Obsah

Vestavěné predikáty - seskupování
- použití predikátů findall, bagof a setof

Úvod do logického programování

Prolog a logické programování

Prolog je jazyk pro programování symbolických výpočtů. Název je odvozen se slov PROgramování v LOGice, což naznačuje, že jazyk vychází z matematické logiky. Úspěch jazyka Prolog byl podnětem pro vznik nové disciplíny matematické informatiky - logického programování.

Pohled do historie

První verzi Prologu vyvíjel A.Colmerauer se svou skupinou na univerzitě v Marseille od roku 1972.

V roce 1974 analyzoval R.Kowalski předpoklady a způsob práce Prologu a vytvořil jeho teoretický model. Na tento model pak navázaly různé implementace: mezi nejúspěšnější patří verze D.H.D.Warrena z roku 1977, popsaná ve výborné učebnici W.F.Clocksina a C.S.Mellishe "Programming in Prolog".

Zájem o celé logické programování vzrostl po roce 1981, kdy byl v Japonsku Prolog zvolen za základní jazyk centrální jednotky počítače páté generace. Tento ambiciózní projekt sice nakonec neuspěl, avšak Prolog se díky němu prosadil jako jeden z nejpoužívanějších programovacích prostředků umělé inteligence.

Princip Prologu

Při programovaní v Prologu je třeba se soustředit zejména na dvě otázky:

- Jaké formální skutečnosti a vztahy se v řešeném problému vyskytují?
- Které vztahy jsou vzhledem k řešení problému pravdivé?

Při psaní programů v Prologu tedy vycházíme ze

- specifikace známých faktů o řešeném problému a
- určení známých, či splněných vztahů (relací) mezi těmito fakty.

Samotné řešení problému není nutné chápat jako posloupnost po sobě následujících instrukcí. Programátor se nemusí zabývat podrobným plánováním činnosti programu, nezapisuje jak se má úloha vyřešit. Pouze uvádí (deklaruje) vše, co je možné vypočítat. Vlastní průběh výpočtu se řídí výhradně interpretem Prologu.

Matematicky řečeno, prologovský program se skládá z množiny vhodně zapsaných Hornových klauzulí. Každá klauzule reprezentuje buďto nějaký fakt, nebo pravidlo, které určuje vztah řešení problému k daným faktům, resp. způsob odvození řešení z těchto faktů.

Prolog je jazyk...

- neprocedurální - postup řešení problému není to, co by nás nejvíc zajímalo
- deklarativní - při psaní programu deklarujeme vše, co je možné vypočítat
- konverzační - uživatel klade dotazy, na které mu Prolog odpovídá

- interaktivní - pokud uživatel dotazy neklade, Prolog nepracuje a čeká
- velice zábavný - jak jistě zjistíte v dalších přednáškách a cvičeních

Syntaxe

Predikáty

Velkou výhodou jazyka Prolog je jednoduchá syntaxe. Základní jednotka programu se nazývá predikát.

Predikátem může být

- fakt, neboli nepodmíněný příkaz
- pravidlo, neboli podmíněný příkaz.

Fakta a pravidla jsou uložena ve společné databázi, kterou Prolog prohledává, když se snaží odpovědět na dotaz uživatele.

Predikát označujeme názvem, za který do závorky uvedeme argumenty, oddělené čárkou. Počtu argumentů se říká arita predikátu. Zápis predikátu vždy ukončujeme tečkou.

likes(jim, mary).
likes('Jim','Mary').

Slovo *likes* značí predikát, *jim* a *mary* jsou argumenty predikátu, predikát má aritu 2. Oba uvedené predikáty vyjadřují totéž, liší se pouze zápisem: pokud chceme, aby argument začínal velkým písmenem, musíme jej zapsat do uvozovek (bez nich by Prolog neodlišil hodnotu argumentu od názvu proměnné.)

Název predikátu by měl co nejlépe vystihovat skutečnost, kterou chceme popsat. Proto při popisu kladného mezilidského vztahu dáme přednost slovu "likes" před méně výstižným, ale také použitelným "wxyz".

Interpretace predikátu závisí na vůli programátora. Nemáme-li bližší informace, význam předchozího predikátu si můžeme vyložit buď tak, že Jim má rád Mary, nebo že Mary má ráda Jima.

? Určete název, argumenty a aritu predikátů a vysvětlete, co popisují:
play(jim,jane, golf).
hot(water).

První predikát *play* má argumenty *jim, jane a golf* a aritu 3. Vystihuje skutečnost, že Jim a Jane hrají golf. Druhý predikát má aritu 1 a zmiňujete se o vysoké teplotě vody.

Můžeme definovat predikát village(prague).?

Ano, Prolog samozřejmě nepozná, že predikát je ve skutečnosti nepravdivý.

S prologovskou databází pracujeme v dotazovacím režimu tak, že do dialogového řádku za znaky ?- píšeme dotaz, ukončený tečkou.

Po zadání dotazu se Prolog snaží vyhledat v databázi vyhovující fakta. Pokud je najde, odpovídá pozitivním hlášením "yes", v opačném případě sděluje "no". Proces porovnávání dotazů s fakty s cílem vyhledat shodu se označuje anglickým

termínem *matching*, jemuž se pravděpodobně nejvíce podobá český výraz unifikace :-)

Dvě fakta lze unifikovat, jestliže predikáty jsou stejně pojmenované, mají stejnou aritu a korespondující argumenty jsou rovněž shodné.

Velmi jednoduchý dotaz může vypadat takto:

?-hot(water).

Zde se ptáme na skutečnost, zda voda je horká, resp. jestli Prolog zná fakt, že voda je horká. Pokud bude databáze obsahovat příslušný řádek, dostaneme výstižnou odpověď: *yes*

Vždy, když dostaneme odpověď "yes", víme, že Prolog nalezl v databázi vyhovující fakt. Při odpovědi "no" je situace složitější - buď databáze neobsahuje daný fakt, anebo jej obsahuje, ale náš program dobře nefunguje.

Proměnné

V dotazech můžeme používat proměnné, a to velmi pohodlně, protože Prolog nevyžaduje jejich inicializaci. Proměnná vždy začíná velkým písmenem.

V okamžiku, kdy zadáváme dotaz, je proměnná *nespecifikovaná*, tedy neobsahuje hodnotu. Ve chvíli, kdy se hodnotu podaří přiřadit, stává se proměnná *specifikovanou*. Po vyhodnocení dotazu Prolog vypíše specifikované proměnné.

?-likes(mary, X).

Ptáme se, zda Mary má někoho ráda a současně chceme zjistit jeho jméno. V dotazu uvedeme proměnnou X, do níž se Prolog bude snažit dosadit. Najde-li fakt *likes(mary, john).*, dostaneme odpověď

X=john

? **Myslíte, že můžeme položit tento dotaz?**
likes(mary,Clovek-ktereho-ma-rada).

Ano, důležité je, aby označení proměnných začínalo velkým písmenem.

Jakým dotazem bychom zjistili všechny dvojice, které se mají rády?

Aritmetické operátory

Matematické operace

Prolog umožňuje provádět jednoduché výpočty. K provádění základních aritmetických operací slouží následující operátory:

- +...sčítání
- - ...odčítání
- * ...násobení
- / ...dělení
- // ... celočíselné dělení
- mod ...zbytek po dělení

2+3 ... součet dvou čísel
7 * 8 ... součin dvou čísel
20 mod 3 ... výpočet zbytku z čísla 20 po dělení třemi
A - B ... odečtení obsahu dvou proměnných, A i B musí být specifikované

Kromě uvedených operací bývají v jednotlivých implementacích Prologu dostupné různé další matematické funkce, například pro výpočet absolutní hodnoty nebo pro generování náhodného čísla - v LPA-Prologu to jsou funkce **abs** a **rand**. Úplný výčet funkcí je součástí dokumentace.

Vyčíslení

Pro vyčíslení výrazu se nepoužívá rovnítko, jak bychom asi čekali, ale operátor **is**.

A is 2+3 ... součet dvou čísel, A = 5
B is 7 * 8 ... součin dvou čísel, B = 56
C is 20 mod 3 ... výpočet zbytku z čísla 20 po dělení třemi, C = 2
D is A - B ... odečtení obsahu dvou proměnných, A i B musí být specifikované

Rovnítko

Symbol "=" (rovnítko) v Prologu také najdeme, ale nepoužívá se k zápisu výpočtu, nýbrž k porovnání proměnných. Nastat mohou různé situace:

- porovnáváme dvě specifikované proměnné - například jestliže A = 3 a B = 3, pak platí, že A = B.
- porovnáváme specifikovanou proměnnou a nespecifikovanou proměnnou - když A = 3, bude výsledkem porovnání B = A zjištění, že B = 3.
- porovnáváme dvě nespecifikované proměnné - pokud A i B jsou nespecifikované, zápis A = B zajistí, že se proměnné stanou sdílenými. Až jedna z nich nabude hodnotu, stejnou hodnotu získá i druhá proměnná.

