhľadanie všetkých ciest sa dosiahne úpravou:

```
depth3(S0,SN) :- retractall(node(_)), store(S0), depth0(S0,SN),
                 nl, fail.
```

V prvom pravidle definície predikátu <u>path</u> bolo vlastne účelnejšie použiť ako prvý subcieľ <u>retract(node(X))</u> , čím by sa súčasne aj čistila databáza. Keď ale použijeme ako hlavný cieľ <u>depth3</u> , musíme zachovať aj po výpise ten stav databázy, ktorý bol vo chvíli nájdenia príslušnej cesty. Jedine tak prebehne korektne navracanie, ktorým začína hľadanie ďalšej cesty.

<u>Príklad 8.2.</u>

Navrhnite predikát pre hľadanie cesty v grafe, ktorý bude budovať už nájdenú dielčiu cestu ako zoznam v jednom zo svojich argumentov.

Riešenie:

Jedná sa vlastne o akumulačný cyklus (kapitola 4.6), pre realizáciu ktorého navrhneme pomocný predikát <u>collect</u> . Hlavný cieľ bude nasledovný:

```
depth(S0,SN,Path) :- collect([S0],SN,Path).
```

Prvý argument predikátu <u>collect</u> slúži k postupnému vytváraniu zoznamov uzlov ležiacich na ceste z uzla <u>S0</u> po uzol <u>SN</u> . Po dosiahnutí cieľa sa nájdená cesta prenesie do tretieho argumentu.

Jadrom programu je rekurzivne pravidlo, realizujúce akumulačný cyklus, v rámci ktorého sa postupne doplňuje vytváraný zoznam o nové uzly. Situáciu schématicky znázorňuje obr. 8.1.

```
                                   -- oh ->
+--+    +--+          +--+    +---+    +---+          +--+
|S0|-->-|  |-->...->-|  |-->-|Old|-->-|New|-->...->-|SN|
+--+    +--+          +--+    +---+    +---+          +--+

|------------T----------->|-Old-|
```

obr. 8.1

Čiastkový zoznam (reprezentujúci už nájdený úsek cesty) je tvorený hlavou Old a telom T . Keď sa zisti, že existuje hrana vedúca od uzla Old k uzlu New , potom sa nový uzol pripojí k čiastkovému zoznamu:

```
collect([Old|T],SN,P) :- oh(Old,New), collect([New,Old|T],SN,P).
```

Keď v grafe existujú cykly, predikát sa môže dostať do nekonečnej slučky. Tomu je možné zabrániť vložením subcieľa

```
not member(New,[Old|T])
```

medzi subciele oh a collect v tele rekurzívneho pravidla (predikát member testuje, či prvý argument je prvkom zoznamu v druhom argumente, viď príklad 2.6).

Nakoniec je potrebné ošetriť hraničný prípad rekurzie ( New=SN , t.j. hľadanie cesty bolo úspešne ukončené). V tomto kroku sa prenesie zoznam uzlov cesty postupne vytvorený v predošlých krokoch (prvý argument) do výsledku (tretí argument). Nakoľko sa čiastkový zoznam budoval od uzla S0 k uzlu SN , cesta je v ňom uložená v opačnom poradí. V hraničnom pravidle sa preto použije predikát rev (viď príklad 2.9) na obrátenie poradia prvkov zoznamu. Program pre vyhľadanie cesty v grafe nadobudne tak konečnú podobu:

```
depth(S0,SN,Path)          :- collett([S0],SN,Path).

collect([SN|T],SN, Path) :- rev([SN|T],Path).
collect([Old|T],SN,Path) :- oh(Old,New),not member(New,[Old|T]),
                            collect([New,Old|T],SN,Path).
```

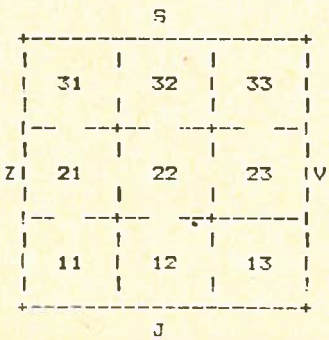
Keď cesta v grafe existuje, uvedený program ju nájde. V prípade že nás zaujímajú aj ďalšie cesty, vedúce v zadanom grafe z východzieho uzla S0 do uzla SN , je možné zadávaním bodkočiarky vyvolať navracanie a nájsť ďalšie alternatívy.

Uvedený postup predstavuje prehľadávanie grafu do hĺbky . Pri rozsiahlejších grafoch môže byť tento postup pomerne neefektívny. Často trvá veľmi dlho, než systém zistí, že sa dostal do slepej uličky. Keď používateľ má istú predstavu o možnej dĺžke cesty, alebo keď ho zaujímajú iba cesty, ktorých dĺžka neprekročí zvolenú hranicu L , potom je možné doplniť uvedený program o príslušný test a výrazne zvýšiť efektívnosť prehľadávania. Po naznačených zmenách bude mať program nasledujúci tvar:

```
depth(S0,SN,Path,L)          :- collect([S0],SN,Path,L).

collect([SN|T], SN,Path,_) :- rev([SN|T],Path).
collect([Old|T],SN,Path,L) :-
    oh(Old,New), not member(New,[Old|T]), length([Old|T],LL),
    ( LL>L, !, fail ; collect([New,Old|T],SN,Path,L) ).
```

## 8.3. HĽADANIE CESTY V BLUDISKU

Typickým príkladom stavového priestoru je bludisko (obr. 8.2), v ktorom uzly grafu reprezentujú miestnosti a hrany (v tomto prípade neorientované ! ) predstavujú prechody medzi nimi.




obr. 8.2

Príklad 8.3.

Navrhnite reprezentáciu stavov úlohy podľa obr. 8.2.

Riešenie:

Aby nebolo potrebné zadávať dvojicu faktov oh pre každý prechod, je vhodné zaviesť pomocné pravidlo:

```
oh(X,Y) :- h(X,Y) ; h(Y,X).
```

a zadať každý prechod jediným faktom h :

```
h(11,21).  h(21,31).  h(21,22).  h(31,32).  h(22,32).
h(12,22).  h(12,13).  h(22,23).  h(23,33).
```

Symetričnosť predikátu oh nemožno vyjadriť faktami typu

```
oh(11,21). oh(21,31).  ...  oh(23,33).
```

a pravidlom

```
oh(X,Y) :- oh(Y,X).
```

nakoľko by sa výpočet zacyklil pri neúspešnom pokuse o splnenie cieľa oh . Ani pri úspešnom pokuse nie je chovanie takto navrhnutého predikátu korektné, pri navracaní generuje stále to isté riešenie.

Keď hľadáme cestu zo stavu 11 do stavu 33 , potom jednotlivé riešenia nájdeme prostredníctvom nasledujúceho dialógu:

```
?-depth(11,33,P).
P = [11,21,31,32,22,23,33] ;
P = [11,21,22,23,33] ;
no
```

Príklad 8.4.

Navrhnite inú alternatívu popisu "sveta" zadanej úlohy, kde stavy budú popísané štruktúrami typu vertikal@horizontal a sú definované štyri akcie, reprezentujúce pohyb z danej miestnosti v smere príslušnej svetovej strany: S , V , Z , J .

Riešenie:

Deklarujme operátor op(10,xfx,@) . Každej akcii môžeme priradiť jedno pravidlo, v tele ktorého je test na aplikovateľnosť príslušnej akcie (existencia dverí z miestnosti I@J v zvolenom smere, viď obr. 8.2):

```
oh(I@J,K@J):-K is I+1,member(I@J,[1@1,2@1,1@2,2@2,2@3]). /* S */
oh(I@J,I@L):-L is J+1,member(I@J,[2@1,3@1,2@2,1@2]).     /* V */
oh(I@J,K@J):-K is I-1,member(I@J,[2@1,3@1,2@2,3@2,3@3]). /* J */
oh(I@J,I@L):-L is J-1,member(I@J,[2@2,3@2,1@3,2@3]).     /* Z */
```

a príslušný dialóg bude mať tvar:

```
?-depth(1@1,3@3,P).
P = [1@1,2@1,3@1,3@2,2@2,2@3,3@3] ;
P = [1@1,2@1,2@2,2@3,3@3] ;
no
```

Poradie faktov pre b resp. oh určuje v akom poradí PROLOG nájde jednotlivé riešenia.

## 8.4. PRELIEVANIE VODY V NÁDOBÁCH

### Príklad 8.5.

K dispozícii sú dve nádoby s objemom 2 a 5 litrov. Väčšia z nich je plná vody, menšia je prázdna. Úlohou je dosiahnuť stav, keď v menšej nádobe je presne 1 liter vody.

Riešenie:

Aj v tomto príklade by bolo možné zostaviť graf stavového priestoru (tvorí ho 9 uzlov a 18 hrán) a zapísať do databázy príslušné fakty. Účelnejším sa javí zapísať niekoľko pravidiel, vyjadrujúcich povolené akcie danej úlohy (v zadaní neboli explicitne formulované, ale budú zrejmé zo zápisu v PROLOGu). Tento postup dáva kompaktnejší program, ktorý je naviac univerzálny: po zmene konštánt udávajúcich objemy nádob je možné ho použiť aj na riešenie iných úloh tohoto typu

Zvoľme na popis stavu štruktúru s(V,M) , kde V a M udávajú objem vody (v litroch) vo veľkej a malej nádobe. Jednotlivé povolené akcie sú popísané nasledujúcimi skupinami pravidiel

a. úplné vyprázdnenie jednej nádoby do okolia

```
oh(s(V,M), s(V,0)).
oh(s(V,M), s(0,M)).
```

b. prelievanie z malej nádoby do veľkej

```
oh(s(V,M), s(W,0)) :- V+M < 5, W is V+M.
oh(s(V,M), s(5,N)) :- V+M >= 5, N is V+M-5.
```

prvá klauzula reprezentuje preliatie celého obsahu malej nádoby do veľkej - to je možné za podmienky, že celkový objem vody je menší ako objem veľkej nádoby; druhá klauzula rieši práve prípad, keď je táto podmienka nesplnená (veľká nádoba sa doleje doplna a zbytok ostáva v malej (tento prípad pri pôvodnom zadaní nemôže nastať, ale uvádzame ho kvôli tomu, aby navrhnuté predikáty boli všeobecné a riešili aj úlohy s inými "počiatočnými podmienkami")

c. prelievanie z veľkej nádoby do malej

```
oh(s(V,M), s(W,2)) :- V+M > 2, W is V+M-2.
oh(s(V,M), s(0,N)) :- V+M =< 2, N is V+M.
```

v prípade že celkový objem vody v obidvoch nádobách je väčší, ako objem malej nádoby (prvé pravidlo), potom bude malá nádoba po skončení tejto akcie plná (2 litre) a zbytok V+M-2 ostane vo veľkej nádobe; v prípade, že celkový objem vody v obidvoch nádobách je menší ako 2 litre, potom sa celý tento objem V+M dostane do malej nádoby a veľká ostane prázdna.

Konečné riešenie úlohy sa získa v dialógu:

```
?- depth(s(5,0),s(_,1),P).
P = [s(5,0),s(3,2),s(3,0),s(1,2),s(1,0),(0,1)] ;
no
```

## 8.5. ÚLOHA O HANOJSKEJ VEŽI

### Príklad 8.6.

Na jednej z troch tyčiek sú nastoknuté kotúče 1,2,3,... N podľa veľkosti (dole je najväčší kotúč N ). Úlohou je premiestniť celú vežu na druhú tyčku, pričom jediná akcia (povolený elementárny krok) je preloženie voľného kotúča (neleží na ňom žiadny iný) z jednej tyčky na druhú tak, že všetky čiastkové veže na jednotlivých tyčkách sa musia stále zužovať smerom nahor (kotúč je teda možné preložiť buď na prázdnu tyčku alebo na väčší kotúč na inej tyčke).

Riešenie:

Počet stavov a počet hrán grafu prudko narastá so zväčšujúcim sa počtom kotúčov:

| | | | | | |
|---|---|---|---|---|---|
| počet kotúčov | 1 | 2 | 3 | 4 | 5 |
| počet stavov | 3 | 9 | 27 | 81 | 243 |
| počet hrán grafu | 3 | 12 | 39 | 120 | 363 |

Je teda zrejme neúčelné pokúšať sa o popis stavového priestoru pomocou izolovaných faktov pre každú jednotlivú hranu. Výhodnejšie je opísať prologovskými pravidlami povolené akcie. Definujme stav ako štruktúru s(L1,L2,L3) , kde jednotlivé argumenty sú zoznamy, reprezentujúce čiastkové veže na prvej, druhej a tretej tyčke (hlava zoznamu predstavuje voľný kotúč, ktorý je možné premiestňovať, resp. na ktorý je možné položiť iný kotúč).

Povolený elementárny krok je možné popísať vhodným pravidlom: napríklad z prvej tyčky možno preniesť voľný kotúč na druhú tyčku vtedy, keď na prvej tyčke vôbec nejaký kotúč je a keď druhá tyčka je buď prázdna, alebo keď je horný kotúč na druhej tyčke väčší, ako premiestňovaný kotúč, čo prostriedkami jazyka PROLOG zapíšeme nasledovne:

```
oh(s([H1|T1],L2,L3),s(T1,[H1|L2],L3)) :- L2 = [H2|_], H1<H2;
                                         L2 =  [].
```

Podobne možno zapísať ďalšie pravidlá pre zvyšných päť kombinácií dvojíc tyčiek 2-->1, 1-->3, 3-->1, 3-->2, 2-->3 :

```
oh(s(L1,[H2|T2],L3),s([H2|L1],T2,L3)) :- L1 = [H1|_], H2<H1;
                                         L1 =   [].

oh(s([H1|T1],L2,L3),s(T1,L2,[H1|L3])) :- L3 = [H3|_], H1<H3;
                                         L3 =   [].

oh(s(L1,L2,[H3|T3]),s([H3|L1],L2,T3)) :- L1 = [H1|_], H3<H1;
                                         L1 =   [].

oh(s(L1,L2,[H3|T3]),s(L1,[H3|L2],T3)) :- L2 = [H2|_], H3<H2;
                                         L2 =   [].

oh(s(L1,[H2|T2],L3),s(L1,T2,[H2|L3])) :- L3 = [H3|_], H2<H3;
                                         L3 =   [].
```

Otázka pre dva kotúče vyzerá nasledovne:

```
?- depth(s([1,2],[],[]),s([],[1,2],[]),P).
```

pre ktorú existuje 12 riešení, z ktorých najkratšiu cestu predstavuje zoznam:

```
P = [s([1,2],[],[]),s([2],[],[1]),s([],[2],[1]),s([],[1,2],[])]
```

Pre tri kotúče už existuje 1740 riešení, najkratšie pozostáva z ôsmych stavov a najdlhšie z 27 stavov. V tomto prípade možno voľbou obmedzenej hĺbky prehľadávania významne zefektívniť proces riešenia.

Úlohu o Hanojskej veži je možné oveľa elegantnejšie riešiť metódou kľúčového operátora [9]. Prepis tejto metódy do PROLOGu je uvedený napr. v [2]. Pri riešení sa využíva rekurzia a to na dvoch miestach v tele kľúčového pravidla. Keď označíme tyčku, z ktorej sa má premiestniť veža z N kotúčov ako Z , tyčku na ktorú sa premiestňuje ako Na a tretiu, pomocnú tyčku ako Pom , potom možno definovať predikát prekladania veže nasledovne:

```
vpreloz(0,_,_,_)      :- !.
vpreloz(N,Z,Na,Pom) :- M is N-1,          vpreloz(M,Z,Pom,Na),
                       kpreloz(N,Z,Na), vpreloz(M,Pom,Na,Z).
```

t.j. veža o N-1 kotúčoch sa preloží z tyčky Z na tyčku Pom, spodný N -tý) kotúč sa preloží z tyčky Z na Na (štvrtý argument predikátu vpreloz je označenie pomocnej tyčky). Predikát kpreloz realizuje nájdené kroky riešenia, v najjednoduchšom prípade výpisom týchto krokov na terminál:

```
kpreloz(N,Z,Na):-write('preloz kotuc cislo '), write(N),
                 write(' z tycky '),           write(Z),
                 write(' na tycku '),          write(Na), nl.
```

## 8.6. SVET KOCIEK

### Príklad 8.7.

Klasickou úlohou pre testovanie systémov s umelou inteligenciou je rekonfigurácia východiskovej štruktúry kociek na zadanú cieľovú štruktúru. V najjednoduchšom variante sveta kociek sa pracuje s niekoľkými kockami rovnakých rozmerov, ktoré môžu ležať buď priamo na podložke, alebo na inej kocke (na každej kocke môže ležať iba jediná kocka). Premiestňovať je možné naraz iba jednu kocku, a to takú, na ktorej iná kocka neleží. Navrhnite reprezentáciu stavov pre úlohy tohoto typu.

Riešenie:

Každému stavu prislúcha jedna, alebo viacej veží zostavených z kociek: spodná kocka leží na podložke a jedine hornú kocku veže je možné premiestňovať. Jednotlivé veže je v PROLOGu účelné reprezentovať zoznamom (podobne, ako v prípade Hanojskej veže - priklad 8.6), ktorého hlava predstavuje premiestniteľnú kocku. Vzhľadom k tomu, že počet veží môže byť v rôznych stavoch rôzny, použijeme pre reprezentáciu každého stavu zoznam zoznamov jednotlivých veží. Príklad jedného stavu a jeho popis v PROLOGu je na obr. 8.3.

```
   +---+
   | a |
   +---+                 +---+
   | b |                 | e |        [[a,b,c],[d],[e,f]]
   +---+     +---+       +---+
   | c |     | d |       | f |
---+---+-----+---+-------+---+---
```

obr. 8.3

Prechod zo stavu From do stavu To realizujú nasledujúce akcie:

1. prelož kocku H1 z niektorej veže na inú

2. zlož kocku H z jednej veže na podložku

Prvej akcii odpovedá pravidlo:

```
oh(From,To):- vyber([H1|T1],From,F1),
              vyber([H2|T2],F1,  F2),
              To = [ [H1,H2|T2], T1|F2 ].
```

Predikát vyber (viď príklad 7.1) má v tejto aplikácii vždy viazaný druhý argument - úplný zoznam - a generuje postupne všetky možné dvojice:

1. prvok zoznamu (prvý argument)

2. zvyšok zoznamu po odstránení tohoto prvku (tretí argument)

Prvý subcieľ odstráni z konfigurácie From vežu [H1|T1] čím vznikne konfigurácia F1 . Z tej sa odstráni veža [H2|T2] a vznikne tak konfigurácia F2 . Z prvej veže preložíme kocku H1 na druhú vežu a obidve takto zmenené veže [H1,H2|T2] a T1 pridáme späť ku konfigurácii F2 (tretí subcieľ). Na poradí veží nezáleží, preto sme mohli tieto zmenené veže pridávať na začiatok zoznamu zoznamov. Použitím predikátu vyber sme na druhej strane umožnili prístup ku všetkým prvkom (vežiam) zoznamu zoznamov (konfigurácie).

Je však potrebné ošetriť ešte špeciálny prípad, keď prvá veža pozostáva iba z kocky H1 (preloženie kocky z podložky na vežu) a zoznam T1 je teda prázdny a nemá zmysel ho pridávať v

treťom subcieli ku konfigurácii F2 (dokonca by to viedlo k nekorektnej práci celého programu). Výsledný tvar pravidla bude teda:

```
oh(From,[[H1,H2|T2]|T]) :- vyber([H1|T1],From,F1),
                           vyber([H2|T2],F1  ,F2),
                           (T1 \= [], T = [T1|F2] ;
                            T1  = [], T =    F2   ).
```

Druhej akcii (polož kocku H z nejakej veže na podložku) odpovedá pravidlo:

```
oh(From,[T,[H]|F])  :- vyber([H|T],From,F), T \= [].
```

v ktorom T označuje zbytok veže z ktorej sa odoberala kocka H. V novej konfigurácii pribudne "jednoposchodová" veža [H]. Test na konci tela pravidla vylúči zbytočné preloženie kocky H z jedného miesta na podložke na iné.

Pre takto navrhnutý popis zákonitostí sveta kociek nemožno priamo použiť predikát depth a ním volaný predikát collect podľa príkladu 8.2. Pri popise stavov sveta sme použili zoznam veží, pretože ich počet je premenlivý. Ako už bolo spomenuté, v danej úlohe nezáleží na poradí veží, avšak u zoznamov je poradie prvkov významné a tak rozdielnym zoznamom, napr.:

```
[[a,b,c],[e,f],[d]]     [[a,b,c],[d],[e,f]]
```

môže odpovedať ten istý stav.

Naznačený rozpor sa prejaví v definícii predikátu collect na dvoch miestach:

- pri testovaní na dosiahnutie koncového stavu (premenná SN v hlave prvého pravidla)

- pri testovaní, či sa nový stav New už raz nevyskytol v doteraz nájdenej postupnosti stavov, reprezentujúcej čiastkové riešenie úlohy (predikát member v tele druhého pravidla)

Potrebujeme vyjadriť skutočnosť, že keď z jedného zoznamu môžeme vytvoriť druhý zoznam zámenou poradia prvkov v prvom z nich, potom obidva zoznamy reprezentujú tú istú konfiguráciu veží vo svete kociek. Použijeme k tomuto účelu predikát permut podľa príkladu 7.6:

```
depth(S0,SN,Path)          :- collect0([S0],SN,Path).

collect0([H|T],  SN,Path) :- permut(H,SN), rev([SN|T],Path).
collect0([Old|T],SN,Path) :- oh(Old,New),
                             not member0(New,[Old|T]),
                             collect0([New,Old|T],SN,Path).

member0(P,[X|_]) :- permut(P,X).
member0(P,[_|T]) :- member0(P,T).
```

Keď chceme vežu z troch kociek prestavať v opačnom poradí, potom príslušné riešenia sa nájdu v nasledujúcom dialógu:

```
?- depth([[a,b,c]],[[c,b,a]],P).
P= [[[a,b,c]],[[b,c],[a]],[[b,a],[c]],[[c,b,a]]] ;
P= [[[a,b,c]],[[b,c],[a]],[[c],[b],[a]],[[b,a],[c]],[[c,b,a]]] ;
no
```

## 8.7. PREHĽADÁVANIE GRAFU DO ŠÍRKY

V niektorých úlohách (najmä keď sa hľadá iba najkratšia cesta v grafe) býva prehľadávanie do hĺbky veľmi neefektívne. V týchto prípadoch je účelnejšie použiť prehľadávanie do šírky. Systém PROLOG pri navracaní používa prehľadávanie do hĺbky, preto program, ktorý mu "vnúti" inú stratégiu prehľadávania - v našom prípade do šírky - bude v porovnaní s programom na prehľadávanie do hĺbky o niečo zložitejší.

Príklad 8.8.

Navrhnite predikát na prehľadávanie grafu do šírky. Graf je zadaný rovnakým spôsobom ako v predošlých príkladoch.

Riešenie:

Použijeme modifikáciu algoritmu, uvedeného v [16] a program bude ukážkou procedurálneho prístupu k programovaniu v PROLOGu. Budeme pracovať s dvoma skupinami faktov:

gen(to,from) uzol to je kandidátom na ďalšie vyšetrovanie, bol zvolený preto, že z from k nemu smeruje hrana grafu (gen ... uzol bol generovaný )

exp(to,from) uzol to bol vyšetrený; keď sa ukáže v ďalšom, že leží na najkratšej ceste, údaj from predstavuje predošlý uzol na tejto ceste (exp ... uzol bol expandovaný ).

Algoritmus prehľadávania je nasledovný:

1. zaraď gen(S,*) do databázy ( S je východzí uzol hľadanej cesty)

2. vyraď prvý fakt gen(Actual,Old) z databázy a zaraď fakt exp(Actual,Old) do databázy; keď to nie je možné (už žiadny fakt s predikátom gen v databáze neexistuje), prehľadávanie je neúspešné => cesta v grafe neexistuje; v opačnom prípade pokračuj ďalším bodom

3. nájdi uzol New , do ktorého vedie hrana z Actual ; keď taký uzol New neexistuje, potom sa vráť k bodu 2.; v opačnom prípade pokračuj ďalším bodom

4. prever, či v databáze sa už nevyskytuje niektorý z faktov gen(New, ) , resp. exp(New, ) ; keď áno, svedčí to o existencii cyklu a preto sa vráť k bodu 3.; v opačnom prípade pokračuj ďalším bodom

5. zaraď na koniec databázy fakt gen(New,Actual)

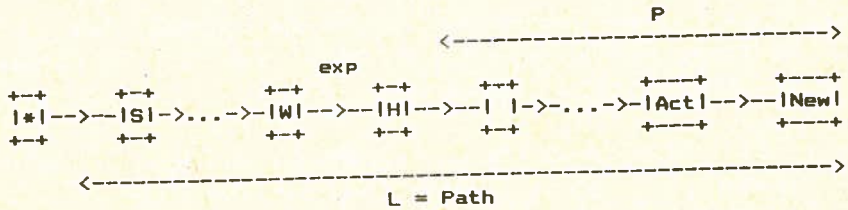
6. prever, či uzol New nepatrí do zoznamu cieľových uzlov; keď áno, prehľadávanie bolo úspešné a pokračuj ďalším bodom; keď nie, pokračuj v prehľadávaní bodom 3.

7. pomocou faktov exp vyhľadaj spätne všetky uzly, ležiace na nájdenej najkratšej ceste (pomocný predikát find to realizuje rekurzívnym akumulačným cyklom)

Prepis uvedeného algoritmu do jazyka PROLOG vyzerá nasledovne (v komentároch sú uvedené čísla krokov vyššie uvedeného algoritmu):

```
search(S,Goal,Path) :-
            retractall(gen(_,_)), retractall(exp(_,_)),
/*1*/       assertz(gen(S,*)), !,
/*2*/       retract(gen(Actual,Old)), assertz(exp(Actual,Old)),
/*3*/       oh(Actual,New),
/*4*/       not gen(New,_), not exp(New,_),
/*5,6*/     assertz(gen(New,Actual)), member(New,Goal),
/*7*/       find(Actual,[New],Path).
```

Pred začatím práce algoritmu je potrebné odstrániť z databázy všetky fakty gen a exp (prvé dva subciele uvedeného pravidla). Prechody na predošlé body algoritmu pri neúspešných testoch sa realizujú prostredníctvom mechanizmu navracania. Symbol rezu za tretím subcieľom vyvoláva v súlade s bodom 2. algoritmu neúspech pri pokuse o navracanie v prípade, že v databáze už neexistuje žiadny fakt gen . Zabráni sa tak zacykleniu, ktoré by spôsobilo opätovné splnenie subcieľa assertz(gen(S,*)) .




obr. 8.4

Predikát find je definovaný nasledovne:

```
find(*,L,L).
find(H,P,L) :- exp(H,W), find(W,[H|P],L).
```

a jeho funkciu pri zostavovaní zoznamu uzlov, ležiacich na nájdenej najkratšej ceste ilustruje obr. 8.4.

Predikát search možno použiť pre príklady 8.3 - 8.7.

Príklad 8.9.

Upravte program z príkladu 8.8 tak, aby bolo možné sledovať proces postupného expandovania a generovania uzlov v procese hľadania najkratšej cesty.

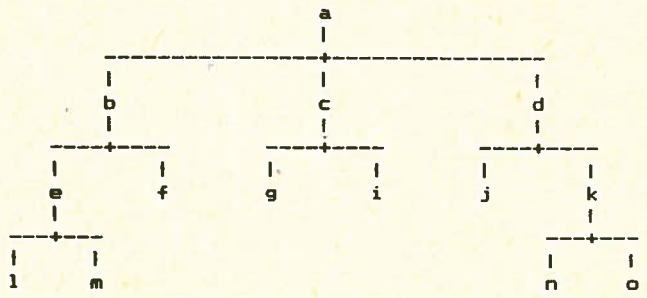
Riešenie:

Stačí program doplniť o dva pomocné výpisy:

- pred subcieľ oh(Actual,New) v tele pravidla definujúceho predikát search doplníme nl, write(Actual), write(':'),

- pred subcieľ member(New,Goal) doplníme tab(1), write(New),

Na každom riadku sa vypíše meno vyšetrovaného uzla a za dvojbodkou mená tých uzlov, do ktorých vedú hrany z vyšetrovaného uzla.




obr. 8.5

Pre príklad grafu podľa obr. 8.5 sú fakty nasledovné:

```
oh(a,b).  oh(b,e).  oh(c,g).  oh(d,j).  oh(e,l).  oh(k,n).
oh(a,c).  oh(b,f).  oh(c,i).  oh(d,k).  oh(e,m).  oh(k,o).
oh(a,d).
```

a dialóg má tento tvar:

```
?- search(a,[n],p).
a: b c d
b: e f
c: g i
d: j k
e: l m
f:
g:
i:
j:
k: n
p = [a,d,k,n] ;
 o                    /* patri ešte do riadku k: n */
l:
m:
n:
o:
no
```

# 9. MODELOVANIE SYSTÉMU STRIPS

## 9.1. PODSTATA SYSTÉMU STRIPS

Systém STRIPS (akronym STanford Research Institute Problem Solver) [7] využíva formalizáciu podobnú stavovému priestoru. Tzv. modely sveta reprezentujú množinu situácií, ktoré môžu v uvažovanom type úloh nastať. Každý model sveta je tvorený množinou formúl predikátového počtu a každá akcia f odpovedá zmene jedného modelu sveta na iný, čo sa formálne označuje:

$$M2=f(M1)$$

pričom akcia je charakterizovaná trojicou:

$$f = (Cond,Del,Add)$$

kde

Cond - podmienka aplikovateľnosti operátora f

Del - množina formúl, ktorá musí byť z modelu sveta vypustená pri použití operátora f

Add - množina formúl, ktorá musí byť pridaná k modelu sveta pri použití operátora f

Úloha sa pre STRIPS zadáva počiatočným modelom sveta M0 a množinou cieľových formúl G . Riešenie úlohy sa hľadá v tvare postupnosti:

$$RF = (f1,f2,...,fn)$$

pre ktorú existuje postupnosť modelov sveta:

$$RM = (M1,M2,...,Mn)$$

taká, že platí:

$$M1 = f1(M0)$$
$$M2 = f2(M1) = f2(f1(M0))$$
$$. . .$$
$$Mn = fn( ... f2(f1(M0)) ... )$$

a naviac množina Mn musí byť podmnožinou G .

Systém STRIPS rieši naznačený problém prostrednictvom automatického dokazovania teorémov matematickej logiky, využívajúc dôkaz sporom a tzv. rezolučný princíp . Nakoľko v zásade zhodný princíp využíva aj systém PROLOG, je možné jeho prostriedkami popísať činnosť STRIPSu bez toho, aby bolo potrebné podrobnejšie sa zaoberať otázkami automatického dokazovania.

V prvom kroku sa overuje, či množina cieľových formúl nie je obsiahnutá vo východzom modeli sveta M0 . Keď áno, úloha je vyriešená, keď nie, hľadajú sa tie akcie, ktoré pridávajú požadované formuly k modelu sveta a podmienky aplikovateľnosti týchto akcií sa stávajú novým subcieľom. Tento postup sa opakuje tak dlho, kým východzí model sveta nezaručí splnenie príslušných subcieľov. Naznačený proces vytvára postupne hľadaný lineárny plán v tvare postupnosti akcií.

## 9.2. NÁVRH PROGRAMU PRE STRIPS

Príklad 9.1.

Navrhnite predikát, simulujúci vyššie uvedené činnosti systému STRIPS. Model sveta reprezentujte vhodnými faktami v databáze.

Riešenie:
Zvoľme špeciálnu štruktúru:

Action : Cond --> Del % Add

pomocou ktorej budeme v ďalšom pri konkrétnych úlohách zadávať charakteristickú trojicu pre každú akciu ( Action je jej meno). Predpokladajme, že východzí model sveta je uložený v databáze vo forme faktov. Aplikácia akcie vyvolá príslušnú zmenu modelu sveta pridaním a odobraním faktov tvoriacich zoznamy Add a Del . V prvom priblížení možno navrhnúť hlavný predikát pre splnenie jediného cieľa v tvare:

```
goal(G) :- G.                    /* cieľ je obsiahnutý v modeli sveta */
goal(G) :-
  Action:Cond-->Del%Add,                  /*char. trojica pre akciu */
  member(G,Add),                  /* výber akcie pridávajúcej cieľ */
  goal(Cond),          /* rekurziou sa zaistí splnenie podmienok */
  retract(Del), asserta(Add).              /* úprava modelu sveta */
```

Príklad 9.2.

Prvý problém v riešení predošlého príkladu je, že hlavný cieľ G aj subciele Cond sú zoznamy. Navrhnite predikát, ktorý zaistí volanie predikátu goal pre všetky prvky tvoriace zoznam G resp. Cond (je potom potrebné zameniť v tele druhého pravidla tretí subcieľ).

Riešenie:

```
goall([]).
goall([H|T]) :- goal(H), goall(T).
```

Príklad 9.3.

Ďalším problémom je, že keď chceme plne využiť systém PROLOG pre zvolenú úlohu, musíme zaistiť korektné navracanie, t.j. keď STRIPS zistí, že sa dostal do "slepej uličky", musí sa vedieť vrátiť k predošlým modelom sveta. Navrhnite predikáty formujúce model sveta tak, aby umožnili rekonštrukciu pôvodného stavu databázy. Naviac by tieto predikáty mali pridávať a odoberať nie jednotlivé fakty, ale celé zoznamy faktov Add a Del.

Riešenie:

Pre opätovne splniteľné zaradenie zoznamu faktov navrhneme preto nový predikát:

```
assertl([]).
assertl([H|T])     :-assertb(H), assertl(T).

assertb(X)         :-X, !.
assertb(X)         :-asserta(X).
assertb(X)         :-retract(X), !, fail.
```

Predikát assertl realizuje volanie predikátu assertb pre každý prvok zoznamu. Prvá klauzula definície assertb zisťuje, či daný fakt už v databáze existuje: keď áno, potom cieľ assertb je úspešne splnený (ale nedá sa opätovne splniť, aby sa pri navracaní cez toto miesto nevyradil fakt, ktorý už tam bol skôr a nebol zaradený na tomto mieste). Druhá klauzula realizuje zaradenie príslušného faktu na začiatok databázy. Posledná, tretia klauzula odstraňuje pri navracaní fakt, ktorý bol na tomto mieste zaradený druhou klauzulou. Kombinácia !, fail zaistí, že sa po rekonštrukcii pôvodného stavu databázy pokračuje v navracaní a že subcieľ assertb nie je v pravom slova zmysle opätovne splniteľný (neprodukuje novú alternatívu riešenia). Uvedený predikát je názornou ukážkou využitia symbolu rezu pre dva rôzne účely (viď kapitola 3.5).

Úplne analogicky sa zostavia klauzuly definujúce komplementárny predikát retractl :

```
retractl([]).
retractl([H|T])     :- retractb(H), retractl(T).

retractb(X)         :- not X, !.
retractb(X)         :- retract(X).
retractb(X)         :- asserta(X), !, fail.
```

Príklad 9.4.

V procese formulácie nových subcieľov sa môže stať, že sa medzi podmienkami objaví "opakovaný" subcieľ, t.j. taký, ktorého splnenie sa požadovalo už v niektorom z predošlých krokov. Navrhnite úpravu doterajšieho riešenia, ktorá by riešila tento problém.