Porovnávat lze i celé struktury. Platí, že dvě struktury se rovnají, když mají shodný název i argumenty. Toto je ukázka porovnání dvou seznamů:

seznam(a,b,c) = seznam(X,b,Y) ... výsledkem porovnání je zjištění, že struktury se rovnají a že X = a, Y = c

? **Jak dopadne porovnání: cisla(1, B, C, 6) = cisla (M, N, [7,8,9], 6) ?**

struktury se rovnají a platí M = 1, B = N, C = [7,8,9].

Kromě početních operací lze také porovnávat hodnoty pomocí relačních operátorů, jejichž součástí je i vyhodnocení výrazů:

- >, <, >=, =< ... větší, menší, větší nebo rovno, menší nebo rovno včetně vyhodnocení výrazů na obou stranách
- =:= ... vyhodnocením na obou stranách s ověřením rovnosti
- =\= ... vyhodnocení na obou stranách s ověření nerovnosti

K <L ... testování relace "menší než" pro proměnné K, L
3+5 = : = 8 ... výsledek sčítání 3 + 5 se porovnává s číslem 8
1 + 2 = : = 2 + 1... výsledky dvou sčítání se porovnávají
A - B =\= 5 ... vypočtený rozdíl má být různý od 5

K porovnání bez přiřazení slouží operátory = = a \= = .

R = = S ... porovnání, v němž si nepřejeme nasdílet proměnné R a S

? **Jak zareaguje Prolog na zápis 1 + 2 = 3?**

Hlášením "no". Uvedený řádek totiž neobsahuje požadavek na vyčíslení výrazu na levé straně. Bez vyčíslení ale musíme s oběma stranami jako s řetězci.

! **Vypočítejte do proměnné A součet tří čísel a výsledek přiřaďte i do proměnné B.**

Vestavěné predikáty - vstup a výstup

Stejně jako další programovací jazyky umožňuje i Prolog zpracovávat vstupy a výstupy prostřednictvím několika standardních predikátů.

Načítání a výpisy

- **read(A)** - načtení vstupu z klávesnice do proměnné A (zadání musí končit tečkou). Chceme-li načíst řetězec začínající velkým písmenem, uvedeme jej v apostrofech.
- **write(B), write('ahoj')** - výpis proměnné B nebo řetězce na obrazovku (řetězec píšeme do apostorfů)
- **get(C)** - načtení tisknutelného znaku a uložení jeho ascii-hodnoty do proměnné C, vstup se neukončuje tečkou
- **get0(D)** - načtení tisknutelného nebo i netisknutelného znaku a uložení jeho ascii-hodnoty do proměnné D, vstup se neukončuje tečkou
- **nl** - odřádkování výpisu
- **tab(N)** - výpis N mezer

Jednoduchou a jistě srozumitelnou ukázkou použití uvedených predikátů je například tento program:

```
ahoj:-

        write('Jak vam rikaji?'),nl,
        read(Jmeno),
        tab(5),write('Ahoj '),write(Jmeno),nl,
        write('k ukonceni programu stisknete libovolnou klavesu'),nl,
        get0(B),
        write('stisnute klavese odpovida ascii hodnota '),write(B),nl.
```

Po spuštění může program probíhat takto:

```
| ?- ahoj.
Jak vam rikaji?
|: 'Tome'.
Ahoj Tome
k ukonceni programu stisknete libovolnou klavesu
|: g
stisnute klavese odpovida ascii hodnota 103
yes
```

Konverze řetězců a seznamů

Někdy se může hodit také predikát **name**, kterým lze převést posloupnost tisknutelných znaků na seznam ascii-hodnot těchto znaků:

```
slovo-na-seznam:-read(Slovo),name(Slovo,X),write(X),nl.
```

Do proměnné Slovo uložíme vstup, který převedeme na seznam ascii hodnot a ten vypíšeme. Toto je možné volání programu:

```
| ?- slovo-na-seznam.
|: praha.
[112,114,97,104,97]
yes
```

Jakým programem byste převedli seznam ascii-znaků na řetězec?

?

Například: seznam-na-slovo:-X=[112,114,97,104,97],name(Slovo,X),write(Slovo),nl.

!

Napište program, kterým přečtete dvě čísla, sečtete je a vypíšete výsledek.

Seznam - definice a použití

Seznam je datová struktura, velmi často používaná v symbolických výpočtech. Je to pojmenovaná posloupnost libovolného počtu položek. Položkou může být jak jediná hodnota, tak i libovolná struktura. V Prologu zapisujeme seznam do hranatých závorek a položky oddělujeme čárkami. Seznam může být i prázdný a může také obsahovat další seznamy.

[7, 5, 9]...seznam tří položek
[]...prázdný seznam
[a,[b,c],d,e,f]...seznam pěti prvků, jehož druhým prvkem je opět seznam

Pro potřeby zpracování odlišujeme první prvek seznamu, kterému říkáme hlava (H), a zbytek seznamu, neboli tělo (T).

[a,b,c]...H = a,T = [b,c]
[[a,b,c],d,e]...H = [a,b,c],T = [d,e]
[a]...H = a,T = []
[]... hlava i tělo chybí

Chceme-li provést rozdělení na hlavu a tělo v pravidle, používáme svislou čáru: S = [A|B]... seznam S má hlavu A a tělo B. Kdybychom potřebovali zpracovat naráz první i druhý prvek seznamu, psali bychom S = [A,B|C]. Praktické použití proměnných je patrné z ukázky.Výpis prvního prvku seznamu:

```
prvni1(P,Sez):-Sez=[H|T],P=H.
```

Pravidlo voláme například:

```
prvni1(X,[1,2,3,4]).
```

Dostaneme odpověď X = 1.

? **Napadá vás způsob, jak uvedený predikát zestručnit?**

Dvě možnosti jsou:

- ***prvni2(P,Sez):-Sez=[P|_].***
- ***prvni3(P,[P|_])***.

Potřebujeme-li se odkázat na seznam uložený v databázi, můžeme to udělat takto:

```
seznam([1,2,3,4,5]).
prvni(P,Sez):-seznam(Sez),Sez=[P|_].
```

! **Napište predikát pro výpis druhého prvku seznamu!**

Seznam - jednoduchá pravidla

Pravidla pro práci se seznamem jsou dost důležité - využijeme je později ve složitějších programech, ve kterých bude seznam často používán coby užitečná datová struktura.

Předchozí kapitolu jsme zakončili pravidlem pro výpis prvního prvku seznamu. Nyní si vysvětlíme několik dalších pravidel.

Výpis posledního prvku ze seznamu

```
posledni(P,[P|[]]).
posledni(P,[_|T]):-posledni(P,T).
```

- Pravidlo má dva argumenty: do proměnné P má být přiřazen poslední prvek seznamu, druhý argument obsahuje zkoumaný seznam.
- První řádek pravidla reaguje tehdy, když v seznamu zbývá jediný, tedy poslední prvek.
- Pro ostatní případy - když seznam obsahuje více než jen jeden prvek - slouží druhý řádek, který obsahuje jediný příkaz: rekurzivní volání, do něhož jako argumenty vstupují opět proměnná P a seznam, zkrácený o hlavičku.

? **Zkuste napsat predikát pro výpis předposledního prvku seznamu!**

```
predposledni(P,[P,K]).
predposledni(P,[_|T]):-predposledni(P,T).
```

Výpis seznamu po prvcích od začátku do konce

```
vypis([]):-write('konec'),nl.
vypis([H|T]):-write(H),nl,vypis(T).
```

Pravidlo pracuje s jediným argumentem, jímž je zpracovávaný seznam.

- Pokud je seznam prázdný, končí výpis upozorněním.
- Jestliže seznam lze rozdělit na hlavičku H a tělo T, vypíšeme obsah proměnné H, odřádkujeme a pokračujeme výpisem T.

? **Jak bychom museli pravidlo pro výpis po prvcích upravit, kdybychom nechtěli seznam vkládat jako argument, ale převzít jej z databáze? Předpokládejme, že v databázi je třeba *seznam([1,2,3,4,5]).***

Samotný výpis bychom zachovali, dopsali bychom pouze nový predikát *vypisuj* pro zavolání výpisu. Program by vypadal takto:

```
vypis([]):-write('konec'),nl.
vypis([H|T]):-write(H),nl,vypis(T).

vypisuj(S):-seznam(S),vypis(S).
```

Člen seznamu

```
clen(A,[A|T]).
clen(A,[H|T]):-H\=A,clen(A,T).
```

Pravidlo ověřuje, zda se argument A vyskytuje v seznamu, který je druhým argumentem. Pravidlo má opět dvě části pro dvě možné situace:

- buď je prvek A hlavou seznamu
- anebo se A vyskytuje někde v těle seznamu - pak je třeba znovu zavolat totéž pravidlo, již jen s tělem seznamu.

? **Pravidlo *clen* vrací odpověď yes nebo no podle toho, jestli A je nebo není prvkem seznamu. Jak bychom měli pravidlo upravit, aby odpovídalo celou větou?**

Například takto:
clen(A,[A|T]):-write(A),write(' se vyskytuje v seznamu.'),nl.
clen(A,[H|T]):-H\=A,clen(A,T).

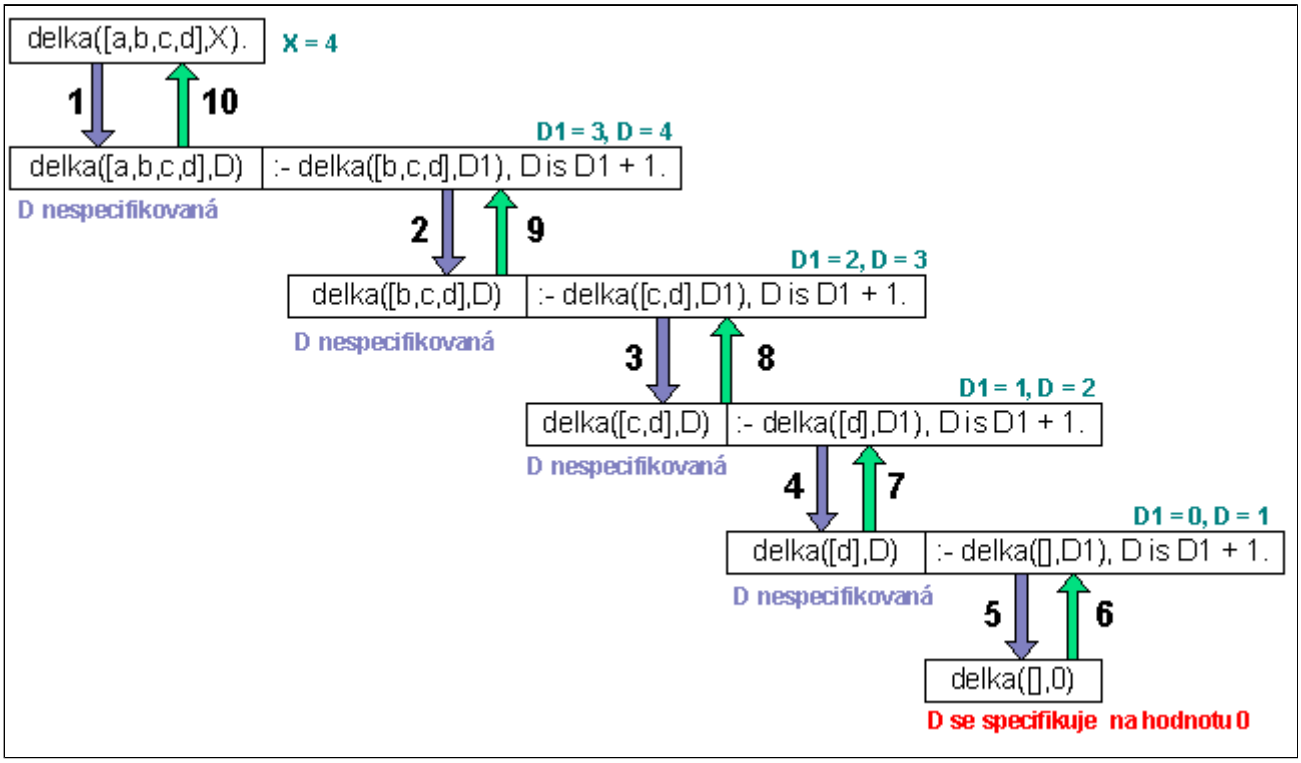
Délka seznamu

delka([],0).
delka([H|T],D):-delka(T,D1), D is D1 + 1.

Zde jako první argument vložíme seznam, druhým argumentem je proměnná, do níž má být uložena vypočtená délka. Pravidlo funguje takto:

- prázdný seznam má délku 0
- délku neprázdného seznamu určíme tak, že zjistíme délku seznamu zkráceného o hlavičku a přičteme jedničku.

Činnost predikátu je znázorněna i na obrázku:



? **Stalo by se něco, kdybychom v predikátu uvedli sčítání a rekurentní volání v opačném pořadí - takto:**
delka([],0).
delka([H|T],D):- D is D1 + 1,delka(T,D1).?

Ano, stalo. Pravidlo by nefungovalo - marně by se snažilo sečíst do proměnné D jedničku a obsah doposud nespecifikované proměnné D1.

Vytknutí prvku ze seznamu

```
vytkni(X,[X|T],T).
vytkni(X,[H|T],[H|T1]):-vytkni(X,T,T1).
```

"Vytknutím" rozumíme oddělení prvku X ze seznamu (druhý argument) tak, že vznikne nový seznam (třetí argument),neobsahující prvek X.

Postup vytýkání se opět dělí na dva kroky:

- buď je prvek X hlavou seznamu - pak zbytkem po vytknutí bude tělo T původního seznamu
- nebo je prvek X obsažen někde v těle T - pak zaznamenáme hlavičku původního seznamu (H) do plánovaného výsledku vytýkání a pokračujeme rekurentním voláním, v němž chceme vytknout X již jen z těla původního seznamu (T) a vytvořit tak seznam T1.

Smazání prvního výskytu prvku ze seznamu

```
smazej(X,[],[]).
smazej(X,[X|T],T).
smazej(X,[H|T],[H|V]):-H\=X,smazej(X,T,V).
```

Smazat chceme prvek X z druhého argumentu a vytvořit tak třetí argument pravidla. Pravidlo má tři části:

- buď je zkoumaný seznam prázdný, takže neobsahuje prvek X a výsledkem mazání bude opět prázdný seznam
- nebo druhý argument obsahuje prvek X jako hlavičku - pak výsledkem mazání je tělo zkoumaného seznamu>
- nebo hlavička H zkoumaného seznamu je různá od prvku X - tedy hlavičku obdobně jako v predikátu vytkni přesuneme do výsledku mazání a voláme stejný predikát, do něhož předáme prvek X, tělo T a opět proměnnou V, která má být naplněna.

? Co se stane, budeme-li chtít smazat prvek, který se v seznamu nevyskytuje, například *smazej(1,[2,3,4,5],A).?*

Pravidlo vrátí A = [2,3,4,5]. Do posledního rekurentního volání vstoupí smazej(1,[],A), na což zareaguje první řádek, v němž dojde ke specifikaci proměnné A = []. Při navracení se k tomuto prázdnému seznamu postupně připojí prvky 5,4,3,2.

! Napište predikát pro výpis seznamu po prvcích odzadu, tedy od posledního prvku k prvnímu.

Proměnná ve výpočtu

Hodnoty proměnných během výpočtu

Při rekurzivních voláních pravidla se neustále vytvářejí nové kopie proměnných, do nichž jsou ukládány nové hodnoty. Připomeňme si jednoduché pravidlo pro výpis prvků seznamu:

```
vypis([]).
vypis([H|T]):-write(H),nl, vypis(T).
```

Zadáme-li například požadavek *vypis([a,b,c,d,e])*, při prvním volání Prolog rozdělí argument na H = a a T = [b,c,d,e]. Do druhého kroku vstupuje argument T = [b,c,d,e], který je ihned rozdělen do dvou proměnných H = b a T = [c,d,e] a tak dále. Proměnná T tedy v každém kroku rekurze obsahuje něco jiného.

Ale pozor: v pravidlech se mohou vyskytovat i proměnné, jejichž hodnota se v průběhu rekurzivních volání nemění. Typickým příkladem je proměnná A vystupující v pravidle *clen*:

```
clen(A,[A|_]).
clen(A,[H|T]):-clen(A,T).
```

Když zavoláme *clen(5, [2,3,5,7,9])*, znamená to, že A = 5 jak v prvním, tak ve všech dalších krocích. Proměnná A je tedy po celou dobu výpočtu specifikovaná a obsahuje totéž.

Naopak v pravidle

```
posledni(P,[P|[]]).
posledni(P,[_|T]):-posledni(P,T).
```

je při zavolání *posledni(S,[9,8,7,6])* proměnná S nespecifikovaná tak dlouho, dokud Prolog nedojde k poslednímu prvku seznamu. I proměnná S však beze změny prochází výpočtem.

Zvláštním typem proměnné je proměnná anonymní. Označuje se podtržítkem a používáme ji tehdy, když konkrétné hodnotu nepotřebujeme znát.

Akumulátor

Zvláštním případem proměnné je takzvaný akumulátor: jeho hodnota se mění v každém kroku rekurze a při posledním volání se hodnota z akumulátoru přesune do proměnné, která má obsahovat výsledek operace. Uveďme dvě ukázky - generování seznamu a otáčení seznamu.

Vytvoření vzestupně uspořádaného seznamu N čísel

```
zdola(N,S):-zdola(N,[],S).
zdola(0,S,S).
zdola(N,Ak,S):- N > 0, N1 is N-1, zdola(N1,[N|Ak],S).
```

Opět voláme predikát pomocí dvou argumentů - čísla N a proměnné pro výsledný seznam. Tento predikát si vyžádá jiný predikát o třech argumentech: přibývá prostřední proměnná, čili akumulátor. Do akumulátoru se připraví prázdný seznam. Následně se Prolog snaží uskutečnit jednu ze dvou možností:

- Je-li třeba vygenerovat 0 prvků, přesune stane se obsah akumulátoru S výsledkem výpočtu.
- Je-li třeba generovat ještě N prvků, připojí se na začátek akumulátoru nový prvek, tedy číslo N, a spustí se nové volání predikátu zdola s nižší hodnotou prvního argumentu.