Riešenie:

Zacykleniu je možné zabrániť sledovaním už sformulovaných, ale ešte nesplnených subcieľov zaradením faktu pending(G) do databázy na začiatku rekurzívneho pravidla pre goal a jeho vyradením na konci pravidla (ovšem pomocou predikátov assertb a retractb ) a vložením nového pravidla pre goal medzi dve doterajšie:

```
goal(G) :- pending(G), !, fail.
```

Toto pravidlo spôsobí okamžitý neúspech pri pokuse splniť subcieľ, ktorého splňovanie už raz bolo zahájené, ale ešte nebolo ukončené.

Príklad 9.5.

Po uvedených úpravách bude program fungovať korektne, ale ešte neregistruje postup riešenia. Navrhnite potrebnú úpravu programu.

Riešenie:

Na konci rekurzívneho pravidla pre goal je potrebné doplniť subcieľ assert(plan(Action)) , ktorý po úspešnom splnení subcieľa zaradí do databázy meno akcie, ktorá toto splnenie realizovala. Pomocný predikát collect vytvorí z príslušných faktov v databáze zoznam, obsahujúci postupnosť akcií, reprezentujúcich hľadané riešenie úlohy (viď príklad 4.18):

```
collect(P,L) :- retract(plan(X)), collect([X|P],L), !.
collect(L,L).
```

Po uvedených úpravách bude program modelujúci činnosť systému STRIPS vyzerať nasledovne (bez definície pomocných predikátov, ktoré sa použijú v tvare uvedenom v predošlom texte):

```
strips(G,Plan) :- goall(G), collect([],Plan).

goall([]).
goall([H|T]) :- goal(H),goall(T).

goal(G) :- G.
goal(G) :- pending(G), !, fail.
goal(G) :- assertb(pending(G)),
           Action:Cond-->Del%Add, member(G,Add),
           goall(Cond), retractl(Del), assertl(Add),
           retractb(pending(G)), assertb(plan(Action)).
```

Na začiatku programu je potrebné ešte deklarovať príslušné operátory, napr. vo tvare:

```
?- op(9,xfx,:), op(9,xfx,-->), op(9,xfx,%).
```

Uvedený program má niektoré obmedzenia, ktoré je možné obísť vhodnou formou popisu modelu sveta a jednotlivých akcií. Jedná sa o to, že model sveta, aj podmienky aplikovateľnosti akcií sa zadávajú ako jednoduchá množina faktov, čo nie je vždy vyhovujúci prostriedok (nedá sa napríklad vyjadriť negácia nejakého faktu ako podmienka). Naviac jednotlivé podmienky aplikovateľnosti akcie sa síce chápu ako konjunkcia subcieľov, ale vyšetrujú sa oddelene a môže sa stať, že pri splňovaní niektorej podmienky zo zoznamu Cond sa z databázy odstráni niektorá z už splnených podmienok tohoto zoznamu, čo môže viesť k nekorektnej činnosti programu. Konkrétne príklady uvedené v ďalšom budú ilustrovať aj spôsob, ako sa naznačeným problémom vyhnúť.

## 9.3 VYSVETĽOVACÍ MECHANIZMUS

Pre lepšie pochopenie systému STRIPS a najmä pre prípadné ladenie konkrétnych príkladov je síce možné použiť ladiace prostriedky jazyka PROLOG, ale aj v prípade selektívneho ladenia je množstvo informácií príliš veľké a pre účely sledovania STRIPSu pomerne neprehľadné.

Príklad 9.6.

Navrhnite vysvetľovací mechanizmus, ktorý vytvorí prehľadný protokol o postupe STRIPSu pri hľadaní riešenia úlohy.

Riešenie:

Uvedieme iba definície tých predikátov, ktoré sa oproti predošlým príkladom zmenili:

```
strips(G,Plan) :- retractall(poc(_)), asserta(poc(1)),
                  write('Subor:'), read(File), tell(File),
                  goall(G), collect([],Plan), told.

goal(G) :- G, tab(8), write(G), nl.
goal(G) :- pending(G), !, fail.
goal(G) :- assertb(pending(G)),
           Action:Cond-->Del%Add, member(G,Add),
           choice(X), ppi(X,4), write(' G: '), write(G),
```

```
            write(' C: '), write(Cond), nl,
            Y is X+1, asserta(poc(Y)), goall(Cond),
            retractb(pending(G)), retractl(Del), assertl(Add),
            assertb(plan(Action)), write(>), ppi(X,3),
            write(' A: '), write(Action), nl.

choice(X) :- retract(poc(X)), !.

ppi(X,M) :- rad(X,0,N),
            (M>=N, SP is M-N, tab(SP), write(X) ;
                  cykl(0,M)                     ), !.

  /* cykl(0,M) tlačí M hviezdičiek */
  cykl(H,H) :- !.
  cykl(I,H) :- write(*), J is I+1, cykl(J,H).

  /* rad(X,0,N) určí rád N čísla X */
  rad(0,0,1) :- !.
  rad(0,L,L) :- !.
  rad(X,K,L) :- Y is X/10, Z is K+1, rad(Y,Z,L).
```

Fakt poc slúži na uchovávanie poradového čísla vytýčeného cieľa (toto číslo sa vo výpise objaví aj pri nájdenej akcii, ktorá tento cieľ splní). Predikát choice zabráni nežiadúcemu opätovnému splneniu subcieľa retract(poc(X)). Predikát ppi(X,M) realizuje výstup čísla X na M pozíciách. Keď počet pozícií nestačí (potrebný počet pozícií sa udáva treťom argumente predikátu rad ), tlačí sa M hviezdičiek. Predikát tell prepne výstupný prúd (tlač protokolu) do súboru File a na konci zase predikát told vráti výstupný prúd na terminál.

Vlastný protokol obsahuje jednotlivé ciele a subciele tak, ako ich STRIPS postupne analyzuje (spolu s podmienkami aplikovateľnosti príslušnej akcie) a po ich splnení sú v protokole uvedené akcie - a to v poradí, ako sa majú vykonávať (čísla odkazujú na miesto, kde bol príslušný cieľ, resp. subcieľ formulovaný). Okrem toho sa vypisujú fakty, použité v danej etape odvodzovania.

## 9.4. SVET KOCIEK

**Priklad 9.7.**

Navrhnite predikáty, umožňujúce popis modelu sveta a elementárnych akcií v tzv. "svete kociek".

Riešenie:

K popisu jednotlivých modelov sveta použijeme štyri predikáty, deklarované ako operátory:

```
X na Y ............... op(7,xfx,na).
X odkryty ............ op(7,xf,odkryty).
X zdvihnuty .......... op(7,xf,zdvihnuty).
X na_podlozke ........ op(7,xf,na_podlozke).
```

a jeden fakt:

```
chapadlo_prazdne.
```

K popisu elementárnych akcií použijeme ďalšie štyri operátory:

```
X zdvihni_z_podlozky ... op(7,xf,zdvihni_z_podlozky).
X poloz_na_podlozku .... op(7,xf,poloz_na_podlozku).
X poloz_na Y ........... op(7,xfx,poloz_na).
X zdvihni_z Y .......... op(7,xfx,zdvihni_z).
```

a príslušné štruktúry majú tvar:

```
X zdvihni_z_podlozky
: [X na_podlozke,X odkryty,chapadlo_prazdne]
-->[X odkryty,chapadlo_prazdne,X na_podlozke]
% [X zdvihnuty].

X zdvihni_z Y
: [X na Y,X odkryty,chapadlo_prazdne]
-->[X odkryty,X na Y,chapadlo_prazdne]
% [X zdvihnuty,Y odkryty].
```

```
   X poloz_na_podlozku
 : [X zdvihnuty]
-->[X zdvihnuty]
 % [X odkryty,chapadlo_prazdne,X na_podlozke].

   X poloz_na Y
 : [Y odkryty,X zdvihnuty]
-->[X zdvihnuty,Y odkryty]
 % [X odkryty,X na Y,chapadlo_prazdne].
```

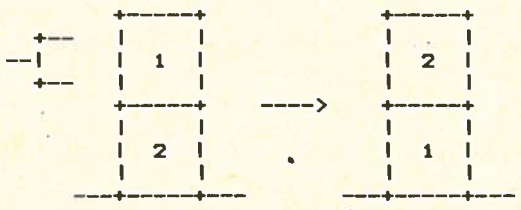
U prvých dvoch akcií je účelné nechať chapadlo_prazdne ako poslednú podmienku, aby sa pri splňovaní X_odkryty neporušila. Podobne u poslednej akcie je potrebné zabezpečiť, aby Y bolo odkryté a až potom zdvihnúť X . V prípade opačného usporiadania podmienok robot nemôže so zdvihnutou kockou X odkrývať Y (iba tak, že X zase položí).

Keď zvolíme východzí a cieľový stav podľa obr. 9.1, potom databáza musí na začiatku obsahovať fakty:

```
1 na 2.  2 na_podlozke.  1 odkryty.  chapadlo_prazdne.
```

a dialóg so systémom má nasledujúci priebeh (bez protokolu):

```
?- strips([2 na 1],P).
P = [1 zdvihni_z 2,1 poloz_na_podlozku,
     2 zdvihni_z_podlozky,2 poloz_na 1]
yes
```




obr. 9.1

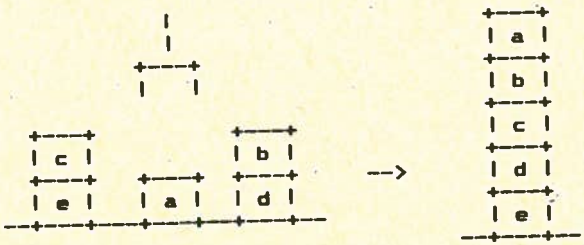
Protokol o hľadaní riešenia je nasledovný:

```
  1 G: 2 na 1 C: [1 odkryty,2 zdvihnuty]
       1 odkryty
  2 G: 2 zdvihnuty C: [2 na_podlozke,2 odkryty,chapadlo_prazdne]
       2 na_podlozke
  3 G: 2 odkryty C: [_325 na 2,_325 odkryty,chapadlo_prazdne]
       1 na 2
       1 odkryty
       chapadlo_prazdne
> 3 A: 1 zdvihni_z 2
  4 G: chapadlo_prazdne C: [_527 zdvihnuty]
       1 zdvihnuty
> 4 A: 1 poloz_na_podlozku
> 2 A: 2 zdvihni_z_podlozky
> 1 A: 2 poloz_na 1
```

Podčiarnik s rôznymi číslami reprezentuje vnútorné mená neviazaných premenných.

Na obr. 9.2 je naznačená zložitejšia úloha, pre ktorú východzí model sveta je popísaný faktami:

```
c na e.   b na d.   e na_podlozke.   a na_podlozke.
d na_podlozke.      c odkryty.       a odkryty.
b odkryty.          chapadlo_prazdne.
```




obr. 9.2

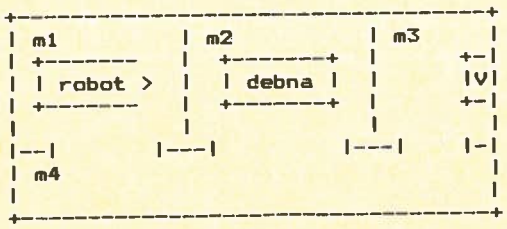
Dialóg so systémom je nasledujúci:

```
?- strips([d na e, c na d, b na c, a na b],P).
P = [c zdvihni_z e,          c poloz_na_podlozku,
     b zdvihni_z d,          b poloz_na_podlozku,
     d zdvihni_z_podlozky,   d poloz_na e,
     c zdvihni_z_podlozky,   c poloz_na d,
     b zdvihni_z_podlozky,   b poloz_na c,
     a zdvihni_z_podlozky,   a poloz_na b]
yes
```

## 9.5. ROBOTICKÁ VERZIA ÚLOHY "OPICA & BANÁN"

Príklad 9.8.

Pri testovaní systémov umelej inteligencie sa často používa úloha o opici a banáne: strapec banánov visí vo výške, kam opica nedosiahne; keď si však pod banány pristrčí debnu a vylezie na ňu, sladká odmena je na dosah. Na obr. 9.3 je schéma robotickej verzie: robot má rozsvietiť svetlo stlačením vypínača V v miestnosti m3 , avšak vypínač je tak vysoko, že robot ho môže zatlačiť iba keď vylezie na debnu (debna má pre tento účel špeciálnu nájazdovú rampu). Na začiatku úlohy je debna v miestnosti m2 a robot v miestnosti m1 . Navrhnite príslušné predikáty pre riešenie naznačenej úlohy pomocou systému STRIPS.




obr. 9.3

Riešenie:

Svet tejto úlohy je popísaný nasledovnými klauzulami:

```
robot v m1. debna v m2.  spoj(m1,m4). spoj(m3,m4). spoj(m2,m4).

dvere(X,Y) :- spoj(Y,X) ; spoj(X,Y).
```

Akcie potrebné k realizácii uvedenej úlohy sú:

```
    chod(X,Y)                 : [dvere(X,Y),robot v X]
--> [robot v X]               % [robot v Y].

    daj(D,X,Y)                : [dvere(X,Y),D v X,robot v X]
--> [robot v X,D v X]         % [robot v Y,D v Y].

    vylez(D)                  : [D v Z,robot v Z]
--> [robot v Z]               % [robot na D].

    zlez(D)                   : [D v Z,robot na D]
--> [robot na D]              % [robot v Z].

    zapal                     : [debna v m3,robot na debna]
--> []                        % [svieti].
```

Je potrebné deklarovať operátory použité v opise "sveta":

```
?- op(7,xfx,na), op(7,xfx,v).
```

Riešenie úlohy sa nájde v dialógu:

```
?- strips([svieti],P).
P = [chod(m1,m4), chod(m4,m2), daj(debna,m2,m4),
     daj(debna,m4,m3), vylez(debna), zapal]
yes
```

# 10. SYNTAKTICKÁ ANALÝZA PRI ROZPOZNÁVANÍ TVAROV

Rozpoznávanie je klasifikačná úloha, pri ktorej sa zisťuje, ku ktorej triede vyšetrovaný predmet patrí. Klasifikátor pritom neanalyzuje predmet priamo, ale prostredníctvom jeho obrazu - vhodne voleného formálneho opisu. Jedna skupina efektívnych metód rozpoznávania využíva štrukturálny opis predmetov [5], vychádzajúc pritom z teórie formálnych gramatík. V tomto prípade sú predmety reprezentované súborom primitív (elementárnych jednotiek obrazu) a relácií medzi primitívami. Každej rozpoznávanej triede prislúcha jedna gramatika.

Formálna gramatika G sa vyjadruje štvoricou

$$G = (Vn, Vt, P, S)$$

kde

Vn je množina neterminálnych symbolov jazyka (pomocné symboly potrebné pri analýze a odvodzovaní viet, ale nevyskytujúce sa v týchto vetách)

Vt je množina terminálnych symbolov jazyka (z nich sú tvorené vety daného jazyka)

P je množina substitučných pravidiel , ktorá definuje syntax formálne správnych viet jazyka

S je štartovací symbol , ktorým sa začina analýza/odvodzovanie vety.

Pri rozpoznávaní je obraz predmetu reprezentovaný vetou , ktorú tvoria terminálne symboly . Predmet je zaradený do tej triedy, ktorej gramatika generuje príslušnú vetu. Overovanie toho, či niektorá gramatika danú vetu generuje alebo nie, sa označuje ako syntaktická analýza .

Pri rozpoznávaní vizuálnych obrazov sa často používajú tzv. reťazcové gramatiky s jednou reláciou - zreťazením. Sú vhodné pre obrazce, ktoré možno nakresliť jedným ťahom. Jedna z možností, ako voliť primitíva je uvedená na obr. 10.1.

```
          g
          ^
          |
          |
e <----- ----->  a
```

obr. 10.1

Ďalšie obrázky reprezentujú príklady obrazov, patriacich do dvoch tried - znaky L a znaky C (obr. 10.2).

```
                                  ----->
^                                 ^
|                                 |
|                                 |          ----->----->
^          ^                      ^          ^
|          |                      |          |
|          |                      |          |
<-----     <-----<-----           <-----     <-----<-----

 e g g      e e g                  e g g a    e e g a a
```

obr. 10.2

Príslušné gramatiky GL a GC sú definované nasledovne:

VtL = { e,g }  VtC = { a,e,g }

VnL = { S,Q,R }  VnC = { S,T,U }

PL: S --> eQ
Q --> R
Q --> eQ
R --> g
R --> gR

PC: S --> eTa
T --> U
T --> eTa
U --> g
U --> gU

Pozn.: Reťazec e e g a nie je generovaný gramatikou GC a nepovažujeme ho za znak C .

Príklad 10.1.

Navrhnite syntaktický analyzátor, ktorý rozpoznáva znaky L a C (veta, reprezentujúca obraz znaku je tvorená zoznamom príslušných primitív).

Riešenie:

```
sa([e|Q],'L') :- q(Q).                          /* S --> eQ  */
sa([e|T],'C') :- concat(Y,[a],T), t(Y).         /* S --> eTa */

q(Q)      :- r(Q).                              /* Q --> R   */
q([e|Q]) :- q(Q).                               /* Q --> eQ  */

r([g]).                                         /* R --> g   */
r([g|R]) :- r(R).                               /* R --> gR  */

t(T)      :- u(T).                              /* T --> U   */
t([e|T]) :- concat(Y,[a],T), t(Y).              /* T --> eTa */

u([g]).                                         /* U --> g   */
u([g|U]) :- u(U).                               /* U --> gU  */
```

Predikát concat slúži pôvodne k spájaniu dvoch zoznamov (viď príklad 2.7). Tentokrát ho používame na oddelenie/pripojenie posledného prvku zoznamu.

Pravidlá definujúce predikáty u a r sú prakticky rovnaké, a preto je možné vynechať pravidlá pre u a upraviť prvé pravidlo definujúce predikát t na tvar:

```
t(T) :-r(T).
```

Test:

```
?- sa([e,e,g,g,g],X).
X = L
yes
```

```
?- sa([e,e,g,g,a,a],X).
x = C
yes

?- sa([e,e,g,g,g,a],X).
no
```

Priklad 10.2.