Otočení seznamu s využitím akumulátoru

Pravidlo se velice podobá předchozímu generování.

```
reverse(S,S1) :- rev(S,[],S1).
rev([],S1,S1).
rev([H|T],Ak,S1):-rev(T,[H|Ak],S1).
```

Voláme predikát *reverse* a předáváme mu jako první argument seznam a jako druhý argument proměnnou, do níž má být seznam otočen. Predikát *reverse* vyvolá další predikát *rev*, v kterém vystupuje jako akumulátor nový (prostřední) argument, jímž je zpočátku prázdný seznam.

Predikát *rev* má dvě části:

- První část se spustí v případě, že otáčený seznam je prázdný. Tehdy se obsah akumulátoru S1 stane výsledkem otáčení tak, že přejde i do posledního argumentu.
- Druhá část predikátu se spustí, když otáčený seznam není prázdný. V tom případě se hlavička H připojí na počátek seznamu uloženého v akumulátoru a pokračuje se novým voláním predikátu *rev*.

Dobrá zpráva na závěr: v LPA je predikát pro otáčení seznamu již vestavěn, nazývá se *reverse*.

Efektivnost programu

V Prologu často programujeme úlohy, které lze řešit i bez počítače a bez hlubší logick
hrubou silou, tedy prohledáváním mnoha kombinací hodnot proměnných. Pokud ale t
program, měli bychom se snažit, aby jeho postup nebyl hrubý, nýbrž aby směřoval k
nejpřímočařeji a bez zbytečných oklik. Ukažme si tedy, jakým způsobem docílit, aby
efektivní, nebo spíše, jak to udělat, aby nebyl neefektivní.

Řazení podmínek

První ukázkou bude problém magického čtverce: do mřížky 3x3 pole máme dosadit čí
(každé použijeme jen jednou) tak, aby součty čísel v řádcích, sloupcích a úhlopříčkác
shodné. Zde je možné začít psát řešení takto:

```
ctverec(A,B,C,D,E,F,G,H,I):-
   cislo(A), cislo(B), cislo (C), cislo(D),cislo(E), cislo(F), cislo (G), cislo(H), c
  A\=B, A\=C, ...,
  A + B + C =:= D + E +F, ...
```

Takový program by ale pracoval velmi dlouho. Prolog by totiž nejprve zkusil dosadit c
proměnných číselné hodnoty, a to zpočátku do všech stejné, například A = B = C = D
= H = I = 1. Po dosazení do všech devíti proměnných by narazil na testování, musel
dosazení změnit. Navracení s novým dosazováním by se opakovalo tak dlouho, dokuc
splněna všechna "různá od". Potom teprve potom by se Prolog dostal ke sčítání s
vyhodnocováním. Při každém neúspěchu by se znovu vracel a měnil hodnoty proměn
nejméně příznivém případě by bylo třeba probrat úplně všechny kombinace hodnot p
až I, než by byla nalezena jedna kombinace, splňující všechna omezení.

Lepší řešení by mohlo začínat takto:

```
ctverec(A,B,C,D,E,F,G,H,I):-
   cislo(A),
   cislo(B),A\=B,
   C is 15 - A - B, cislo(C),B\=C,A\=C,
   cislo(D),A\=D,B\=D,C\=D,
   G is 15 - A - D,cislo(G),A\=G,B\=G,C\=G,D\=G, ...
```

Změna je v tom, že veškerá porovnání provedeme ihned, jak je to možné.Například h
naplnění proměnných A a B požadujeme, aby jejich hodnoty byly navzájem různé. Dá
využíváme poznatek, že součet čísel v jednotlivých řadách je vždy 15, díky čemuž mů
hodnotu proměnné C ihned spočítat.

Poučení z uvedené ukázky: vždy se snažíme upořádat podmínky tak, abychom se vyh
nevhodným kombinacím.

? **Jaké další zlepšení programu je možné?**

**Místo čísel vyvolávaných z databáze bychom mohli použít seznam, z nějž byc
postupně vytýkali prvky A až I. Každé nové vytknutí by proběhlo ze zbytku p
předchozím vytknutí. Tím bychom zajistili, že hodnoty proměnných A až I bu
aniž použijeme porovnávání A \= B, A \= C atd.**

Počet potřebných volání

Kromě vhodného seřazení podmínek musíme dbát také na to, aby podcílů, které mus
v rámci jednotlivých kroků rekurze, nebylo neúnosně mnoho. Hezkým odstrašujícím p
výpočet n-tého členu Fibonacciho posloupnost.

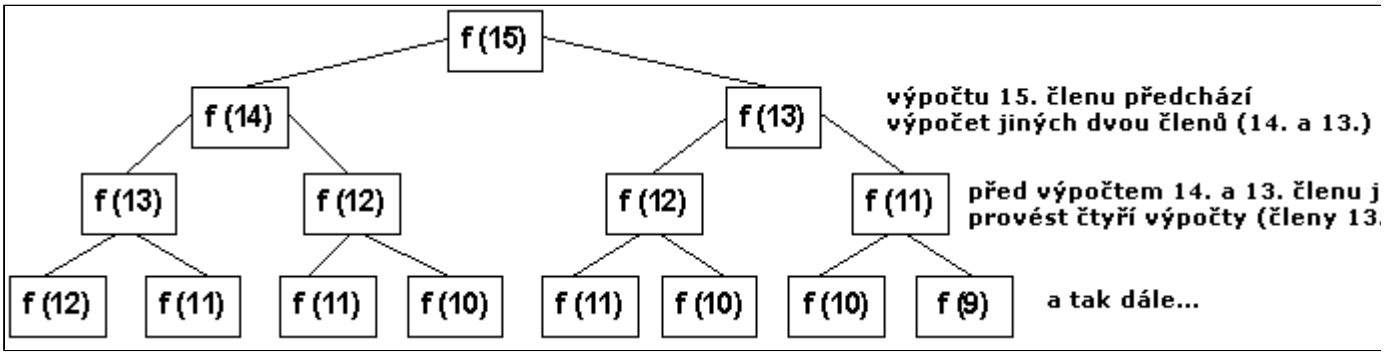
Posloupnost je definována takto:

f(0) = 1
f(1) = 1
f(N) = f(N-1) + f(N - 2),

tedy členy posloupnosti jsou čísla 1,1,2,3,5,8,13,21, ...

Výpočet n-tého členu můžeme provádět buď shora, nebo zdola. Řekněme, že počítám
člen.

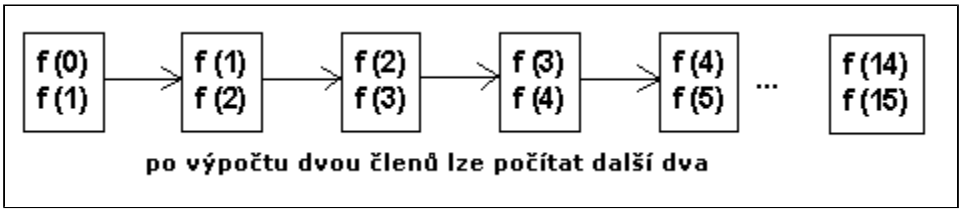
Postup shora znamená, že před nalezením 15. členu musíme spočítat členy s indexy
ale určíme f(14), musíme spočítat f(13) a f(12), před výpočtem f(12) musíme znát f(
tak dále - podívejte se na obrázek. Je na něm zakreslena stromová struktura postupu
Výpočet členu v každém uzlu stromu předpokládá, že dříve byly spočteny všechny čle
úrovních stromu.



Program pro tento způsob výpočtu je jednoduchý, ale trvá velmi dlouho a hodí se nar
výpočtu zmíněného patnáctého členu, poté dochází k přetečení paměti:

```
f(0,1).
f(1,1).
f(N,X):-N>1,N1 is N-1, N2 is N-2, f(N1,X1), f(N2,X2), X is X1 + X2
```

Postup zdola je mnohem efektivnější. Prostě počítáme členy od prvního nahoru a u n-
skončíme. V každém kroku stačí si pamatovat pouze dvě předchozí hodnoty.



Program vypadá takto:

```
fib(N,X):-N>=0,fib(0,1,1,N,X).
fib(N,X,_,N,X):-!.
fib(I,J,K,N,X):-I1 is I+1,K1 is K+J,fib(I1,K,K1,N,X).
```

Poučení z ukázky: použití pomocných proměnných a akumulátorů může zmenšit poče rekurzivních kroků, čímž se běh programu zkrátí.

Repeat - příkaz cyklu

Predikát repeat se používá k vytvoření cyklu. Je vždy splněn - úplně vždy, tedy i při navracení. Proto Prolog, pokud přes repeat jednou přejde, se nemůže vrátit zpět před něj.

```
vypis1:-read(ahoj),write('nazdar'),nl.
vypis2:-repeat,read(ahoj),write('nazdar'),nl.
```

První pravidlo čeká, že uživatel zadá z klávesnice slovo 'ahoj'. Pokud se tak stane, program také pozdraví a skončí. Jestliže uživatel napoprvé nezadá 'ahoj', program ihned končí neúspěchem.

Druhé pravidlo již pracuje v cyklu. Po automaticky splněném příkazu repeat čeká Prolog na zadání slova 'ahoj'. Pokud se nedočká, žádá další vstup z klávesnice, a to tak dlouho, dokud není zadáno 'ahoj'. Pak program pozdraví a skončí.