Upravte predošlý program tak, aby analyzovaná veta bola zadávaná v tvare atómu.

Riešenie:

```
sa(S,'L') :- name(S,[101|Q]), q(Q).        /* L:  S -->  eQ   */
sa(S,'C') :- name(S,[101|T]),
             concat(Y,[97],T), t(Y).       /* C:  S -->  eTa  */

q(Q)        :- r(Q).                       /* L:  Q -->  R    */
q([101|Q])  :- q(Q).                       /* L:  Q -->  eQ   */

t(T):-r(T).                                /* C:  T -->  R    */
t([101|T]) :- concat(Y,[97],T), t(Y).      /* C:  T -->  eTa  */

r([103]).                                  /*     R -->  g    */
r([103|R]) :- r(R).                        /*     R -->  gR   */
```

Test:

```
?- sa(eeggg,X).
X = L
yes

?- sa(eegggaa,X).
X = C
yes

?- sa(eeggga,X).
no
```

Príklad 10.3.

Upravte pôvodný program tak, aby v prípade otázok

sa(Z,'L') resp. sa(Z,'C')

generoval všetky možné reťazce Z reprezentujúce príslušný znak, pričom rozmery znaku spĺňajú podmienky: šírka =< 2, výška =< 3.

Riešenie:

Pôvodný program je potrebné doplniť o počítadlá rekurzívnych volaní a ich testovanie:

```
sa([e|Q],'L') :- q(Q,0).                        /* S --> eQ  */
sa([e|T],'C') :- concat(Y,[a],T), t(Y,0).       /* S --> eTa */

q(Q,_)      :- r(Q,0).                          /* Q --> R   */
q([e|Q],I)  :- J is I+1, J<2, q(Q,J).           /* Q --> eQ  */

r([g],_).                                       /* R --> g   */
r([g|R],I)  :- J is I+1, J<3, r(R,J).           /* R --> gR  */

t(T,_)      :- r(T,0).                          /* t --> R   */
t([e|T],I)  :- J is I+1, J<2,
               concat(Y,[a],T), t(Y,J).         /* t --> eTa */
```

Test:

```
?- sa(Z,'C').
Z = [e,g,a] ;
Z = [e,g,g,a] ;
Z = [e,g,g,g,a] ;
Z = [e,e,g,a,a] ;
Z = [e,e,g,g,a,a] ;
Z = [e,e,g,g,g,a,a] ;
- nekonečný cyklus-
```

Predikát concat generuje do nekonečna stále dlhšie a dlhšie zoznamy Y a I , ktoré však nevedú k žiadnemu riešeniu Z . Uvedenému zacykleniu možno zabrániť náhradou predikátu concat predikátom join , ktorý má obmedzený počet rekurzívnych volaní:

```
join(L1,L2,L3) :- jp(L1,L2,L3,0).

jp([],L,L,_).
jp([H|L1],L2,[H|L3],N) :- N>=5, !, fail;
                                  M is N+1, jp(L1,L2,L3,M).
```

K organizácii cyklu môžme použiť aj "peanovské štruktúry", viď záver kapitoly 4.5:

```
join1(L1,L2,L3):-jp1(L1,L2,L3,s(s(s(s(s(s(0)))))))).

jp1([],L,L,_).
jp1([H|L1],L2,[H|L3],s(X)) :- jp1(L1,L2,L3,X).
```

# 11. EXPERTNÉ SYSTÉMY

## 11.1. ZÁKLADNÉ POJMY

Expertné systémy sú programy pre riešenie úloh, ktoré sú vo všeobecnosti pokladané za obtiažne a ktoré je schopný uspokojivo riešiť iba špecialista v danom odbore - expert. Existujú dva typy expertných systémov: diagnostické a plánovacie . Ďalšie úvahy sa budú týkať podstatne rozšírenejších diagnostických systémov, ktorých úlohou je zistiť, ktorá z možných cieľových hypotéz sa vzťahuje na konkrétny vyšetrovaný problém (cieľové hypotézy predstavujú napr. diagnózy chorôb, typ technického zariadenia, atď.).

Expertný systém (ES) pozostáva z troch častí:

- prázdny ES , tvorený riadiacim mechanizmom, vysvetľovacím mechanizmom a aktuálnym modelom vyšetrovaného problému

- báza vedomosti , tvorená informáciami, ktoré zadáva expert a ktoré sú pre riešený okruh problémov nemenné

- báza údajov , tvorená informáciami, ktoré zadáva používateľ a týkajú sa iba jedného konkrétneho problému (môže byť zadaná formou vyplneného dotazníku, ale obvykle sa zadáva dialógovo v priebehu konzultácie)

Konzultácia s expertným systémom pozostáva z otázok, kladených systémom a z odpovedí používateľa, na základe ktorých systém postupne modifikuje aktuálny model riešeného prípadu. Sled otázok nie je pevne stanovený, ale závisí na stave aktuálneho modelu (t.j. na odpovediach na predošlé otázky), pričom systém kladie vždy tú otázku, ktorej zodpovedanie v danej etape riešenia problému prináša "najbohatšiu" informáciu. Dialóg pokračuje tak dlho, kým nie sú zodpovedané všetky relevantné otázky (ich počet býva výrazne menší ako počet všetkých možných otázok pre danú triedu problémov). Na konci konzultácie systém oznámi, ktorá z vyšetrovaných cieľových hypotéz odpovedá danému prípadu.

Vedomosti môžu byť reprezentované rôznymi spôsobmi, z ktorých najčastejšie používaným sú produkčné pravidlá . Jedná sa o formálny prepis výrokov "KEĎ predpoklady POTOM výsledok , kde predpoklady predstavujú zvyčajne konjunkciu, resp. disjunkciu výrokov a výsledok predstavuje výrok, ktorý vyplýva z predpokladov. Báza vedomostí tvorená produkčnými pravidlami, predstavuje potom tzv AND-OR graf , kde cieľové hypotézy (výroky) odpovedajú koreňovým uzlom a kde odpovede používateľa na jednotlivé otázky prislúchajú listovým uzlom. Medziľahlé uzly reprezentujú subciele, ktoré umožňujú dekompozíciu celej úlohy. Sú potrebné vtedy, keď cieľovej hypotéze odpovedá produkčné pravidlo, ktoré má na pravej strane zložitejší logický výraz (t.j. nie výhradne disjunkciu, alebo konjunkciu). Dekompozícia pomocou medziľahlých uzlov je výhodná v úlohách, kde výrok odpovedajúci takémuto uzlu predstavuje predpoklad viacerých cieľových hypotéz (resp. subcieľov vyššej úrovne).

## 11.2. REPREZENTÁCIA BÁZY VEDOMOSTÍ A BÁZY ÚDAJOV

Prepis AND-OR grafu bázy vedomostí do jazyka PROLOG je veľmi jednoduchý: každému produkčnému pravidlu odpovedá pravidlo prologovské, pričom je účelné použiť rôzne predikáty pre koreňové, medziľahlé a listové uzly.

Príklad 11.1.

Zapíšte v jazyku PROLOG bázu vedomostí, reprezentovanú AND-OR grafom podľa obr 11.1 (dvojitej čiare odpovedá AND uzol, jednoduchej OR uzol).

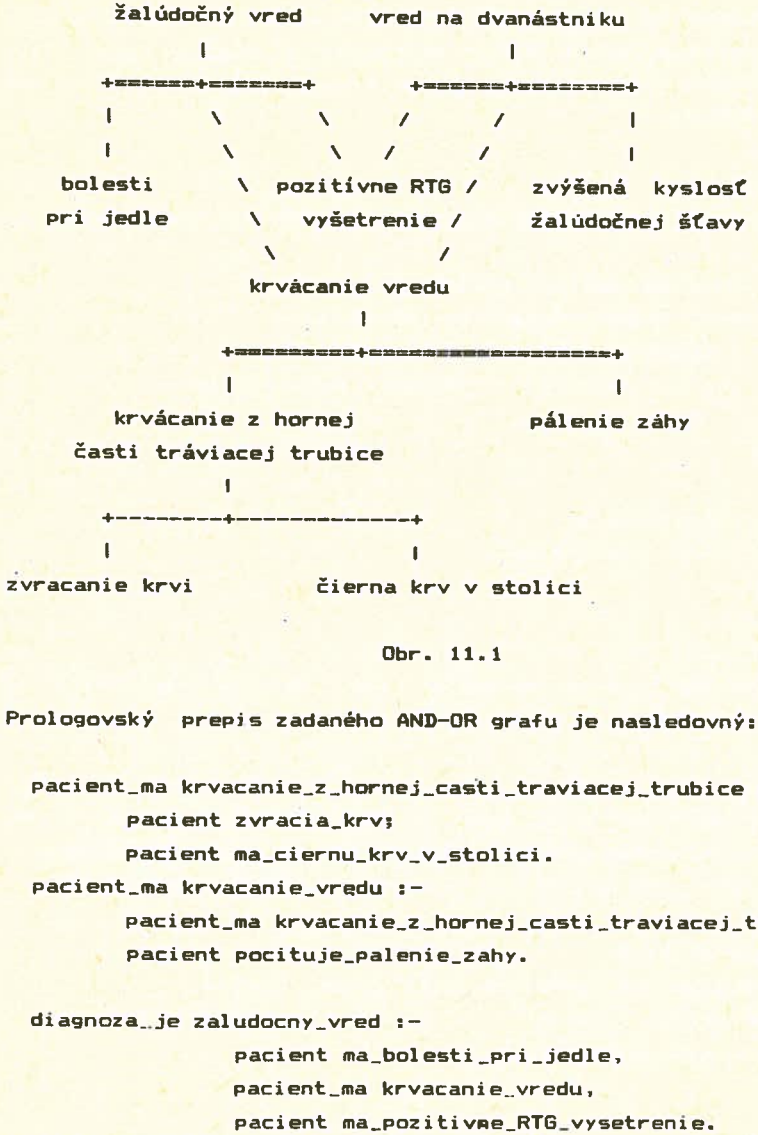
Riešenie:

Pre označenie jednotlivých typov uzlov použijeme nasledujúce predikáty (deklarované ako operátory):

- koreňové uzly (diagnózy) ..... op(7, fx, diagnoza_je )

- medziľahlé uzly (dielčie diagnózy) ... op(7, fx, pacient_ma )

- listové uzly (otázky) ..... op(7, fx, pacient )




Obr. 11.1

Prologovský prepis zadaného AND-OR grafu je nasledovný:

```
pacient_ma krvacanie_z_hornej_casti_traviacej_trubice :-
        pacient zvracia_krv;
        pacient ma_ciernu_krv_v_stolici.
pacient_ma krvacanie_vredu :-
        pacient_ma krvacanie_z_hornej_casti_traviacej_trubice,
        pacient pocituje_palenie_zahy.

diagnoza_je zaludocny_vred :-
                pacient ma_bolesti_pri_jedle,
                pacient_ma krvacanie_vredu,
                pacient ma_pozitivne_RTG_vysetrenie.
diagnoza_je vred_na_dvanastniku :-
                pacient ma_pozitivne_RTG_vysetrenie,
                pacient_ma krvacanie_vredu,
                pacient ma_zvysenu_kyslost_zaludocnej_stavy.
diagnoza_je nezistitelna.
```

V najjednoduchšom prípade môže byť báza údajov (odpovede používateľa) reprezentovaná faktami s predikátom pacient pre každú kladnú odpoveď. Ako riadiaci mechanizmus ES možno použiť priamo system PROLOG-u.

Príklad 11.2.

Ako bude vyzerať dialóg používateľa s ES, keď báza údajov je:

```
pacient pocituje_palenie_zahy.
pacient ma_ciernu_krv_v_stolici.
pacient ma_pozitivne_RTG_vysetrenie.
pacient ma_zvysenu_kyslost_zaludocnej_stavy.
```

Riešenie:

```
?- diagnoza_je X.
X = vred_na_dvanastniku
yes
```

Príklad 11.3.

Upravte zápis bázy vedomostí tak, aby sa vypisovali aj dielčie diagnózy.

Riešenie:

Nahradíme predikát diagnoza_je predikátom pacient_ma a poslednú klauzulu zameníme za pacient_ma_nezistitelnu_diagnozu . Potom bude dialóg pre vyššie uvedenú bázu údajov nasledovný:

```
?- pacient_ma X.
X = krvacanie_z_hornej_casti_traviacej_trubice ;
X = krvacanie_vredu ;
X = vred_na_dvanastniku
yes
```

Príklad 11.4.

Upravte riadiaci mechanizmus tak, aby ES získaval údaje v priebehu dialógu s používateľom.

Riešenie:

Dialóg systému s používateľom bude realizovať pravidlo:

```
pacient MaPriznak :- write('Je pravda, ze pacient '),
                     write(MaPriznak), write(' ? '), read(a).
```

kde sa predpokladá reprezentácia kladnej odpovede atómom a . Napriklad báze údajov podľa prikladu 11.2 odpovedá nasledujúci dialóg (uvažujeme pôvodnú definíciu predikátov pacient_ma a diagnoza_je podľa príkladu 11.1):

```
?- diagnoza_je Dg.
Je pravda, ze pacient ma_bolesti_pri_jedle ? n.
Je pravda, ze pacient ma_pozitivne_RTG_vysetrenie ? a.
Je pravda, ze pacient zvracia krv ? n.
Je pravda, ze pacient ma_ciernu_krv_v_stolici ? a.
Je pravda, ze pacient pocituje_palenie_zahy ? a.
Je pravda, ze pacient ma_zvysenu_kyslost_zaludocnej_stavy ? a.

Dg = vred_na_dvanastniku
yes
```

## 11.3. ZVÝŠENIE EFEKTÍVNOSTI PREHĽADÁVANIA AND-OR GRAFU

V niektorých príkladoch je dialóg vedený doterajšou verziou riadiaceho mechanizmu nelogický:

```
?- diagnoza_je Dg.
Je pravda, ze pacient ma_bolesti_pri_jedle ? a.
Je pravda, ze pacient zvracia_krv ? n.
Je pravda, ze pacient ma_ciernu_krv_v_stolici ? a.
Je pravda, ze pacient pocituje_palenie_zahy ? a.
Je pravda, ze pacient ma_pozitivne_RTG_vysetrenie ? n.
Je pravda, ze pacient ma_pozitivne_RTG_vysetrenie ? ...
```

Expertný systém zavrhol hypotézu zaludocny_vred a začína vyšetrovať novú hypotézu (vred_na_dvanactniku ) zopakovaním už raz zodpovedanej otázky. V ďalšom dialógu sa tiež zavrhne prvá hypotéza - tentokrát kvôli nesplneniu subcieľa krvacanie_vredu a

pri pokuse o dokázanie druhej hypotézy sa tento subcieľ začne vyšetrovať znova:

```
?- diagnoza_je Dg.
Je pravda, ze pacient ma_bolesti_pri_jedle ? a.
Je pravda, ze pacient zvracia_krv ? n.
Je pravda, ze pacient ma_ciernu_krv_v_stolici ? n.
Je pravda, ze pacient ma_pozitivne_RTG_vysetrenie ? a.
Je pravda, ze pacient zvracia_krv ?   ...
```

Treti typ nelogicky vedeného dialógu ilustruje príklad:

```
?- diagnoza_je Dg.
Je pravda, ze pacient ma_bolesti_pri_jedle ? a.
Je pravda, ze pacient zvracia_krv ? a.
Je pravda, ze pacient pocituje_palenie_zahy ? n.
Je pravda, ze pacient ma_ciernu_krv_v_stolici ?  ...
```

Systém po kladnej odpovedi na druhú otázku odvodil platnosť subcieľa krvacanie z hornej časti trávicej trubice a po neúspechu, vyvolanom zápornou odpoveďou na tretiu otázku sa pokúša dokázať tento subcieľ ešte raz, ale iným spôsobom. Tento postup je z hľadiska PROLOGu správny, ale z hľadiska ES úplne zbytočný: odpoveď na poslednú otázku už nijako nemôže ovplyvniť skutočnosť, že subcieľ krvacanie vredu je nesplnený.

Príklad 11.5.

Upravte pôvodný program tak, aby sa v dialógu nevyskytli uvedené nedostatky a aby systém oznámil aj dokázané dielčie hypotézy.

Riešenie:

```
pacient MaPriznak :-
    write('Je pravda, ze pacient '),
    write(MaPriznak), write(' ? '), read(Odp),
    (Odp = a, asserta( pacient MaPriznak :- !), ! ;
              asserta((pacient MaPriznak :- !, fail)), !, fail).
```

```
pacient_ma krvacanie_z_hornej_casti_traviacej_trubice :-
      zaciatok(krvacanie_z_hornej_casti_traviacej_trubice),
      (pacient zvracia_krv ; pacient ma_ciernu_krv_v_stolici),
      koniec(krvacanie_z_hornej_casti_traviacej_trubice), !.
pacient_ma krvacanie_vredu :-
      zaciatok(krvacanie_vredu),
      pacient_ma krvacanie_z_hornej_casti_traviacej_trubice,
      pacient pocituje_palenie_zahy,
      koniec(krvacanie_vredu), !.

zaciatok(Cosi) :- asserta((pacient_ma Cosi :- !, fail)).

koniec(Cosi)   :- asserta(pacient_ma Cosi :- !),
                  write('Pacient ma '), write(Cosi), nl.

expert :- retract( pacient_ma _ :- !        ), fail.
expert :- retract((pacient_ma _ :- !, fail)), fail.
expert :- retract( pacient    _ :- !        ), fail.
expert :- retract((pacient    _ :- !, fail)), fail.
expert :- diagnoza_je D, nl, write('Diagnoza je '), write(D).
```

Po zodpovedaní otázky sa zaradí do databázy informácia o tejto odpovedi pre prípad, že ES ju bude ešte potrebovať pri svojej ďalšej činnosti. Keď sa začne vyšetrovať nejaký subcieľ, uloží sa do databázy informácia o jeho nesplnení a v prípade, že sa predsa len podarí splniť tento subcieľ, tak sa o tom uloží nová informácia do databázy <u>pred</u> informáciu o neúspechu. Naviac sa vypíše oznam o splnení subcieľa (t.j. o dokázaní čiastkovej hypotézy). Uvedené úpravy riešia prvé dva typy problematických dialógov. Tretí problém riešia symboly rezu na konci pravidiel pre jednotlivé subciele databázy.

Pred každým novým dialógom je potrebné obnoviť pôvodný stav databázy odstránením pomocných klauzúl, reprezentujúcich zodpovedané otázky a vyšetrené medziľahlé uzly. Pre tento účel nemožno použiť štandardný predikát <u>retractall</u> , nakoľko ten by odstránil aj klauzuly pre medziľahlé uzly a pravidlo kladenia otázok.

Po uvedených úpravách vyzerá typický dialóg takto:

```
?- expert.
Je pravda, ze pacient ma_bolesti_pri_jedle ? n.
Je pravda, ze pacient ma_pozitivne_RTG_vysetrenie ? a.
Je pravda, ze pacient zvracia krv ? n.
Je pravda, ze pacient ma_ciernu_krv_v_stolici ? a.
Pacient ma krvacanie_z_hornej_casti_traviacej_trubice
Je pravda, ze pacient pocituje_palenie_zahy ? a.
Pacient ma krvacanie_vredu
Je pravda, ze pacient ma_zvysenu_kyslost_zaludocnej_stavy ? a.

Diagnoza je vred_na_dvanastniku
yes
```

## 11.4. EFEKTÍVNEJŠIA FORMA ZÁPISU BÁZY VEDOMOSTÍ

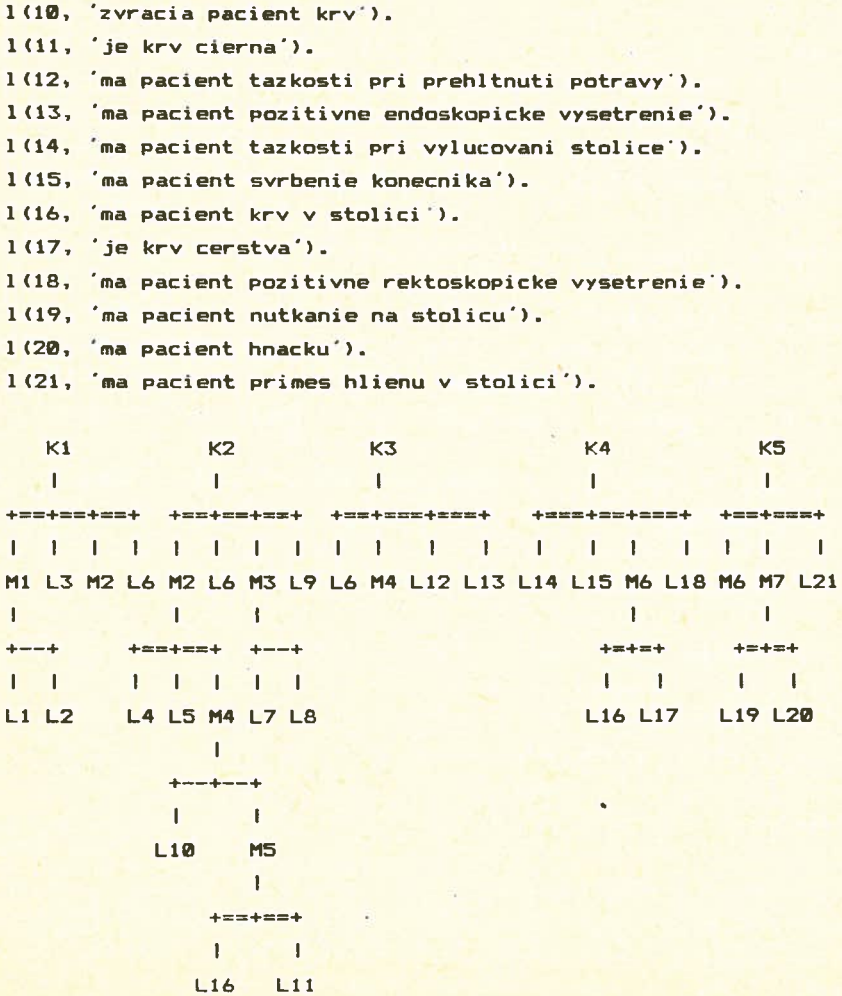
V prípade rozsiahlejších báz vedomostí sa stáva uvedený spôsob zápisu jednotlivých pravidiel nepohodlným predovšetkým preto, že sa pri každom výskyte niektorého uzla odvolávame naňho plným textom. Výhodnejším sa javí označiť každý uzol jednoznačným identifikátorom a v zápise pravidiel používať iba tieto identifikátory. Na priradenie plných textov k týmto identifikátorom sa použijú samostatné fakty.

Príklad 11.6.

Navrhnite expertný systém podľa vyššie uvedených zásad. AND-OR graf bázy vedomostí je na obr.11.2. Texty, prislúchajúce jednotlivým uzlom sú uvedené v nasledovných faktoch:

```
k(1, 'pacient ma krvacanie zaludocneho vredu').
k(2, 'pacient ma krvacanie vredu na dvanastniku').
k(3, 'pacient ma varixy na pazeraku').
k(4, 'pacient ma zlatu zilu').
k(5, 'pacient ma vredovity zapal na hrubom creve').
k(6, 'je nezistitelna').

m(2, 'pacient ma krvacanie vredu').
m(4, 'pacient ma krvacanie z hornej casti traviacej trubice').
m(6, 'pacient ma krvacanie z dolnej casti traviacej trubice').
```

```
l(1,  'ma pacient znizenu kyslost zaludocnej stavy').
l(2,  'ma pacient nedostatok zaludocnej stavy').
l(3,  'ma pacient bolesti pri jedle, miznuce po 2-3 hodinach').
l(4,  'ma pacient obcasne bolesti v oblasti zaludku').
l(5,  'ma pacient palenie zahy').
l(6,  'ma pacient pozitivne RTG vysetrenie kontrast. latkou').
l(7,  'ma pàcient ma bolesti 2 - 3 hodiny po jedle').
l(8,  'ma pacient nocne bolesti').
l(9,  'ma pacient zvysenu kyslost zaludocnej stavy').
l(10, 'zvracia pacient krv').
l(11, 'je krv cierna').
l(12, 'ma pacient tazkosti pri prehltnuti potravy').
l(13, 'ma pacient pozitivne endoskopicke vysetrenie').
l(14, 'ma pacient tazkosti pri vylucovani stolice').
l(15, 'ma pacient svrbenie konecnika').
l(16, 'ma pacient krv v stolici').
l(17, 'je krv cerstva').
l(18, 'ma pacient pozitivne rektoskopicke vysetrenie').
l(19, 'ma pacient nutkanie na stolicu').
l(20, 'ma pacient hnacku').
l(21, 'ma pacient primes hlienu v stolici').
```




obr.11 2

Identifikátory uzlov sú celé čísla, pričom sa rozlišujú tri typy uzlov (v zátvorke sú označenia podľa obr. 11.2):

- koreňové meno predikátu k , uzly K1 - K5
- medziľahlé meno predikátu m , uzly M1 - M7
- listové meno predikátu l , uzly L1 - L21

Riešenie:

Hlavný predikát expertného systému bude definovaný takto:

```
expert :- retract(  l(_):-!          ), fail.
expert :- retract(( l(_):-!, fail )), fail.
expert :- retract(  m(_):-!          ), fail.
expert :- retract(( m(_):-!, fail )), fail.
expert :- k(X), k(X,Diag), write('diagnoza: '), write(Diag).
```

Riadenie dialógu bude riešené predikátom

```
l(X) :- l(X,Otazka),
        repeat, write(Otazka), write(' ? '), read(Odpoved),
        (Odpoved = a, asserta(  l(X):-!        ), !        ;
         Odpoved = n, asserta(( l(X):-!,fail )), !, fail ).
```

V prípade nedefinovanej odpovede (rôznej od a alebo n ) sa otázka opakuje.

Vlastný AND-OR graf prepíšeme do PROLOGu pomocou klauzúl

```
k(1) :- m(1), l(3), m(2), l(6).
k(2) :- m(2), l(6), m(3), l(9).
k(3) :- l(6), m(4), l(12), l(13).
k(4) :- l(14), l(15), m(6), l(18).
k(5) :- m(6), m(7), l(21).
k(6).

m(1) :- b(1), ( l(1) ; l(2) ),    e(1), !.
m(2) :- b(2), l(4), l(5), m(4),   e(2), !.
m(3) :- b(3), ( l(7) ; l(8) ),    e(3), !.
m(4) :- b(4), ( l(10) ; m(5) ),   e(4), !.
```

```
m(5) :- b(5), l(16), l(11),        e(5), !.
m(6) :- b(6), l(16), l(17),        e(6), !.
m(7) :- b(7), ( l(19) ; l(20) ),   e(7), !.
```

Pritom ukladanie informácie o vyšetrených medziľahlých uzloch a výpis pomocných diagnóz (textov medziľahlých uzlov, ktoré sú v tejto verzii už nepovinné) realizujú pomocné predikáty:

```
e(X) :- asserta(m(X):-!), (m(X,SubDg), write(SubDg), nl; true).

b(X) :- asserta((m(X):-!,fail)), !.
```

Aj keď uvedený príklad bol veľmi jednoduchý, je potrebné si uvedomiť, že pre obvykle oveľa rozsiahlejší AND-OR graf reálnej úlohy stačí iba doplniť príslušné pravidlá pre jednotlivé uzly. Zväčší sa tak iba rozsah programu, ale nie jeho zložitosť. Pri veľmi rozsiahlych AND-OR grafoch je účelné ešte ďalšie zvýšenie efektívnosti naznačeného postupu dôsledným hierarchickým členením tohoto grafu, optimálnym radením pravidiel a subcieľov v ich telách a použitím ďalších symbolov rezu. Tieto úpravy sú však závislé na konkrétnom okruhu problémov, pre ktoré je ES navrhovaný a na možnostiach použitej implementácie.

Uvedeným spôsobom bol zostavený expertný systém pre diagnostiku bolesti v oblasti hrudníka (26 hypotéz, t.j. diagnóz,180 otázok, 68 medziľahlých uzlov). Pre určenie diagnózy je obvykle potrebné zodpovedať 8-12 otázok, doba odozvy systému je 5-10 sekúnd.

# PRÍLOHA 1
# ŠTANDARDNÉ PREDIKÁTY JAZYKA PROLOG

Nasledujúci prehľad predikátov obsahuje tieto informácie:

1. meno predikátu a jeho argument

2. údaj o tom, či je daný pedikát opätovne splniteľný: keď áno, je za jeho menom <<, keď to závisí na argumente, tak je za jeho menom <

3. keď je predikát štandardne deklarovaný ako operátor, potom je uvedená jeho precedencia a asociativita (platí pre RT-11 PROLOG a pre PROLOG-80, viď ešte popis op)

4. typy argumentov:

   - výstupný argument (povinne neviazaná premenná)

   ? argument môže, ale nemusí byť viazaný

   + vstupný argument (povinne viazaný);

   špeciálne prípady povinného viazania:

   A atóm

   L zoznam

   G platný cieľ

   F platné meno súboru

   - gx0:MENO v RT-11 PROLOGu, pričom súbor MENO musí mať príponu TXT

   - v PROLOG-80 pod operačným systémom CP/M platí konvencia tohoto OS (keď sa neuvedie

prípona, predpokladá sa plg )

- v PROLOG-80 sú platnými menami súborov aj user , buffer

E aritmetický výraz, ktorý môže byť

- celé nazáporné číslo

- v PROLOG-80 reťazec s jedným znakom - "z", pričom je možný aj zápis #z (v obidvoch prípadoch sa berie ASCII kód znaku z )

- v PROLOG-80 globálna premenná (bližšie viď set gvar)

- výraz, tvorený z dielčích aritmetických výrazov E1,E2 pomocou nasledujúcich operátorov:

  - E1 + E2 súčet

  - E1 - E2 rozdiel

  - E1 * E2 súčin

  - E1 / E2 celočíselné delenie

  - E1 mod E2 zbytok po celočíselnom delení E1 / E2

Všetky hodnoty, výsledky a medzivýsledky musia byť v PROLOG-80 a RT-11 PROLOGu z intervalu 0-16383; RT-11 PROLOG nehlási prekročenie týchto hraníc (počíta sa modulo 16384). Štandardný PROLOG pripúšťa aritmetické výrazy iba na pravej strane operátora is a na oboch stranách operátorov $\leq$ , $\geq$ , $=\leq$ ,

>= , =:= a =\= . V našom prehľade sú uvedené označenia E podľa možností PROLOG-80.

5.typ predikátu

| | | | |
|---|---|---|---|
| S | štandardné predikáty podľa Clocksina & Mellisha | | |
| R | -"- | -"- | definované v RT-11 PROLOGu |
| 3 | -"- | -"- | definované vo verziách 3.X PROLOG-80 |
| 4 | -"- | -"- | definované vo verziách 4.X PROLOG-80 |

abort 34RS

Zruši všetky bežiace výpočty a vráti riadenie na hornú úroveň interpretu PROLOGu, uvoľní pamäť, ktorú využívajú jednotlivé ciele. Databáza ostáva nezmenená. Vypíše sa oznam na terminály. V PROLOG-80, verzie 4.X neruší breaky.

ancestor(X,Y) E - 34RS

Unifikuje X -tého predchodcu aktuálneho cieľa s premennou Y. Prvý predchodca je volaním predchodcu samého.

ans R

Výpis posledných desiatich predchodcov vyšetrovaného cieľa pomocou predikátu print.

ans(N) E R

Výpis posledných N predchodcov vyšetroyaného cieľa pomocou predikátu print.

arg(X,Y,Z) E + ? 34RS

X -tý argument v terme Y (povinne štruktúra) je Z . Možno použiť na zistenie objektu, na ktorý je X -tý

argument viazaný, alebo na viazanie ešte neviazanej premennej v terme Y (vytvorenom napr. pomocou predikátu functor ).

array(assert,X,Y,Z) A A E + 34

Vytvorí sa nové jednorozmerné globálne pole s menom X , počtom prvkov Y , pričom všetky prvky sa nastavia na počiatočnú hodnotu Z (celé číslo alebo atóm).

array(get,X,Y,Z) A A E ? 34

Unifikuje sa Y -tý prvok poľa (ktorého meno je X ) so Z . Indexy sa počítajú od nuly. Pri navracaní sa vracia parameter Z do pôvodného stavu. Pole už musí byť vytvorené.

array(put,X,Y,Z) A A E + 34

Y -tý prvok poľa s menom X sa nastaví na hodnotu Z (atóm, alebo celé číslo). Indexy sa počítajú od nuly. Pole už musí existovať.

array(retract,X,Y,Z) A A E ? 34

Zruší sa pole s menom X ; Y a Z nemajú význam, ale Y musí byť viazané na platný celočíselný výraz.

assert(X) + R

To isté ako assertz .

asserta(X) + 34RS

Ukladá klauzulu X do databázy pred ostatné ako predikát.

assertz(X) + 34RS

Ukladá klauzulu X do databázy za ostatné ako predikát.

assertz(X,Y) + L 34

To isté ako assertz(X) , ale mená premenných v zozname Y sú použité v novej klauzule. Y musí byť získané prostredníctvom predikátu read(Z,Y) . Použité napr. v definícii consult .