Z předchozího příkladu je patrné, že příkaz repeat využijeme při načítání z klávesnice. Můžeme výzvu k zadání vstupu opakovat tak dlouho, dokud vstup neodpovídá požadavkům.

? **Jak bude fungovat následující pravidlo?**
pis:-repeat, read(X), X<5,X>1.

Pravidlo bude opakovat výzvu k zadání vstupu tak dlouho, dokud vložená hodnota nebude z rozmezí 1 - 5.

Příkaz repeat lze kombinovat s příkazem fail.

? **Jak bude fungovat toto pravidlo?**
vypis:-repeat,write('ahoj'), nl, fail.

Program se zacyklí - bude se stále vracet od nesplněného fail k vždy splněnému repeat a zpět. Uvedené pravidlo je ukázkou nekonečného cyklu.

Kombinací repeat a fail můžeme vytvořit program, který bude pracovat cyklu tak dlouho, dokud nevydáme požadavek na ukončení.

```
druhamocnina:-

        repeat,
        write('Chcete vypocitat druhou mocninu nejakeho cisla
        (ano/ne)? '),
        read(X),

                (X='ne';
                write('Zadejte cislo:'),
                read(C),
                V is C*C,
                write('Vysledek je '),write(V), nl, fail).
```

Uvedený program neustále nabízí výpočet druhé mocniny. Podle odpovědi na výzvu buď končí (proběhne část před středníkem), nebo počítá (část za středníkem). Po výpisu výsledku výpočtu pravidlo selže a vrací se až k repeat, tedy

k nové výzvě.

! **Napište pravidlo, které bude načítat hodnoty z klávesnice tak dlouho, dokud uživatel nezadá dvakrát po sobě stejnou hodnotu.**

Příkaz řezu - !

Řez, v Prologu označovaný vykřičníkem, slouží k oddělení dvou částí výpočtu. Před řez není možné se vrátit, takže jej lze využít k zefektivnění výpočtu - nenutíme Prolog, aby se navracel o mnoho podcílů zpět, pokud víme, že tak nenajde nová řešení.

pocitej:-

```
repeat,
write('Zadej cislo mensi nez 100'),
read(S),
S<100,!,
faktorial(S).
```

Zde je řez použit k oddělení načítání čísla, které má být menší než 100,od výpočtu faktoriálu. (Samotný predikát faktoriál v ukázce neuvádíme). Pokud se z nějakého důvodu nepodaří faktoriál vypočítat, Prolog se nebude vracet k novému načítání čísla a rovnou skončí.

Jak vidíte, řezem je možné ovlivnit průběh výpočtu tak, že zamezíme zbytečným návratům. Kromě toho lze ale řezem zabránit i nežádoucímu pohybu dopředu, přesněji řečeno zbytečnému vykonávání alternativních větví pravidla.

```
zjisti1(A):-0 is A mod 2, write('cislo je sude'),nl,!.
zjisti1(A):-1 is A mod 2, write('cislo je liche'),nl.
```

Pravidlo má zjistit, zda argument A je sudé, nebo liché číslo. Pokud se vykoná první větev pravidla - tedy číslo bude sudé - Prolog už nebude zkoumat, jestli je číslo liché.

Předchozí variantu je možné zkrátit takto:

```
zjisti2(A):-0 is A mod 2, write('cislo je sude'),nl,!.
zjisti2(A):-write('cislo je liche')
```

Všimněte si, že kdybychom z pravidla *zjisti1* odstranili řez, na jeho fungování by se nic nezměnilo. Avšak kdybychom řez vynechali v pravidle *zjisti2*, program by nefungoval.

? **Jak by se projevila chyba, kdybychom odstranili řez z predikátu zjisti2 ?**

Pro sudá čísla by pravidlo vracelo dvě řešení - hlášení, že číslo je sudé i že je liché. Pro lichá čísla by predikát fungoval dobře, neboť jeho první část by se nespustila.

Odlišujeme tedy dva druhy řezů:

- zelený řez, který je možné odstranit, aniž by se tím změnilo fungování programu
- červený řez, který nelze odstranit bez poškození programu

? **Řezy v pravidlech zjisti1 a zjisti2 jsou...?**

Řez v pravidle *zjisti1* můžeme označit jako zelený, řez v pravidle *zjisti2* je červený řez.

Řez v mnoha případech urychlí program. Avšak jeho použití je třeba vždy zvážit. Někdy je vhodnější složit program z několika kratších, přehlednějších pravidel, než jej složitě členit pomocní mnoha řezů.

! **Napište dvě varianty pravidla - s řezem a bez řezu - pro zjištění maxima ze dvou čísel.**

Seznam - zpracování dvou seznamů

Kromě potřeby zpracovat jeden seznam se často setkáme s nutností provést manipulaci s více seznamy naráz. Velmi často musíme spojit dva seznamy nebo s nimi provést některou množinovou operaci.

Spojení dvou seznamů

```
spoj([],S,S).
spoj([H|T],S,[H|V]):-spoj(T,S,V).
```

Spojujeme dva seznamy (první a druhý argument) do seznamu v třetím argumentu. V průběhu spojování rozebíráme první seznam a jeho prvky po jednom přesunujeme do výsledného seznamu. Ve chvíli, kdy první seznam je prázdný, zareaguje první řádek a dojde ke zkopírování celého druhého seznamu do třetího argumentu.

Množinové operace

Zacházíme-li se seznamem jako s s množinou, často užíváme dříve vysvětlený predikát *člen* k ověření, zda množina obsahuje daný prvek. Nemá-li množina prvek obsahovat, kombinujeme predikát *člen* s podmínkou not. Alternativně můžeme místo podmínky not užít vhodně umístěný řez. To je vidět na dalších pravidlech pro sestrojení sjednocení a průniku, která uvádíme v obou podobách - s not i s řezem.

Sjednocení

Třetí argument má obsahovat sjednocení seznamů z prvního a druhého argumentu.

```
sjednoceni([],M,M).
sjednoceni([X|Y],L,S) :- clen(X,L),sjednoceni(Y,L,S).
sjednoceni([X|Y],L,[X|S]) :- (not(clen(X,L))),sjednoceni(Y,L,S).
```

Do sjednocení chceme vložit celý druhý seznam a některé prvky z prvního seznamu (ty, které se v druhém seznamu nevyskytují). Postupujeme tak, že rozebíráme první seznam na hlavičku X a tělo Y a zjišťujeme, zda se aktuální hlavička X vyskytuje nebo nevyskytuje v druhém seznamu L.

- Prvek X, který se v L vyskytuje, přeskočíme a voláme ihned další krok rekurze. (řádek 2)
- Prvek X, který se v L nevyskytuje, připojíme k vytvářenému sjednocení a voláme další krok rekurze. (řádek 3)
- Když se dobereme k prázdnému seznamu v prvním argumentu, přesuneme celý druhý argument M do připravovaného výsledku. (řádek 1)

Program můžeme zestručnit, když nahradíme testování členu-nečlenu řezem:

```
sjednoceni2([],M,M).
sjednoceni2([X|Y],L,S) :- clen(X,L), !, sjednoceni2(Y,L,S).
sjednoceni2([X|Y],L,[X|S]) :- sjednoceni2(Y,L,S).
```

Když požádáme o sjednocení dotazem *sjednoceni([2,3,5,7],[1,3,2,4,5],X),*

? **jak bude X vypadat? Pozor na pořadí prvků!**

X = [7,1,3,2,4,5]

Průnik

Průnikem seznamů z prvního a druhého argumentu bude seznam v třetím argumentu.

```
prunik2([],_,[]).
prunik2([X|Y],L,[X|S]) :- clen(X,L), prunik2(Y,L,S).
prunik2([X|Y],L,S) :- (not(clen(X,L))),prunik2(Y,L,S).
```

Sestrojení průniku se velice podobá sjednocení. Opět rozhodujeme, které prvky z prvního seznamu do průniku přesunout a které nikoli. Z prvního seznamu odebíráme hlavičky X a kontrolujeme, zda se vyskytují v druhém seznamu L. Jestliže se X vyskytuje v L, (druhý řádek), připojíme X k chystanému výsledku. Není-li X prvkem L (třetí řádek), znamená to, že nepatří do průniku a tedy X k výslednímu seznamu nepřidáváme. Až se dopracujeme k prázdnému seznamu v prvním argumentu, vložíme prázdný seznam i do argumentu třetího. Tím specifikujeme proměnnou S. Před S se pak při návratu z rekurze doplní hlavičky X ze všech předchozích kroků.

I tentokrát můžeme pravidlo přepsat s využitím řezu:

```
prunik([],_,[]).
prunik([X|Y],L,[X|S]) :- clen(X,L), !, prunik(Y,L,S).
prunik([_|Y],L,S) :- prunik(Y,L,S).
```

? **Umíte vysvětlit, jak řez v tomto případě funguje?**

Snad umíte :-) Je to jednoduché: přejde-li Prolog úspěšně přes druhý řádek, nebude se snažit najít alternativní řešení v řádku třetím. Proto také není třeba v třetím řádku testovat člen.