Poznámka: Všetky štyri predikáty typu assert pri plnení môžu uložiť už existujúcu klauzulu do databázy, ale pri pokuse o opätovné splnenie neuspejú. Keď má byť argumentom X týchto predikátov konjunkcia, je potrebné argument uzavrieť do dvojitých zátvoriek - v opačnom prípade by PROLOG chápal každý člen konjunkcie ako ďalší argument.

atom(X) + 34RS

Cieľ, ktorý uspeje iba ak X je atóm.

atomic(X) ? 34RS

Cieľ, ktorý uspeje iba keď X je atóm, alebo celé číslo.

backtrace 34

Výstup desiatich posledných predchodcov aktuálneho cieľa na obrazovku.

backtrace(N) ? 34

Výstup N posledných predchodcov aktuálneho cieľa na obrazovku.

block(X,Y) ? G 34

Vstup do nového bloku s menom X (splnené vždy). Výstup z bloku sa realizuje buď splnením cieľa Y , jeho nesplnením, alebo predikátom end block . Od verzie 3.9.

break 34RS

Preruší bežiaci výpočet a vytvorí novú kópiu hornej úrovne interpretu PROLOGu. Pre výskok zadaj znak konca súboru, alebo end ako cieľ. Po výstupe z break -u výpočet pokračuje od ďalšieho cieľa za prerušením.

call(X) < G 34RS

Vyvolanie cieľa X .

clause(X,Y) < + ? 34RS

Cieľ uspeje, keď v databáze existuje klauzula s hlavou X a telom Y .

constrain(X,Y) ? ? 4

Keď je X viazaná premenná, vykoná sa cieľ Y (v tomto prípade musí byť Y viazané na nejaký cieľ už pri "volaní" constrain) . Keď je X zatiaľ neviazaná premenná, odloží sa vykonanie cieľa Y na okamžik tesne po viazaní premennej X (do tohoto okamžiku musí byť aj Y viazané na nejaký cieľ !). Pri navracaní sa cieľ spojený s premennou X zruší a pokračuje sa v navracaní. Tento predikát zavádza do PROLOGU kváziparalelné spracovanie. Od verzie 3.9.

consult(X) F 34RS

Otvára súbor X pre vstup a načítava z neho program. Všetky fakty a pravidlá sa pridávajú do databázy a všetky ciele v tvare ?-Cieľ sa vykonajú (bez navracania). Vonkajšie mená premenných sa v PROLOGu-80 uchovávajú pre ďalšie výstupy pomocou listing. Po vykonaní consult aktuálny vstup a výstup ostanú nezmenené a teda consult -y možno do seba vnárať.

copy_file(X,Y) F F 34

Kopíruje súbor X do súboru Y . Po kopírovaní sú oba súbory uzavreté.

core 34R

Vypíše na terminál informáciu o obsadení pamäte, konkrétne rozsah lokálneho a globálneho zásobníka a rozsah voľnej pamäte.

debugging 34RS

Vypíše mená aktuálne činných trasovacích bodov (spypoints).

deny(X,Y) < + ? 34R

Odstráni z databázy tú klauzulu, ktorej hlava je unifikovateľná s X a telo s Y . Pre fakty sa používa telo true . Neviazané premenné sú po odstránení klauzuly z databázy viazané na príslušné objekty.

display(X) ? 34RS

Výstup termu X v prefixovej notácii so všetkými zátvorkami.

edit 34

To isté ako edit(X) , ibaže do vyrovnávacej pamäte editora urobí listing . Slúži k oprave celej databázy PROLOGu. Vhodné pre menšie programy.

edit(X) + 34

Vytvára novú vyrovnávaciu pamäť editora, urobí listing(X) do nej a vyvolá obrazovkový editor. Po výstupe z editora obsah vyrovnávacej pamäte nahradí v databáze tie klauzuly, ktoré do editora vstúpili. Automaticky sa vyvolá old buffer k obnoveniu pôvodného stavu vyrovnávacej pamäte. Hodí sa pre menšie opravy v databáze a pre prípady keď je program príliš rozsiahly a nie je možné uschovať súčasne v pamäti aj príslušný zdrojový text a robiť opravy pomocou edit buffer .

edit_buffer 34

Vyvoláva obrazovkový editor PROLOG-80. Aktuálny obsah vyrovnávacej pamäte editora (má v PROLOGu meno buffer ) možno meniť pomocou príslušných funkčných kláves (viď implementačnú príručku pre príslušný počítač).

end_block(X) G 34

Výstup z najbližšieho bloku, ktorého meno je unifikovateľné s X . Je možné odovzdávať informácie von z bloku vďaka unifikácii. Všetky lokálne alternatívy vnútri bloku sa zrušia (ako u rezu). Od verzie 3.9.

fail 34RS

Cieľ, ktorý nikdy neuspeje a vyvoláva navracanie.

functor(X,Y,Z) + - - / ? A E 34RS

X je term (povinne štruktúra), ktorej funktor je Y a árnosť je Z . Obvyklé použitie je +-- (určenie funktora a árnosti) a -AE (konštrukcia termu s funktorom Y a so Z neviazanými argumentami).

get(X) ? / E 34RS

To isté ako get0, iba ignoruje riadiace (neviditeľné) znaky.

get_bin(X) ? 4

Z aktuálneho vstupu sa prečíta jeden byte bez akejkoľvek transformácie a unifikuje sa s X . Koniec súboru sa odovzdá ako číslo 256.

get_key(X) ? 3

Otestuje stav klávesnice. Keď nie je stlačená žiadna klávesa, vyvolá sa navracanie. V opačnom prípade sa unifikuje ASCII kód stlačenej klávesy (celé číslo) s X (vo verziách 4.X realizuje analogickú činnosť is key ).

get0(X) ? / E 34RS

Číta ďalší znak z aktuálneho vstupu a unifikuje jeho ASCII hodnotu s X . Predikát get0 dáva control-z (26) keď narazí na koniec súboru, pre znak konca riadku dáva CR (13).

goto_xy(X,Y) E E 4

Adresácia kurzora. Ďalší znak sa zapíše do X -tého stĺpca Y -tého riadku.

halt 34

Výskok zo systému PROLOG-80 do monitora. Je ekvivalentný zadaniu koncového znaku súboru na hornej úrovni interpreta PROLOGu (mimo rámca break ).

integer(X) ? 34RS

Cieľ, ktorý uspeje iba ak X je celé číslo. Rozsah čísiel v PROLOG-80 a v RT11-PROLOG je 0-16383.

X is Y 40,xfx ? E 34RS

X sa unifikuje s hodnotou výrazu Y .

is_key(X) ? 4

Test stavu klávesnice. Keď nie je stlačená žiadna klávesa, vyvoláva navracanie. Inak unifikuje ASCII kód klávesy s X , nezávisle na nastavení aktuálneho vstupu.

length(X,Y) L - / L E 34RS

Y je dĺžka zoznamu X.

listing 3

Vypíše deklarácie všetkých operátorov a všetky klauzuly z databázy na aktuálny výstup.

listing 4

Vypíše všetky klauzuly z databázy na aktuálny výstup.

listing(X) A / L 34RS

Vypíše všetky klauzuly s atómom X ako predikátom, resp. všetky klauzuly, ktorých mená tvoria zoznam X .

listing(op) + 4

Vypíše všetky aktuálne definície operátorov podľa priorít.

listing(X,Y) L F / A F 34RS

Uskutočňuje listing(X) s výstupom do súboru Y . Potom súbor uzavrie. Aktuálny výstup sa nemení.

listing('',X) F 34

Vypíše všetky klauzuly z databázy na aktuálny výstup. Ak sú známe vonkajšie mená premenných tak tieto sa zachovávajú.

lc 34

Nastavenie režimu lc , v ktorom premenné začínajú veľkým a atómy malým písmenom. Po zavedení systému PROLOG-80 sa tento režim nastaví automaticky.

name(X,Y) A - / A L / - L 4RS

Y je zoznam hodnôt ASCII kódov znakov, z ktorých pozostáva meno atómu X . Je možné použiť ho pre tvorbu mena atómu, resp. pre rozklad mena atómu.

name(X,Y) A - / A + / - + 3

Funguje rovnako ako name string vo verziách 4.X PROLOG-80.

name_hash(X,Y) A ? / - E 34

Y je hash-kód (0-1023) atómu X. Ak pri volani typu -E atóm s príslušným kódom ešte neexistuje, vyvolá sa navracanie.

name_string(X,Y) A - / A + / - + 4

Funguje podobne ako name , ale miesto zoznamu je v druhom argumente reťazec.

new_buffer 34

Otvára novú prázdnu vyrovnávaciu pamäť editora. Stará vyrovnávacia pamäť sa uschováva pre pozdejšie použitie (údaje a smerníky sa zachovávajú). Organizácia odpamätávaných vyrovnávacích pamäti je zásobníková).

new_program 34

Odstráni všetky používateľské klauzuly z databázy.

nl 34RS

Na aktuálnom výstupe sa vypíše nový riadok.

nodebug 34RS

Odstraňuje všetky trasovacie body (spypoints).

NOLC 34

Nastaví sa režim NOLC v ktorom sa menia automaticky veľké písmená na malé. Premenné môžu začínať iba podčiarnikom.

nonvar (X) ? 34RS

Cieľ, ktorý uspeje iba ak X je viazané.

nospy(X) 250,fx A / A(E) / L 3RS

Odstraňuje všetky trasovacie body (spypoints) z predikátov X , kde X je špecifikácia predikátu, alebo ich zoznam, viď spy .

nospy(X) 250,fx A / A/E / L 4

Rovnaké, ako nospy pre verzie 3.X, iba árnosť sa oddeľuje od mena predikátu lomítkom a nie zátvorkami.

not(X) 60, fx G 34RS

Cieľ, ktorý uspeje vtedy a len vtedy, keď X neuspeje. Je ekvivalentné definícii not(X):-X,!,fail;true. Nemožno ho použiť samotný na generovanie alternatív. Keď je argument X konjunkcia subcieľov, potom je nutné ju uzavrieť do dvojitých zátvoriek, aby jednotlivé členy konjunkcie nechápal PROLOG ako samostatné argumenty predikátu not .

notrace 34RS

Vypína úplné trasovanie.

old_buffer 34

Zničí obsah aktuálnej vyrovnávacej pamäte a rekonštruuje situáciu pred posledným volaním predikátu new buffer. všetky údaje a smerníky zo starej vyrovnávacej pamäte sú znovu prístupné. Keď zásobník vyrovnávacej pamäte je prázdny, vytvára sa nová vyrovnávacia pamäť.

op(X,Y,Z) E A A 34RS

Definuje nový operátor s menom Z a s precedenciou X typu Y . Čím vyššia je precedencia, tým slabšie je operátor viazaný so svojim operandom ( ± má tak vyššiu precedenciu ako ± ). Precedencia musi byť z intervalu 2-255 (PROLOG-80 a RT11-PROLOG). Neodporúča sa voliť precedencie líšiace sa iba o jednotku. Typy Y môžu byť:

fx prefixový operátor

xf postfixový operátor

xfx všeobecný infixný operátor

yfx vľavo asociativny infixný operátor
(napr. a+b+c = (a+b)+c )

xfy vpravo asociativny infixný operátor
(napr. a.b.c = a.(b.c) )

Pre zrušenie definicie operátora stačí zadať op(1,xfx,meno) ako cieľ. Definície štandardných operátorov (PROLOG-80 a RT-11 PROLOG):

```
255   xfx   :-
255   fx    ?-
254   xfy   ;
253   xfy   ,
250   fx    spy  nospy
 60   fx    not
 51   xfy   .
 40   xfx   is  =..  =:=  =\=  =  \=  <  =<  >=  >  ==  \==
 31   fx    ¤
 31   yfx   +  -
 21   yfx   *  /
 11   xfx   mod
  9   xfy   \  (iba v PROLOG-80)
```

portray(X) + 3RS

Definíciu predikátu musí navrhnúť používateľ. Používa sa k predpísaniu upného formátu pre print (pozri tam).

print(X) ? 3RS

Píše term X predpísaným spôsobom na aktuálny výstup. Ak užívateľ zadal klauzuly definujúce predikát portray(X), potom sú tieto použité pre výstup. V opačnom prípade sa X bude tlačiť pomocou predikátu write. Predikát print vypisuje všetky oznamy systému a správy pri ladení.

put(X) E 34RS

Znak s ASCII kódom X sa vypíše na aktuálny výstup.

put_bin(X) E 4

Do aktuálneho výstupného súboru sa zapíše jeden byte.

read(X) ? 34RS

Číta term X ukončený bodkou a novým riadkom (alebo medzerou) z aktuálneho vstupu. Ak narazí na koncový znak súboru, X sa viaže na term ?-end.

read(X,Y) ? - 34

To isté ako read(X), ale vonkajšie mená premenných vystupujúcich v X vytvoria zoznam Y. Používa sa najmä v kombinácii s assertz(X,Y).

reconsult(X) F 34RS

To isté ako consult , ale pred zaradením novej klauzuly do databázy sa z nej vyradia všetky klauzuly s rovnakým menom a árnosťou. Používa sa na robenie zmien v programoch. Reconsulty nie je možné vnárať (ale ho možno volať v consult a on môže zase volať consult ).

repeat << 34RS

Cieľ, ktorý uspeje ľubovoľne veľa krát.

restart 34R

Je to ekvivalent k príkazu abort , ale na terminál sa nezobrazí žiadna správa. V PROLOG-80 ruší oproti abort naviac aj break -y.

retract(X) < + 34RS

Odstraňuje klauzulu unifikovateľnú s X z databázy. Všetky neviazané premenné v X sú po odstránení X z databázy viazané na príslušné objekty odstránenej klauzuly.

retractall(X) + 34R

Odstraňuje všetky klauzuly, ktorých hlavy sú unifikovateľné s X . Telá klauzúl sú z hľadiska unifikácie nepodstatné. Žiadne viazanie premenných sa nerealizuje.

see(X) F 34RS

Prepína aktuálny vstup na súbor X . Ak X je už otvorené pre čítanie, pokračuje sa v jeho čítaní, v opačnom prípade spôsobí predikát otvorenie súboru X .

seeing(X) - / F 34RS

Meno aktuálneho súboru sa unifikuje s X .

seen 34RS

Uzavrie aktuálny vstupný súbor a uvoľní vyrovnávaciu pamäť, ktorú tento súbor používal. Nastaví aktuálny vstup na user . Pri uzavrení súboru buffer pre vstup sa potom smernik pre čítanie nastaví nastaví na začiatok vyrovnávacej pamäte editora, takže je možné čítať údaje znova od začiatku.

set_gvar(X,Y) A E 34

Nastavenie hodnoty globálnej premennej s menom X na hodnotu Y (predikát nepodlieha mechanizmu navracania a pôvodná hodnota premennej je stratená). Globálnu premennú je možné zrušiť predikátom retract(X) , nakoľko sa uchováva v databáze (ovšem špeciálnym spôsobom). Meno globálnej premennej sa môže vyskytovať v aritmetických výrazoch, kde sa za ňu vždy dosadí aktuálna hodnota.

skip(X) E 34RS

Prečíta aspoň jeden znak z aktuálneho vstupu a pokračuje v čítani až po prečítanie znaku s kódom X .

space 34

Na aktuálny výstup sa vypíše znak medzera.

spy(X) 250,fx A / A(E) / L 3RS

Umiestni trasovacie body (spypoints) na predikáty definované v X , kde X je buď špecifikácia predikátu (meno, alebo meno(árnosť) ), alebo zoznam takýchto špecifikácií. Napriklad f(3) určuje predikát s menom f a s tromi argumentami. Ak je trasovací bod (spypoint) umiestnený na nejakom predikáte, potom systém vypiše informáciu stále keď sa vyvolá tento predikát a užívateľ môže ovplyvniť ďalšie riešenie: pokračovanie je vyvolané znakom nového riadku; prehľad ďalších možností vypíše systém po zadaní znaku h . Pre PROLOG-80, verzie 4.X sú možné tieto voľby:

c - creep , pokračuj vo výpočte a trasuj do prichodu na ďalšiu ovládanú bránu

s - skip , potlač všetky ladiace výpisy do prichodu na ďalšiu bránu tohoto cieľa

l - leap , potlač trasovanie do príchodu na trasovací bod, alebo na bránu tohoto cieľa

r - retry , tento cieľ sa začne splňovať znova od brány Call

; - or , na bráne Exit sa zamietne nájdené riešenie a prejde sa ihneď na Redo

f - fail , cieľ je neúspešný - okamžite sa prejde na bránu Fail

<NL> - pokračovanie bezo zmien ladiacej informácie

g - goal , prečíta sa jeden cieľ z klávesnice, vykoná sa a čaká sa na ďalší pokyn

a - abort , aktuálny výpočet sa ukončí a riadenie sa odovzdá na hornú úroveň interpretu

spy(X) 250,fx A / A/E / L 4

To isté, ako spy pre verzie 3.X, iba árnosť sa od mena predikátu oddeľuje lomítkom (napr. namiesto f(3) bude f/3 ).

subgoal_of(x) < ? 4

Unifikuje X postupne (pri navracaní) so všetkými predchodcami aktuálneho cieľa. Vďaka unifikácii je možné odovzdať informácie oboma smermi, ale pri navracaní sa všetky väzby normálne zrušia s uvažovaním miesta, kde väzba vznikla.

tab(X) E 34RS

Tlačí X medzier na aktuálny výstup.

tell(X) F 34RS

Prepína aktuálny výstup na súbor X . Ak X je už otvorené pre zápis, pokračuje sa v zápise, v opačnom prípade otvára X pre výstup.

telling(X) - / F 34RS

Unifikuje meno aktuálneho výstupného súboru s X .

told 34RS

Uzavrie aktuálny výstupný súbor a uvoľní vyrovnávaciu pamäť, ktorú súbor používal. Nastaví aktuálny výstup na user . Bez korektného ukončenia súboru ho nemožno v ďalšom čítať. Ak

súbor buffer je uzavretý pre výstup, je ešte stále prístupný pre vstup. Nasledujúce vstupné údaje (prvý print, write, atď.) vymažú celú vyrovnávaciu pamäť a vytvárajú ju znova od začiatku.

trace 34RS

Zapína úplné trasovanie.

trimcore 34R

Uvoľňuje nevyužitú oblasť pamäti. Vyvoláva sa automaticky vždy, keď sa objaví nápoveď ?- .

true 34RS

Cieľ, ktorý uspeje vždy.

var(X) ? 34RS

Cieľ, ktorý uspeje iba ak X je neviazaná premenná.

wait_key(X) 4

Čaká na stlačenie klávesy a potom unifikuje jej kód s X .

write(X) ? 34RS

Vypíše term X na aktuálny výstup a zohľadní deklarácie operátorov. Neviazané premenné sa vypisujú v tvare číslo , kde každej premennej v rámci jedného cieľa write je priradené iné číslo (v PROLOG-80 sa vypisujú pôvodné mená premenných, zdieľaných s premennými v rámci aktuálnej otázky). Tento výpis nie je možné vo všeobecnosti prečítať pomocou read.

write(X,Y) ? L 4

Ako write , ale pre mená premenných sa použije zoznam Y .

X < G 34RS

Cieľ daný viazaním premennej X . Zhodné s call(X) .

X,Y < 253,xfy G G 34RS

Cieľ je splnený ak uspeje X a po ňom aj Y (konjunkcia).

X;Y << 254,xfy G G 34RS

Cieľ uspeje, ak uspeje X , alebo Y (disjunkcia).

X<Y 40,xfx E E 34RS

Cieľ, ktorý uspeje iba ak hodnota X je menšia ako hodnota Y .

X=Y 40,xfx ? ? 34RS

Unifikuje X a Y ak je to možné.

X=..Y 40,xfx + L / + - / - L 34RS

Y je zoznam, pozostávajúci z funktora termu X , za ktorým nasledujú jeho argumenty. Je to možné použiť pre rozklad termu, aj pre jeho zostavenie. Je menej efektívny ako arg a functor .

X=:=Y 40,xfx E E 3RS

Cieľ, ktorý uspeje iba ak hodnoty X a Y sú si rovné.

X=<Y 40,xfx E E 34RS

Cieľ, ktorý uspeje iba ak hodnota X je menšia, alebo rovná ako hodnota Y .

X==Y 40,xfx ? ? 34RS

Cieľ, ktorý je splnený iba ak X a Y sú identické. Dva termy sú identické, ak sú inifikovateľné a ak všetky neviazané premenné vystupujúce v X a Y odpovedajú tej istej neviazanej premennej. V priebehu testu identity sa žiadne premenné neviažu.