Rozdíl

Rozdíl prvního a druhého seznamu bude uložen do seznamu třetího.

```
rozdil([],X,[]).
rozdil([H|T],X,[H|Y]):-not(clen(H,X)),!,rozdil(T,X,Y).
rozdil([H|T],X,Y):-rozdil(T,X,Y).
```

Pro změnu začneme variantou s řezem

- první řádek - rozdíl prázdného seznamu a jakéhokoli seznamu je prázdný seznam
- druhý řádek - hlavičku H přesuneme do výsledného seznamu tehdy, když není obsažena v druhém seznamu X. Pak voláme rozdíl pro zbytek T prvního seznamu, druhý seznam X postupuje do dalšího kola nezměněn.
- třetí řádek - nebyl-li splněn druhý řádek, znamená to, že se hlavička H vyskytuje v seznamu X. Proto spustíme další krok, ale H do výsledku neumisťujeme.

! **Přepište predikát rozdíl bez řezu!**

Seznamy - složitější programy

Predikáty pro práci se seznamy, které jsme si vysvětlili dříve, jsme neprobírali samoúčelně: nyní je použijeme ve složitějších programech.

Permutace seznamu

```
permutace([],[]).
permutace(S,[P|Z]):-vytkni(P,S,Zb),permutace(Zb,Z).

vytkni(X,[X|T],T).
vytkni(X,[H|T],[H|T1]):-vytkni(X,T,T1).
```

Oba argumenty pravidla permutace jsou seznamy, první seznam je vstupní, druhý seznam je výstupní - je to permutace prvního seznamu. Při vytváření permutace používáme predikát vytkni.

- Permutací prázdného seznamu je opět prázdný seznam. Tento řádek slouží k ukončení běhu programu.
- Permutaci delšího než prázdného seznamu S provedeme tak, že z něj vytkneme nějaký prvek P, který připojíme jako hlavu k vytvářenému seznamu. Z toho, co zbude po vytýkání - zbytek Zb - vytvoříme permutaci Z.

Vyhledáním všech alternativ řešení získáme všechny permutace. Takto budou vypadat výsledky permutování tříprvkového seznamu [1,2,3]:

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

Otočení seznamu

```
rever([],[]).
rever([H|T],S):-rever(T,T1), spoj(T1,[H],S).

spoj([],S,S).
spoj([H|T],S,[H|T1]) :- spoj(T,S,T1).
```

Pravidlem *rever* otáčíme seznam z prvního argumentu do seznamu v druhém argumentu. Použijeme již známé pravidlo *spoj*.

- Otočením prázdného seznamu dostaneme opět prázdný seznam.
- Otočení delšího než prázdného seznamu provedeme tak, že ze seznamu odstraníme hlavičku, zkrácený seznam otočíme a hlavičku připojíme na jeho konec.

Fail - nikdy nesplněný predikát

Predikát fail není nikdy splněn, neboli je vždy nesplněn. Tedy Prolog při přechodu přes něj vždy selže a musí se vrátit. Pokud již žádná možnost návratu neexistuje, pokus splnit predikát fail končí hlášením "no".

Zdá se Vám, že použití nesplnitelného predikátu nedává smysl? Mýlíte se! Fail se výborně hodí k řízení programu - nejčastěji tehdy, když chceme získat naráz všechna řešení, aniž bychom po každém jednotlivém výpisu museli zadáním středníku žádat o další.

vypis:-cislo(A),write(A),nl,fail.

Pravidlo *vypis* vypíše všechna čísla obsažená v databázi: nalezne v databázi *cislo (A)*, vypíše je, odřádkuje a narazí na fail. Přes něj nemůže přejít, takže se vrací. Odřádkování ani výpis jinak provést nemůže, hledá tedy jiné číslo. Pokud je najde, provede další výpis, odřádkování a opět je u failu. Vrací se tolikrát, kolikrát se podaří nalézt v databázi číslo. Po posledním výpisu následuje hlášení "no".

? **Co myslíte, že se stane, když výše uvedené pravidlo doplníme takto:**
vypis:-cislo(A),write(A),nl,fail,write(' je cislo') . ?

Nestane se nic. Prolog se bude opakovaně snažit přejít přes fail a přitom narážet na fail. Po posledním možném výpisu skončí celé pravidlo neúspěchem. Poslední write se neprovede ani jednou.

? **A co se stane, jestliže napíšeme:**
vypis(X):-cislo(X),fail. ?

Program skončí neúspěchem, aniž by cokoli vypsal - výpis proměnné X je možný pouze po úspěšném ukončení pravidla.

To, že fail není nikdy splněn, má dva důsledky:

- Za prvé, fail používáme vždy na konci pravidla. Prolog přes něj totiž nemůže přejít a tedy se ani nemůže dostat k případným následujícím predikátům.
- Za druhé, pokud potřebujeme, aby pravidlo, v němž se fail vyskytuje, končilo úspěchem (odezvou yes), musíme dopsat ještě prázdný řádek, kterým zařídíme, že celé pravidlo skončí úspěšně:

vypis:-cislo(A),write(A),nl,fail.
vypis:-write('konec výpisu'),nl.

...Po vypsání všech čísel dojde na druhou větev pravidla, která proběhne úspěšně.

! **Otevřete si databázi českých panovníků. Napište predikát pro výpis všech králů, kteří vládli déle než deset let!**

Práce s databází - assert a listing

Assert

Chceme-li změnit obsah databáze, s kterou Prolog pracuje, používáme vkládací příkaz assert. V LPA-Prologu se setkáme se třemi variantami příkazu vkládání:

- assert - vložení na začátek databáze
- asserta - vložení na začátek databáze
- assertz - vložení na konec databáze

Příkazy assert a asserta pracují stejně, pouze v názvu druhého příkazu je zdůrazněno, že vkládání se provede na začátek databáze.

? Proč záleží na pořadí vkládání faktů do databáze?

Protože když se Prolog snaží zodpovědět dotaz, prochází databází shora dolů. Tím, že řekneme, kam se má nový fakt zapsat, zároveň stanovíme, jak brzy (resp. jako kolikáté řešení) bude tento fakt nalezen při odpovídání na dotaz.

Vkládání provádíme buď z příkazového řádku Prologu:

?- asserta(cislo(1)).

nebo přímo v programu, uvnitř pravidla:

vlozeni:-asserta(cislo(1)), write('bylo vlozeno cislo 1'),nl.

Listing

K výpisu obsahu databáze slouží příkaz listing. Pokud jej použijeme v základním tvaru, obdržíme obsáhlý výpis včetně cest k adresářům, což může být nepřehledné. Proto někdy dáváme přednost upřesněnému volání, v němž uvedeme název a případně i aritu predikátu, který chceme vypsat.

?- listing. ... vypíše obsah celé databáze

?- listing(cislo). ... vypíše všechny predikáty "cislo"

?- listing(cislo/1). ... vypíše yes, pokud bude v databázi nalezen predikát "cislo" s aritou 1

Příkazy assert a listing můžeme kombinovat ve složitějších pravidlech. Příkazem assert vkládáme, příkazem listing provádíme průběžné výpisy, jimiž se ujistíme, že vložení správně proběhlo.

Následující pravidlo vloží do databáze čísla od 1 do X, uspořádaná vzestupně, tj. databáze začíná číslem 1 a pokračuje vyššími čísly.

```
nahoru(1):-asserta(cislo(1)).
nahoru(X):-X>1,X1 is X-1,asserta(cislo(X)),nahoru(X1).
```

Při prvním zavolání, řekněme nahoru(5), zareaguje druhý řádek pravidla. Je

vloženo číslo 5 a znovu zavolán predikát nahoru(4). V druhém volání dojde k vložení čísla 4 opět příkazem asserta, tedy na začátek databáze - nad pětku. Obdobně program pokračuje a skončí po vložení čísla 1, po kterém již nedojde k dalšímu kroku rekurze.

! **Napište pravidlo, jímž do databáze vložíte čísla 1 až X uspořádaná sestupně, tedy od největšího po nejmenší.**

Práce s databází - retract a abolish

Retract

Příkazem retract odstraňujeme fakty z databáze. Jako argument předáváme fakt, který má být vymazán.

retract(cislo(X))... odstranění čísla

LPA-Prolog má, pokud se týká obsluhy databáze, jednu zvláštnost: příkaz retract je v něm aktivní i při navracení. To může způsobovat problémy. Kdybychom potřebovali vypsat a současně smazat všechny čísla z databáze a napsali:

hledej-cisla:-cislo(X),retract(cislo(X)),write(X),fail.,

program by správně nefungoval. Po nesplnění failu by se Prolog při navracení snažil znovu zmazat číslo X, což by už nebylo možné. Proto si musíme pomoct modifikovaným příkazem, který si sami vytvoříme:

retract1(X):- retract(X),!.

Program přepíšeme na tvar:

hledej-cisla:-cislo(X),retract1(cislo(X)),write(X),fail.

Abolish

Příkaz abolish je "hromadnou obdobou" příkazu retract. Slouží ke smazání všech predikátů daného jména a arity.

abolish(cislo/1)... odstranění všech predikátů číslo s aritou jedna

Práce s databází - abolish, assert a retract

Nyní musíme zmínit další zvláštnost LPA Prologu, a sice to, že pracuje s dvěma databázemi:

- jednu si vytvoří při překladu programu - uloží do ní fakty, které jsou v předkládaném kódu definovány
- druhou udržuje za běhu programu a vkládá do ní (vyjímá z ní) tehdy, když v pravidlech narazí na assert nebo retract.

Databáze jsou přitom oddělené a část, vytvořená při překladu, není dosažitelná pro assert, retract, abolish ani listing. (Zkuste nahrát do Prologu databázi českých králů a spustit listing!) To znamená, že všechna fakta, která budeme uvedenými čtyřmi příkazy obsluhovat, musí být připraveny v druhé databázi. Aby tomu tak bylo, kombinujeme v programech často příkazy abolish a assert.

```
start:-abolish(cislo/1), asserta(cislo(1)), asserta(cislo(2)), asserta(cislo
(3)).
```

V uvedené ukázce se nejprve odstraní z databáze dříve uložené údaje o číslech a poté se nově zaznamenají tři fakta.

? **Mohl by příkaz abolish být splněn i tehdy, kdyby databáze žádné starší záznamy o číslech neobsahovala?**

Ano, příkaz abolish bude splněn i tehdy, když nedojde k odstranění žádného čísla. Na počtu čísel nezáleží.

Kombinaci abolish - assert - retract využíváme hlavně tehdy, když potřebujeme vytvořit počítadlo, jako v následujícím programu, který zjišťuje, kolik záznamů o číslech je obsaženo v databázi.

```
kolik:-

        abolish(pocitadlo/1),
        asserta(pocitadlo(0)),
        cislo(C),
        retract1(pocitadlo(P)),
        P1 is P+1,
        asserta(pocitadlo(P1)),fail.

kolik:- retract(pocitadlo(P)),write(P),nl.

retract1(X):- retract(X),!.
```

Fungování programu je následující. Nejprve z databáze smažeme případné starší počítadlo a vložíme nové, s hodnotou nula. Následně zjišťujeme, jestli databáze obsahuje nějaké číslo. Pokud ano, počítadlo vyjmeme, k jeho hodnotě přičteme jedničku a nové počítadlo opět vložíme. Pak program selže, protože nedokáže přejít přes fail a vrací se. Při návratu nelze provést jinak ani součet P+1, ani retract1. Může ale být nalezeno nové číslo. Tehdy se program pohne opět dopředu - dojde k navýšení počítadla. Totéž se opakuje tak dlouho, dokud Prolog nalézá

nová čísla v databázi. Při konečném neúspěchu (po nenalezení dalšího čísla a fail) je čas vyzkoušet druhou větev pravidla *kolik*. V ní dojde k vyjmutí počítadla a výpisu jeho aktuální hodnoty, která současně udává celkový počet nalezených čísel.

! **Jak musíme připravit databázi českých panovníků, abychom mohli řádky v ní počítat, třeba zjišťovat počet králů, kteří vládli déle než pět let?**

Vestavěné predikáty - seskupování

Predikátem fail můžeme Prolog přimět, aby vypsal všechna řešení naráz, aniž bychom si každý výsledek vyžádali zadáním středníku. Jestliže potřebujeme ještě víc, a sice uložit výsledky do společné struktury (do seznamu výsledků), fail nestačí. Možné by bylo dílčí výsledky zapisovat do databáze příkazem assert a následně je je vyjímat příkazem retract a po jednom přidávat do seznamu. Ale takto pracný postup není nutný: v Prologu jsou totiž připraveny standardní predikáty findall, setof a bagof.

Findall

Findall(C,Podminky,S) má tři argumenty:

- cíl C,který má být splněn
- soupis podmínek, jimiž musí C vyhovovat
- seznam S pro nalezená C

Uvedeme dva příklady:

```
prvni:-findall(Jmeno,(kral(Jmeno,_,_)),X),write(X),nl.
druhy:-findall(Jmeno,(kral(Jmeno,Od,Do),Do-Od>=10),X),write(X),nl.
```

První program vyhledá všechna jména králů a uloží je do seznamu X. Druhý program vyhledá pouze jména těch králů, kteří vládli deset a více let. Výsledkem volání obou programů bude výpis seznamu X, jehož obsah samozřejmě závisí na rozsahu použité databáze. Může to být třeba:

```
X = [Premysl Otakar I.,Vaclav I.,Premysl Otakar II.,Vaclav II.]
```

Jakým programem byste sestavili seznam letopočtů, v kterých nastoupil k vládě nějaký král?

?

Například: treti:-findall(Rok,(kral(_,Rok,_)),X),write(X),nl.

Než si vysvětlíme další dva predikáty, ukážeme si ještě jeden příklad použití findall. Připravíme si databázi, do které zapíšeme fakty rodič s dvěma argumenty: jméno rodiče a jméno potomka:

```
rodic(jan,eva).
rodic(jan,lenka).
rodic(jiri,karel.
rodic(jiri,david).
rodic(jiri,petr).
```

Zavoláním

```
findall(A,(rodic(A,B)),Rodice).
```

anebo

```
findall(A,(rodic(A,_)),Rodice).
```

zjistíme seznam všech lidí, kteří jsou rodiči. Obě volání navrátí stejný seznam:

```
Rodice = [jan,jan,jiri,jiri,jiri]
```

Jestliže se některé jméno A vyskytuje v databázi opakovaně, objeví se opakovaně i v seznamu Rodice.

Když naopak zavoláme

```
findall(B,(rodic(A,B)),Deti).
```

nebo

findall(B,(rodic(_,B)),Deti).

získáme stejný seznam všech dětí z databáze:

Deti = [eva,lenka,karel,david,petr]

Bagof a setof

Predikát *bagof* se chová trochu jinak a můžeme jej použít dvěma způsoby, což si předvedeme opět na databázi rodičů a dětí.

Za prvé, můžeme naplnit hodnotu proměnné, které je součástí podmínek a pro tuto hodnotu zjistit seznam hodnot jiné proměnné. V ukázce je vidět, jakým způsobem probíhá hledání rodiče A a všech jeho dětí B, které zapisujeme do seznamu Deti. Abychom postupně vypsali jména rodičů a seznamy dětí, musíme opakovaně vyžadovat další výsledky zadáváním středníku:

| ?- bagof(B,(rodic(A,B)),Deti).
B = _ ,
A = jan ,
Deti = [eva,lenka] ;

B = _ ,
A = jiri ,
Deti = [karel,david,petr] ;

no:

Za druhé, můžeme požádat o jediný seznam, v němž jsou sloučeny seznamy výsledků pro různé hodnoty určité proměnné. Voláme:

bagof(B,(A^rodic(A,B)),Deti).

a dostaneme

Deti = [eva,lenka,karel,david,petr]

Podobně jako bagof funguje setof, který navíc setřídí výsledný seznam a případně z něj odstraní duplicitní prvky. Odpovědí na volání:

setof(B,(A^rodic(A,B)),Deti).

je

Deti = [david,eva,karel,lenka,petr]

! **Napište program, kterým sestavíte seznam králů, kteří vládli před rokem 1306.**

První program

První program, kterým začíná většina tutoriálů a sbírek příkladů, uživatele většinou požádá, aby se představil, a poté jej mile pozdraví.

Řešení

```
hello:-

    write('Napiste, prosim, Vase jmeno:'),
    read(Jmeno),
    write('Vyborne ! '), nl,
    write(Jmeno), write(' je hezke jmeno !'), nl,
    write('Tesim se na dalsi spolupraci ... '),
    getb(Char).
```

Komentář

Jak vidíte, program tvoří jediný predikát, obsahující příkazy write (výpis na obrazovku), read (načtení z klávesnice) a getb (reaguje na stisk libovolné klávesy).

Cvičení

Na tomto jednoduchém příkladě si lze zkoušet různé možnosti vstupu a výstupů. Upravte stávající program tak, aby od uživatele získal více informací a vypsal je potom znovu na obrazovku.

Trochu historie

Predikátem kral(Jmeno,Od,Do) jsou zadáni čeští králové včetně období, ve kterém panovali. Úkolem je napsat predikáty:

1. panoval(Rok,Kral) = kdo panoval v zadaném roce ?
2. kolik_let_vladl(Kral,Kolik)
3. predchudce(X,Y)
4. naslednik(X,Y)

Řešení

```
kral('Premysl Otakar I.',1197,1230).
kral('Vaclav I.',1230,1253).
kral('Premysl Otakar II.',1253,1278).
kral('Vaclav II.',1278,1305).
kral('Vaclav III.',1305,1306).
kral('Jindrich Korutansky',1306,1306).
kral('Rudolf I. Habsbursky',1306,1307).
kral('Jindrich Korutansky',1307,1310).
kral('Jan Lucembursky',1310,1346).
kral('Karel IV.',1346,1378).
kral('Vaclav IV.',1378,1419).

panoval(Rok,Kral) :-

    kral(Kral,R1,R2), Rok >= R1, Rok =< R2.

kolik_let_vladl(Kral,X) :-

    kral(Kral,R1,R2), X is R2 - R1.

predchudce(X,Y) :-

    kral(X,R,_), kral(Y,_,R), X \= Y.

naslednik(X,Y) :-

    predchudce(Y,X).
```

Komentář

- kdo panoval v daném roce - Známe rok (Rok) a hledáme krále, jehož jménem chceme naplnit proměnnou Král. Přitom požadujeme, aby daný Rok ležel mezi počátečním rokem R1 a koncovým rokem R2 vlády Krále.
- kolik let vládl král - Známe krále (Král), chceme naplnit proměnnou X (délky vlády). V databázi vyhledáme krále podle jména a jeho počátečním a koncovým rokem vlády naplníme proměnné R1,R2. Poté vypočteme hledané X.
- předchůdce - Král Y je předchůdcem krále X tehdy, když koncový rok vlády krále Y je shodný s počátečním rokem vlády krále X. Použijeme tedy stejně pojmenovanou proměnou R pro označení roku. Podmínka X\=Y zajistí, že nebudou vypsána triviální řešení, tedy skutečnost, že každý král je svým vlastním předchůdcem.
- následník - Víme-li, že X je předchůdcem Y, víme ihned i to, že Y je následníkem X.