X=\=Y 40,xfx E E 3RS

Cieľ, ktorý uspeje iba ak hodnoty X a Y nie sú si rovné.

X>Y 40,xfx E E 34RS

Cieľ, ktorý uspeje iba ak hodnota X je väčšia ako hodnota Y .

X>=Y 40,xfx E E 34RS

Cieľ, ktorý uspeje iba ak hodnota X je väčšia, alebo rovná ako hodnota Y .

X\=Y 40,xfx + + 34RS

Cieľ, ktorý uspeje iba ak X a Y nie sú unifikovateľné. Keby aspoň jeden z argumentov bola neviazaná premenná, tento cieľ by bol stále neúspešný. Korektnejšia definícia nerovnosti je možná s použitím constrain :

```
dif(X,Y) :- constrain(X, constrain(Y, X \= Y) ).
```

X\==Y 40,xfx + + 34RS

Cieľ, ktorý uspeje iba ak X a Y nie sú identické.

! (cut symbol, symbol rezu) 34RS

Je to cieľ, ktorý pri prvom volani vždy uspeje. Ak sa naň však narazí pri navracaní (backtracking), spôsobí nesplnenie nadradeného cieľa (cieľa v hlave pravidla, v tele ktorého je cut symbol). Túto informáciu odovzdá nadradenému cieľu, pričom spôsobí ignorovanie ďalších alternatív (t.j. ďalších pravidiel s tou istou hlavou).

[X] + 34RS

Prvky zoznamu sú mená súborov. Keď pred menom súboru je znak ~, potom sa vykoná reconsult , v opačnom prípade consult ; na terminály sa po ukončení objaví oznam.

'@env'(X,Y,Z) E ? ? 34

Predikát umožňuje užívateľovi pracovať so systémovými premennými. X je poradové číslo systémovej premennej, jej aktuálna hodnota sa unifikuje s Y a potom sa nahradí hodnotou Z . Užívateľovi sa nedoporúča meniť hodnoty systémových premenných s výnimkou 6,9,10 a 11. Význam jednotlivých premenných je vo verziách 4.X nasledovný:

0 - stav trasovania (0-netrasuje sa, 1-trasuje sa)

1 - pre ladenie - leap (číslo cieľa, alebo 0)

2 - pre ladenie - číslo cieľa

3 - pre ladenie(nenulová hodnota potlačí všetky informácie pre ladenie) - je potrebné použiť napr. pri definícii print

4 - pre ladenie - číslo cieľa, kam sa robí skip , alebo 0.

5 - pre ladenie - meno akcie, ktorá sa má uskutočniť ako ďalšia, alebo 0

6 - pre systémový výstup - meno predikátu, ktorý sa použije vnútri systému pre výpis cieľov, hodnôt premenných atď. Pokiaľ takýto cieľ neuspeje, urobí sa write (robí sa takto náhrada za print , ktorý v PROLOG-80 od verzie 4.0 nie je štandardne definovaný)

7 - zvolil užívateľ trasovanie? 0-nie, 1-ano

8 - hĺbka vnorenia pre tabeláciu

9 - leash - číslo 0 - 15, jednotlivé bity indikujú, či sa jedná o bránu ovládanú (0 - riadenie sa odovzdá používateľovi), alebo o bránu neovládanú (1); poradie priradenia bitov je CALL-3, EXIT-2, REDO-1, FAIL-0. Pri štarte systému je 0 (všetky brány sú ovládané), praktické nastavenie je 3 (systém zastavuje iba na ovládaných bránach CALL a EXIT)

10- hĺbka výpisu pre write (pri štarte je 20)

11- meno súboru, do ktorého je smerovaný výstup trasovania

'@translate'(X,Y) + + 4

Definíciu zadáva používateľ. Používa sa v preprocesoroch - keď tento predikát uspeje, pri consult sa do databázy PROLOGu uloží namiesto X klauzula Y .

# PRÍLOHA 2
# SLOVNÍK PROLOGOVSKÝCH VÝRAZOV

V slovníku sú uvedené najčastejšie používané prologovské pojmy v angličtine podľa [3] a príslušné slovenské ekvivalenty, používané v našom učebnom texte. Snažili sme sa zachovať terminológiu podľa českého prekladu [3], ale v niekoľko málo prípadoch sme sa od nej odchýlili.

Anglicky písaná prologovská literatúra je pomerne jednotná, výnimku tvorí microPROLOG, v ktorom aj také pojmy, ako atóm majú odlišný význam.

Uvádzame väčšinou nie samostatné slová, ale celé spojenia tak, ako sa v opisoch najčastejšie vyskytujú. Pojmy bežné vo výpočtovej technike sme neuvádzali s výnimkou niekoľkých, ktoré sú z hľadiska PROLOGu podstatné. Hviezdičkou sú označené alternatívne termíny, vyskytujúce sa v českej a slovenskej prologovskej literatúre. Mená štandardných predikátov, názvy brán blokových modelov a mená trasovacích opcií sa obvykle neprekladajú, preto sme ich v slovníku ani neuvádzali. Nami používaná terminológia sa čiastočne líši od terminológie matematickej logiky (viazaná premenná, unifikácia).

| | | |
|---|---|---|
| ancestor | predchodca | предшественник |
| anonymous (variable) | anonymná (premenná) | анонимная /переменная/ |
| associativity (of an operator) | asociativita (operátora) | ассоциативность /оператора/ |
| atom | atóm | атом |
| backtracking | navracanie * návrat | возврат |
| body (of a rule) | telo (pravidla) | тело /правила/ |

| | | |
|---|---|---|
| box model | blokový model<br>* model skriniek | модель трассировки |
| built-in<br>(predicate) | štandardný<br>(predikát)<br>* zabudovaný p.<br>* vstavaný p. | встроенный<br>/предикат/ |
| clause | klauzula | дизъюнкт<br>* утверждение |
| cut-symbol | symbol rezu<br>* rez | отсечение |
| empty (list) | prázdny (zoznam) | пустой /список/ |
| exhaustive<br>(tracing) | podrobné<br>(trasovanie)<br>* úplné t. | полная<br>/трассировка/ |
| fact | fakt | факт |
| fail | neúspech | неудача |
| to fail | neuspieť<br>* zlyhať<br>* byť neúspešný | заканчиваться<br>неудачей |
| functor | funktor | функтор |
| goal | cieľ | цель<br>* утверждение |
| head (of a rule) | hlava (pravidla)<br>* záhlavie p. | заголовка правила |
| head (of a list) | hlava (zoznamu) | голова /списка/ |
| to instantiate<br>(a variable to) | viazať<br>(premennú na)<br>* inštalovať<br>* špecifikovať | конкретизировать<br>/переменную чем нибудь/<br>* присвоить что нибудь<br>переменной |

| | | |
|---|---|---|
| | * substituovať<br>* konkretizovať | |
| instantiated<br>(variable) | viazaná<br>(premenná)<br>* viď viazať | конкретизированная<br>/переменная/ |
| list | zoznam | список |
| matching | unifikácia<br>* srovnávání<br>* mečování<br>* pasování<br>* zladenie<br>* zhoda<br>* ztotožnenie | согласование<br>ж сопоставление<br>ж соответствие |
| to match | unifikovať<br>* viď unifikácia | соответствовать |
| member (of a list) | prvok (zoznamu)<br>* člen z. | элемент /списка/ |
| operator | operátor | оператор |
| port<br>(of a box model) | brána<br>(blokového modelu) | событие<br>/в модели трассировки/ |
| precedence<br>(of an operator) | precedencia<br>(operátora)<br>* priorita | приоритет<br>/оператора/ |
| predicate | predikát | предикат |
| to re-satisfy | opätovne splniť | передоказать |
| rule | pravidlo | правило |
| to satisfy<br>(a goal) | splniť (cieľ)<br>* splňovať | доказать согласованность<br>/утверждения/<br>выполнить /утверждение/ |

| | | |
|---|---|---|
| to share | zdieľať<br>* združiť | сцеплять |
| spy-point | bod zastavenia<br>* ladiaci bod<br>* trasovací bod<br>* testovací bod | контрольные точки |
| subgoal | subcieľ<br>* podcieľ | субцель |
| to succeed | uspieť<br>* byť úspešný | согласоваться с базой данных<br>* выполняться |
| tail (of a list) | telo (zoznamu) | хвост /списка/ |
| term | term | терм |
| uninstantiated (variable) | neviazaná (premenná)<br>* voľná<br>* viď viazaná p. | неконкретизированная /переменная/<br>* неопределенная п. |

# PRÍLOHA 3
## Tabuľka ASCII-kódov znakov

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 32 | SP | 48 | 0 | 58 | : | 65 | A | 91 | [ | 97 | a | 123 | { |
| 33 | ! | 49 | 1 | 59 | ; | 66 | B | 92 | \ | 98 | b | 124 | \| |
| 34 | " | 50 | 2 | 60 | < | 67 | C | 93 | ] | 99 | c | 125 | } |
| 35 | # | 51 | 3 | 61 | = | 68 | D | 94 | ^ | 100 | d | 126 | ~ |
| 36 | ¤ | 52 | 4 | 62 | > | 69 | E | 95 | _ | 101 | e | 127 | DEL |
| 37 | % | 53 | 5 | 63 | ? | 70 | F | 96 | ` | 102 | f | | |
| 38 | & | 54 | 6 | 64 | @ | 71 | G | | | 103 | g | | |
| 39 | ' | 55 | 7 | | | 72 | H | | | 104 | h | | |
| 40 | ( | 56 | 8 | | | 73 | I | | | 105 | i | | |
| 41 | ) | 57 | 9 | | | 74 | J | | | 106 | j | | |
| 42 | * | | | | | 75 | K | | | 107 | k | | |
| 43 | + | | | | | 76 | L | | | 108 | l | | |
| 44 | , | | | | | 77 | M | | | 109 | m | | |
| 45 | - | | | | | 78 | N | | | 110 | n | | |
| 46 | . | | | | | 79 | O | | | 111 | o | | |
| 47 | / | | | | | 80 | P | | | 112 | p | | |
| | | | | | | 81 | Q | | | 113 | q | | |
| | | | | | | 82 | R | | | 114 | r | | |
| | | | | | | 83 | S | | | 115 | s | | |
| | | | | | | 84 | T | | | 116 | t | | |
| | | | | | | 85 | U | | | 117 | u | | |
| | | | | | | 86 | V | | | 118 | v | | |
| | | | | | | 87 | W | | | 119 | w | | |
| | | | | | | 88 | X | | | 120 | x | | |
| | | | | | | 89 | Y | | | 121 | y | | |
| | | | | | | 90 | Z | | | 122 | z | | |

Pozn. Na počítači ZX SPECTRUM odpovedá kódu 96 "libra" (znak anglickej meny) a kódu 127 "copyright" (C v krúžku).

# PRÍLOHA 4
# PROLOG-80 PRE ZX SPECTRUM

## P4.1 DIALÓG SO SYSTÉMOM

Po zavedení programu sa tento prihlási hlavičkou

```
Prolog-80 version 3.8 (c) 1985
 PeNoSoft
?-
```

a v prípade verzie 4.1 hlavičkou

```
Prolog-80 VI4.1. 1987 PeNoSoft.
?-
```

V pracovných exemplároch verzie 4.1 sa nastaví čierne pozadie, ktoré sa ovšem postupne prepisuje (najjednoduchšie vyvolaním obrazovkového editora).

Vlastný dialóg prebieha v oboch verziách tak, ako to opisuje kapitola 1. Klávesou nového riadku <NL> je v tomto prípade klávesa <ENTER> a znak konca súboru <EOF> sa vyšle kombináciou kláves <SS+i> (SYMBOL SHIFT + i).

Vo verzii 4.1 sa po vyžiadaní ďalšej alternatívy bodkočiarkou nezadáva ešte aj <ENTER>, ako tomu je vo verzii 3.8 (a v kapitole 1).

## P4.2 OBMEDZENIA PROLOG - 80

Všetky obmedzenia implementácie PROLOG-80 sú dané snahou o úspornú vnútornú reprezentáciu objektov jazyka. Porovnanie s microPROLOGom britskej firmy Programming Association, rozšíreným na počítačoch Sinclair ZX Spectrum hovorí jednoznačne v prospech autorov PROLUG-80. Mnohé z obmedzení užívateľ bežne ani nepocíti - obvykle narazí skôr na nedostatok pamäťového priestoru. Pri štarte systému PROLOG-80 má užívateľ k dispozícii 16442 byte.

Implementačné obmedzenia PROLOG-80 sú nasledovné:

-počet rôznych mien (textových konštánt, mien predikátov, mien premenných) pri jednom vyvolaní systému je maximálne 1024. Niektoré využíva systém samotný, takže užívateľ má k dispozícii niečo cez 900 mien; pritom raz už definované mená možno odstrániť zo slovníka iba znovuzavedením systému

-meno(atóm) môže pozostávať maximálne z 253 znakov

-pracuje sa iba s celými nezápornými číslami z intervalu <0,16383> (nakoľko PROLOG nie je určený na numerické výpočty toto obmedzenie nie je podstatné); prekročenie tohoto intervalu sa poklauá za chybu a systém o tom podá správu

-maximálny počet argumentov štruktúry (teda aj predikátu) je 15 (argumenty môžu byť ďalej štruktúrované, alebo je možné použiť zoznam, kde počet prvkov je obmedzený iba kapacitou pamäti)

-pri čítaní termu môže tento obsahovať maximálne 32 rôznych premenných (vrátane anonymných)

-v jednej klauzule v databáze smie byť maximálne 31 premenných, ktoré sa v nej vyskytujú aspoň dvakrát

-neexistujú operátory typu <u>yf</u> a <u>fy</u>

-rozsah precedencie operátorov je <1,255>, pričom každý operátor má ľavú, alebo pravú asociativitu; neexistencia operátora je vyjadrená precedenciou 0; asociativita je riešená odčítaním jedničky u <u>x</u> operandov, preto sa neodporúča používať precedenciu 1 a precedencie líšiace sa navzájom iba o jedničku; každý operátor môže mať iba jednu aktuálnu deklaráciu (nemožno napr. definovať znak - súčasne ako unárne aj binárne mínus) je ale možné meniť v priebehu činnosti systému deklarácie operátorov (aj štandardných)

-PROLOG-80 nie je stavaný na spracovanie cyklických štruktúr a je radno sa im v tejto implementácii vyhýbať, aj keď v niektorých špeciálnych prípadoch ich spracovanie prebehne úspešne

## P4.3 ROZŠÍRENIA PROLOG - 80

Autori PROLOG-80 implementovali niekoľko veľmi elegantných rozšírení oproti štandartnej verzii jazyka:

RAM súbory - pod spoločným menom buffer je možné vytvoriť niekoľko pracovných textových súborov, tvoriacich zásobník (prístupný je teda iba posledný, nad ním môže pracovať obrazovkový editor)

globálne premenné - hodia sa na efektívne uchovávanie informácií / databáze (napr. počítadlá), viď predikát set gvar

polia - použitím poľa namiesto mnohých faktov s rovnakým menom možno ušetriť veľa pamäti; prvkom poľa môže byť ovšem iba číslo alebo atom; viď predikát array

reťazce - štruktúra podobná zoznamu; na oddelenie hlavy od tela sa používa znak \ ; prázdny reťazec sa označuje: "" a reťazec s jedným znakom sa dá zapísať pomocou # :

```
?- "abcd" = H\T.<ENTER>
H = 97 ,                    /* 97 - ASCII kód písmena a  */
T = "bcd" <ENTER>
yes

?- A = #a.<ENTER>
A = 97 <ENTER>
yes
```

Vo verzii 4.1 sú k dispozícii ďalšie rozšírenia:

mechanizmus "zmrazovania" cieľov - inšpirovaný implementáciou PROLOG II; umožňuje niekoľkonásobne urýchliť programy typu generuj & testuj ; viď predikát constrain

bloky - umožňujú elegantne ošetriť výnimočné situácie (inšpirované PROLOGom II a niektorými implementáciami LISPu - "throw" a "catch"); viď predikáty block a end block

<u>prístup k predkom</u> - prostredníctvom predikátu <u>subgoal of</u> je možné odovzdávať informácie cez niekoľko úrovní volania (a šetriť tým pamäť) a zisťovať kontext volania

<u>okná</u> - pre výstup na obrazovku; je ich osem (0-normálny výstup, 6-obrazovkový editor, 7-informácie o práci s magnetofónom v dolnom riadku); používateľ si môže predefinovať tieto okná, alebo si môže nadefinovať svoje vlastné (pomocou riadiacich kódov); výstup sa do okien presmeruje buď riadiacim kódom, alebo nastavením výstupného prúdu do súboru s číslom okna ako fiktívnym menom; výpisy v danom okne roľujú

## P4.4 POSTUP PRI PÍSANÍ A OPRAVOVANÍ PROGRAMU

### P4.4.1 Vytvorenie nového programu

Nový program je výhodné písať do RAM súboru s menom <u>buffer</u> pomocou obrazovkového editora (funkcie editora a príslušné klávesy sú uvedené v tabuľke) a po odladení uložiť na MGP. Typický dialóg vyzerá schematicky takto:

| | |
|---|---|
| ?- edit_buffer. | volanie obrazovkového editora |
| ... | objaví sa prázdna obrazovka |
| nový program | píšeme nový program |
| ... | |
| <SS i> | ukončenie práce editora |
| yes | |
| | |
| ?- [buffer]. | zavedenie programu do DB PROLOGu |
| (Syntax error ... ) | výpis chýb, ak sú nejaké |
| buffer consulted | |
| yes | |
| | |
| ?-edit_buffer. | volanie obrazovkovkového editora |
| ... | objaví sa zdrojový text programu |
| opravovaný program | opravujeme program |
| ... | |
| <SS i> | ukončenie práce editora |
| yes | |

```
?- [buffer].                    prepis obsahu DB PROLOGu opraveným
buffer reconsulted              programom
yes

?- copy_file(buffer,menosub).   zápis na MGP
yes
```

P4.4.2 Súbory na magnetickej páske

Údaje sa zapisujú na MGP do súborov v zadanom formáte (rôznom pre verzie 3.8 a 4.1) a v tomto formáte sa aj načítavajú. Súbory sú členené do blokov konštantnej dlžky, v každom bloku je uložená okrem údajov aj informácia o mene súboru a čísle bloku. To umožňuje opakované čítanie chybne načítaného bloku.

Pri zápise na MGP sa objaví v dolnom riadku výpis (verzia 3.8):

S menosuboru CC

kde CC je číslo práve zapisovaného bloku. Pri čítaní sa objaví analogický výpis

L menosuboru CC

v prípade, že sa načítava iný súbor, resp. iný blok načítavaného súboru, než ktorý má nasledovať, v pravej časti riadku sa objaví o tom informácia, napr.:

S menosuboru CC See inemenosub FF

kde FF je číslo práve načítavaného bloku. Na tomto mieste sa objaví aj chybové hlásenie Read error , ktoré signalizuje chybu pri čítaní.

Vo verzii 4.1 je formát týchto informácii trochu odlišný:

S>menosuboru.PLG 0
priprav kazetu

po štarte (stlačenie ľubovoľnej klávesy) druhý riadok výpisu zmizne a vpravo sa vypisuje vždy číslo aktuálneho bloku. V prípade čítania súboru sa namiesto S vypíše L .

P4.4.3 Práca s programom, ktorý už je na MGP

Keď je potrebné robiť opravy v programe, potom je najlepšie ho nahrať do RAM súboru s menom buffer a opraviť ho pomocou editora:

```
?- copy_file(menosub,buffer).   kopírovanie z MGP do buffer
yes

?- edit_buffer.                 volanie editora
...                             objaví sa text programu
opravovaný program              oprava chybného programu
...
<SS i>                          ukončenie práce editora
yes

?- [buffer].                    zavedenie programu do DB PROLOGu
buffer consulted                kvôli kontrole syntaktickej
yes                             správnosti

?- copy_file(buffer,menosub).   uloženie opraveného programu do
yes                             súboru na MGP
```

Keď chceme iba spustiť prologovský program, uložený na MGP, možno ho uložiť priamo do databázy PROLOGu:

```
?- [menosub].
menosub consulted
yes
```

Uvedeným spôsobom sa program pridáva k obsahu DB. Keď je obava, že by mohlo dôjsť k zdvojeniu klauzúl (v prípade, že DB nie je prázdna) je lepšie použiť [¤menosub]. Keď je program uložený vo viacerých súboroch, potom ich možno nahrať jediným "príkazom":

```
?-  [sub1,*sub2,sub3].
sub1 consulted
sub2 reconsulted
sub3 consulted
yes
```

Keď súbory sub1 a sub2 obsahujú klauzuly pre definíciu rovnakých predikátov, v DB ostanú iba tie, ktoré sú v sub2 .

### P4.4.4 Opravy databázy PROLOGu

Pomocou predikátu edit(meno) je možné opravovať klauzuly definujúce predikát s funktorom meno v obrazovkovom editore. Formát zápisu je trochu odlišný ako v zdrojovom texte. Tento spôsob opravovania je výhodný v prípade, keď chceme vyskúšať drobné zmeny v pomerne rozsiahlom programe a v prípade, že pracujeme s tak veľkým programom, že nemôžeme mať v pamäti počítača súčasne aj zdrojový text (v RAM súbore buffer ) aj jeho vnútornú formu v databáze PROLOGu. Je ovšem nutné mať na pamäti, že tieto opravy sú urobené iba v databáze a že zdrojový text ostal nezmenený. Obsah databázy je možné uložiť na MGP pomocou predikátu listing('',subor) a znovu ho nahrať do databázy rovnako, ako by to bol zdrojový text.

### P4.4.5 Funkcie obrazovkového editora

| FUNKCIA | SPECTRUM+ | SPECTRUM |
|---|---|---|
| posun kurzora | | |
| o znak vpravo | šipka vpravo | CS+8 |
| o znak vľavo | šipka vľavo | CS+5 |
| o riadok hore | šipka hore | CS+7 |
| o riadok dole | šipka dole | CS+6 |
| o tabeláciu vpravo (8 pozícií) | SS+e | SS+e |
| na začiatok riadku | EM&šipka vľavo | EM&CS+5 |
| na koniec riadku | EM&šipka vpravo | EM&CS+8 |
| o obrazovku hore | EM&šipka hore | EM&CS+7 |

| | | |
|---|---|---|
| o obrazovku dole | EM&šipka dole | EM&CS+6 |
| na koniec textu | SS+w | SS+w |
| na začiatok textu | SS+q | SS+q |
| výmaz | | |
| riadku s kurzorom | TRUE VIDEO | CS+3 |
| znaku nad kurzorom | GRAPH | CS+9 |
| znaku pred kurzorom | DELETE | CS+0 |
| vloženie riadku pred riadok s kurzorom | INV VIDEO | CS+4 |
| prepínanie režimov vkladanie/prepis | EDIT | CS+1 |
| ukončenie práce editora | SS+i | SS+i |

P4.4.6 Význam niektorých riadiacich znakov

Vyslaním niektorých riadiacich znakov pomocou štandardného predikátu put je možné v PROLOG-80 realizovať užitočné funkcie:

Verzia 3.8

put(1),put(N) ... nastaví kurzor na N-tý riadok zhora, prvý je 0
put(2),put(N) ... nastaví kurzor na N-tý stlpec zľava, prvý je 0
put(4) ... nastaví režim INV VIDEO
put(5) ... nastaví režim TRUE VIDEO
put(7) ... BEEP
put(8) ... DEL (návrat kurzoru o znak a výmaz)
put(9) ... TAB
put(13) ... NL
put(20),put(N) .. nastavenie farby INK a ovládanie režimov FLASH a BRIGHT (PAPER = 7 - biely; zmenu možno dosiahnúť v kombinácii s put(4) )

| N | INK | BRIGHT | FLASH |
|---|---|---|---|
| 32..39 | 0...7 | 0 | 0 |
| 40..47 | 0...7 | 1 | 0 |
| 48..55 | 0...7 | 0 | 1 |

56..63 0...7 1 1

put(24) ... posun kurzora o jeden znak vľavo
put(25) ... posun kurzora o jeden znak vpravo
put(26) ... posun kurzora o jeden riadok dole
put(28) ... posun kurzora do ľavého horného rohu -HOME
put(29) ... posun kurzora o jeden riadok hore
put(31) ... výmaz znakov od kurzora po koniec obrazovky

<u>Verzia 4.1</u> Pozor !!! k dispozícii sú ešte iba pracovné exempláre, v ktorých niektoré činnosti neprebiehajú korektne

put(1),put(N) ... nastaví kurzor na N-tý riadok zhora, prvý je 0
put(2),put(N) ... nastaví kurzor na N-tý stlpec zľava, prvý je 0
put(3) ... nastaví režim INV VIDEO
put(4) ... nastaví reźim TRUE VIDEO
put(5) ... nastaví kurzor na začiatok logického riadku
put(6) ... nastaví kurzor na koniec logického riadku
put(7) ... BEEP
put(8) ... DEL (návrat kurzoru o znak a výmaz)
put(9) ... TAB
put(10) ... LF
put(11),put(N) .. nastaví farbu pozadia
put(12),put(N) .. nastaví farbu písma
put(13) ... NL
put(17),put(N) .. nastaví výstup do okna číslo <u>N</u> (0 ... 7)
put(18),put(X0),put(Y0),put(RX),put(RY)
... nastaví parametre aktálneho okna
X0 - počiatočný stlpec (0 ... 39)
Y0 - počiatočný riadok (0 ... 20)
RX - šírka okna (1 ... 40-X0)
RY - výška okna (1 ... 21-Y0)
put(19) ... výmaz znaku nad kurzorom
put(20),put(N) .. vsunie znak s kódom <u>N</u> na miesto kurzoru
put(21) ... výmaz riadku na ktorom je kurzor
put(22) ... vsunie prázdny riadok na miesto kurzora
put(23) ... posun kurzora o jeden znak hore
put(24) ... posun kurzora o jeden znak vľavo

put(25) ... posun kurzora o jeden znak vpravo ???
put(26) ... posun kurzora o jeden riadok dole ???
put(28) ... posun kurzora do ľavého horného rohu - HOME
put(29) ... CR - posun kurzora na prvý znak riadku
put(30) ... výmaz znakov od kurzora po koniec riadku
put(31) ... výmaz znakov od kurzora po koniec obrazovky

## P4.5 TRASOVANIE VÝPOČTU

V pripade, že systém odpovedá inak, než očakávame, alebo keď chceme poznať detailne postup, ako systém odvodzuje svoju odpoveď, je vhodné použiť trasovanie. V kapitole 5. bola opisaná forma ladiacich výpisov a symbolika označovania brán blokového modelu. Týka sa verzie 4.1, v ktorej je zhodný formát úplného trasovania (zapína sa predikátom trace a vypina notrace ) aj trasovania selektívneho (zapína sa predikátom spy a vypína predikátom nodebug alebo nospy ). Pri štarte systému sú všetky brány ovládané, t.j. systém vyžiada po každom kroku pokyn používateľa k ďalšej činnosti. Sú možné nasledujúce voľby:

c creep - pokračuje sa vo výpočte a trasuje sa do prichodu na ďalšiu ovládanú bránu

s skip - potlačia sa všetky ladiace informácie do prichodu na ďalšiu bránu tohoto cieľa

l leap - potlačí sa trasovanie do prichodu na trasovací bod alebo bránu tohoto cieľa

r retry - aktuálny cieľ sa začne splňať znou od brány Call (výhodné v kombinácii so skip )

i or - na bráne Exit sa zamietne toto riešenie a prejde sa hneď na Redo

f fail - aktuálny cieľ neuspeje (prejde sa na bránu Fail )

<ENTER> - pokračuj bezo zmien ladiacej informácie

g goal - prečíta sa jeden cieľ z klávesnice, vykoná sa a čaká sa na ďalší pokyn

a abort - aktuálny výpočet sa ukončí a riadenie sa odovzdá inerpretu otázok

h help - výpis vyššie uvedených informáci na obrazovku

Zmenu ovládania brán umožňuje predikát 'envé' , funkcia 9-leash.

Pri nastavovaní trasovacích bodov (spy-points) sa vo verzii 4.1 árnosť zadáva pomocou lomítka, napr.

```
?- spy([lubi/2,dievca/1].
```

Vo verzii 3.8 je formát analogického príkazu trocha odlišný:

```
?- spy([lubi(2),dievca(1)].
```

Pozn. Zadávanie árnosti je nepovinné, používa sa iba keď je potrebné rozlíšiť predikáty s rovnakými menami a rozdielnym počtom argumentov.

Vo verzii 3.8 nie je rovnaký formát informácii pri úplnom a čiastočnom ladení. Pri úplnom ladení (predikát trace ) prebieha plynulý výpis na obrazovku, napr.:

```
?- lubi(peter,Koho).<ENTER>
GOAL lubi(peter,_12)
PROVED lubi(peter,mara)
Koho = mara ;<ENTER>                    ; - chceme ďalšie riešenie

RETRY lubi(peter,mara)
GOAL dievca(_12)
PROVED dievca(dana)
GOAL lubi(dana,pivo)
FAILED lubi(dana,pivo)
RETRY dievca(dana)
PROVED dievca(jana)
GOAL lubi(jana,pivo)
```

```
PROVED lubi(jana,pivo)
PROVED lubi(peter,jana)
Koho = jana ;<ENTER>

RETRY lubi(peter,jana)
RETRY lubi(jana,pivo)
FAILED lubi(jana,pivo)
RETRY dievca(jana)
PROVED dievca(mara)
GOAL lubi(mara,pivo)
FAILED lubi(mara,pivo)
RETRY dievca(mara)
FAILED dievca(_12)
PROVED lubi(peter,pivo)
Koho = pivo <ENTER>
yes
```

Selektívne ladenie poskytuje podobný formát informácií ako vo verzii 4.1.:

```
?- spy(dievca).<ENTER>
Spy-point placed on dievca(1)
yes

?-lubi(peter,Koho).<ENTER>
Koho = mara ;<ENTER>

** (1) Call : dievca(_12) ? <ENTER>
** (1) Exit : dievca(dana) ? <ENTER>
** (1) Redo : dievca(dana) ? <ENTER>
** (1) Exit : dievca(jana) ? <ENTER>
Koho = jana ;<ENTER>

** (1) Redo : dievca(jana) ? <ENTER>
** (1) Exit : dievca(mara) ? <ENTER>
** (1) Redo : dievca(mara) ? <ENTER>
** (1) Fail : dievca(_12) ? <ENTER>
Koho = pivo <ENTER>
yes
```

Pre pokračovanie má užívateľ možnost voliť tieto možnosti (odlišné od verzie 4.1):

c continue pokračuj ( zhodné s <ENTER> )

b break * preušenie výpočtu s možnosťou návratu

a abort * úplné prerušenie beźiaceho výpočtu (neznamená výskok zo systému PROLOG a neruší ani databázu)

s skip potlačenie ladiaceho výpisu až po výstup toho bloku, na ktorého vstupe skip zadávame

r retry návrat na Call prislušného bloku (hodi sa, ked sme skipom "preskočili" nás zaujímajúcu časť výpisu)

f fail * umelo vyvolaný neúspech

o off zrušenie selektivneho trasovania ( = nodebug * )

t trace * zapnutie úplného trasovania

n notrace * vypnutie úplného trasovania

h help výpis tejto tabulky

e exit úplné prerušenie výpočtu ( = halt * ) , na rozdiel od základnej verzie PROLOG-80 sa iba vypíše text See you again ... , ale nevyskočí sa z PROLOG-u

p print * všetky výstupy so zohľadnením portray

w write * všetky výstupy so zohľadnením operátorov

d display * všetky výstupy v neoperátorovom tvare

2 backtrace * výpis postupnosti volania cieľov

Funkcie označené * majú aj rovnomenné štandartné predikáty.

Medzi ladiace činnosti patrí aj možnosť prerušenia výpočtu klávesou <BREAK> (najmä keď je podozrenie, že sa výpočet zacyklil). Vo verzii 3.8 po zadaní <SP> (medzera) sa vo výpočte pokračuje, každá iná klávesa vyvolá výpis:

Break !!! Option?

na ktorú užívateľ môže reagovať klávesami a.b.c.e.n.t.?.m . Ich funkcia je zhodná s tou, ktorá bola uvedená pri selektívnom ladení; m (odpovedá mu štandardný predikát core ) vyvolá výpis informácií o voľnej pamäti a o rozmeroch globálneho a lokálneho zásobníka (v nich ukladá systém medziprodukty svojej práce a ich pretečenie je veľmi častou príčinou havárie programov).

Vo verzii 4.1 sa voľba opcií robí klávesou <ENTER>, každá iná klávesa vyvolá pokračovanie vo výpočte.

Od tohoto typu prerušenia výpočtu je potrebné odlíšiť činnosť systému pri volaní predikátu break resp. pri voľbe opcie b : výpočet sa preruší a systém sa prihlási nápoveďou v rámci nového prostredia (stav databázy sa ovšem nemení); v tomto novom prostredí možno robiť akékoľvek ďalšie výpočty (aj meniť stav databázy); pokračovanie prerušeného výpočtu (ovšem bez obnovenia pôvodného stavu databázy) sa vyvolá znakom konca súboru <SS+i> . Takéto prerušenia je možné do seba vnárať. Vo verzii 4.1 je potrebné pri "vynáraní sa" sledovať aktuálnu úroveň, nakoľko zadanie <SS+i> na najvyššej úrovni vyvolá halt (výskok zo systému).

# LITERATÚRA

[1] BRATKO,I.: Prolog Programming for Artificial Intelligence. 1.ed. Wokingham, Addison-Wesley 1986. 423 p.

[2] BRUHA,I.: Programovací jazyk PROLOG. Informační systémy, 13, 1984, s. 205-225.

[3] CLOCKSIN,W.F.-MELLISH,C.S.: Programming in PROLOG. 1.ed. Berlin Heidelberg, Springer Verlag 1981. 279 p. (ruský preklad: Programirovanije na jazyke PROLOG. 1.vyd. Moskva, Mir 1987. 336 s.; český preklad: Programování v PROLOGu. 1.vyd. Slušovice, JZD Závody aplikované kybernetiky 1986. 352 s.)

[4] COELHO.H.-COTTA,J.C.-PEREIRA,L.M.: How to Solve it with PROLOG. 2-nd ed. Lisboa, Laboratoria National de Engenharia Civil 1980. 215 p.

[5] FU,K.S.: Strukturnyje metody v raspoznavanii obrazov. 1.izd. Moskva, Mir 1977. 319 s.

[6] G.ANESSINI,F. et al.: PROLOG. 1.ed. Wokingham, Addison-Wesley 1986. 260 p.

[7] HAVEL,I.M.: Robotika (Úvod to teorie kognitivních robot ). 1.vyd. Praha, SNTL 1980. 280 s.

[8] JIRKU,P.-ŠTÉPÁNEK,P.-ŠTÉPÁNKOVÁ,O.: Programování v jazyku PROLOG. 1.vyd. Praha, SNTL 1988. (v tlači)

[9] NILSSON,N.J.: Problem-Solving Methods in Artificial Intelligence. 1. ed. New York, McGraw Hill 1971. (ruský preklad: Iskusstvennyj intellekt. 1.izd. Moskva, Mir 1973. 270 s.)

[10] STERLING,L.-SHAPIRO,E.: The Art of PROLOG. 1.ed. Cambridge, MIT Press 1986.

[11] WATERMAN,D.A.: A Guide to Expert Systems. 1.ed. Reading, Addison-Wesley 1986. 420 p.