Cvičení

1. Vymyslete si a implementujte další dotazy nad výše uvedenou databází.
2. Doplňte výsledný program tak, aby se po spuštění a poté vždy po zodpovězení dotazu na obrazovku vypsala nabídka všech možných dotazů, ze které bude možno vybírat stiskem určité klávesy, čímž se program stane uživatelsky přívětivějším.

Hitparáda

Pomocí predikátu song(Interpret,Song,Poradi), je zadána část žebříčku britské hitparády. Aktuální pořadí se ovšem dostatečně pomíchalo a stav je naprosto nepřehledný. Napište program, který přehledně vypíše první desítku.

Řešení

```
song(arrested_development, people_everyday,2).
song(sharmen, boss_drum,3).
song(erasure, who_needs_love_like_that,7).
song(bon_jovi, keep_the_faith,10).
song(madonna, erotica,9).
song(rage, run_to_you,4).
song(charles_and_eddie, would_i_lie_to_you,1).
song(vanessa_paradis, be_my_baby,8).
song(take_that, a_milion_love_songs, 5).
song(whitney_houston, i_will_always_love_you,6).

top_ten :-

    write('Prvni desitka britske hitparady - '),
    napis_datum, nl, nl,vypis(1, 10).

top_ten.

vypis(Cislo, Poc) :-

    Cislo =< Poc,
    song(Interpret,Song,Cislo),
    write(Cislo), write('. '),
    write(Interpret), write(' - '), write(Song),nl,
    Dalsi is Cislo+1, vypis(Dalsi, Poc).

napis_datum :-

    date(Den, Mesic, Rok),
    write(Den), write('.'),
    write(Mesic), write('.'),
    write(Rok), write('.').
```

Komentář

- Program zavoláme predikátem *top_ten*. Dojde k výpisu řetězce a na stejný řádek je vypsáno i aktuální datum. Následuje dvojí odřádkování a výpis hitparády.
- K výpisu data slouží predikát *napis_datum*. Tento predikát pracuje s funkcí date, která naplní proměnné Den, Měsíc a Rok. Hodnoty proměnných jsou vypsány.
- Vlastní výpis hitparády provádí predikát *vypis*. Při prvním zavolání do predikátu předáme hodnoty 1 (požadavek na výpis prvního místa v hitparádě) a 10 (celkový počet míst, který chceme vypsat). Po vyhledání a výpisu řádku se zvýší hodnota prvního argumentu o 1 a znovu je volán predikát *vypis.*
- Po vypsání celé desítky už nelze predikát *vypis* úspěšně vykonat a tedy vypisování i celý predikát *top_ten* končí - z pohledu Prologu - neúspěchem. Proto připojujeme ještě prázdný predikát *top_ten.* Díky němu je činnost programu ukončena úspěšným hlášením "yes".

Cvičení

Upravte stávající program tak, aby při chybějící informaci o obsazení N-tého místa vypsal na daný řádek např.: N. --- nedostatečné zadání ---

Má rád - nemá rád

Uvažujme skupinu chlapců a dívek (jsou zadány pomocí predikátů kluk a holka). Pokud některý z chlapců (X) cítí jisté sympatie k některé z dívek (Y), je v databázi i příslušný predikát ma_rad(X,Y). Napište procedury odpovídající následujícím dotazům:

1. který kluk má rád Ivanu?
2. koho má ráda Jana?
3. koho nemá rád Jiří?
4. kdo nemá rád Jitku?

Máte zadány následující skutečnosti:

```
kluk(martin).
kluk(karel).
kluk(jiri).
kluk(ales).
kluk(tomas).
holka(alena).
holka(jana).
holka(ivana).
holka(dana).
holka(jitka).
ma_rad(jiri,jana).
ma_rad(jiri,ivana).
ma_rad(karel,dana).
ma_rad(karel,jana).
ma_rad(karel,ivana).
ma_rad(karel,jitka).
ma_rad(karel,alena).
ma_rad(ales,jana).
ma_rad(ales,ivana).
ma_rad(ales,jitka).
```

Řešení

```
dotaz1(X) :- % kdo ma rad ivanu

    kluk(X), ma_rad(X,ivana).

dotaz2(X) :- % koho ma rada jana

    write('Nelze zjistit!'), nl.

dotaz3(X) :- % koho nema rad jiri

    holka(X), not(ma_rad(jiri,X)).

dotaz4(X) :- % kdo nema rad jitku

    kluk(X), not(ma_rad(X,jitka)).
```

Komentář

- dotaz1 - Hledáme kluka X, který se vyskytuje ve dvojici s Ivanou.

- dotaz2 - V databázi není predikát, který by popisoval vztah dívek k chlapcům, takže odpověď nelze zjistit.
- dotaz3 - Hledáme holku X, která se nevyskytuje ve dvojici s Jiřím. K vyjádření skutečnosti, že databáze neobsahuje určité tvrzení, slouží predikát not.
- dotaz4 - Hledáme kluka X, který se nevyskytuje ve dvojici s Jitkou.

Cvičení

Vymyslete si a implementujte alespoň dva další dotazy

Kriminální případ

Na večírku, kterého se zúčastnila uzavřená společnost, se stala vražda. Přivolaný policejní komisař zjistil, že Zuzana, která se stala obětí trestného činu, přijela na večírek ve stejném automobilu jako její vrah. Motivem vraždy byla žárlivost a vrah byl starší než Robert. Sebevraždu komisař vyloučil.
Napište program, který odhalí vraha, máte-li zadáno:

```
% účastníci večírku
osoba(robert,22,muz,fiat).
osoba(adam,25,muz,audi).
osoba(jiri,25,muz,fiat).
osoba(tomas,25,muz,audi).
osoba(lucie,24,zena,fiat).
osoba(kveta,21,zena,fiat).
osoba(alena,25,zena,audi).
osoba(zuzana,24,zena,audi).
% milenecké vztahy mezi účastníky večírku
vztah(kveta,jiri).
vztah(kveta,adam).
vztah(lucie,jiri).
vztah(lucie,tomas).
vztah(alena,robert).
vztah(zuzana,jiri).
vztah(zuzana,robert).
```

Řešení

```
vrah(X) :-

    osoba(X,Vek,_,Auto),
    osoba(zuzana,_,_,Auto),
    osoba(robert,Vek1,_,_),
    podezreni(X,Motiv),
    Vek > Vek1,
    X \= zuzana.

podezreni(X,zarlivost) :- % zarlivy muz

    osoba(X,_,muz,_),
    milenci(zuzana,X),
    milenci(zuzana,Y), X \= Y.

podezreni(X,zarlivost) :- % zarliva zena

    osoba(X,_,zena,_),
    milenci(X,Y),
    milenci(zuzana,Y).

milenci(X,Y) :- vztah(X,Y).
milenci(X,Y) :- vztah(Y,X).
```

Komentář

Program spustíme predikátem *vrah(X).* Zajímá nás osoba X. Nevíme, zda vrahem je

muž nebo žena, ale víme něco o věku a autě vraha. Proto zavedeme proměnné Věk a Auto a přidáme podmínky, jimiž obsah proměnných upřesníme (věk Roberta je vyšší, než věk vraha, Zuzana a vrah mají stejné auto. Predikátem *podezreni* vyjádříme obecně skutečnost, že vrah X měl určitý motiv. Pro konkrétní motiv "žárlivost" pak predikát definujeme. Přitom využíváme predikát *milenci*.

Cvičení

Vymyslete si jinou detektivní historku a napište příslušný program.

Barvení mapy

Vytvořte program, který bude generovat všechna možná obarvení mapy střední Evropy (ČR, SR, Polsko, Maďarsko, Rakousko, Německo) třemi barvami (červená, modrá, žlutá). Samozřejmou podmínkou je obarvení sousedních zemí různými barvami.

Řešení

```
obarvi(Cesko,Slovensko,Polsko,Rakousko,Nemecko,Madarsko) :-

    sousede(Cesko,Slovensko),
    sousede(Cesko,Polsko),
    sousede(Cesko,Rakousko),
    sousede(Cesko,Nemecko),
    sousede(Slovensko,Polsko),
    sousede(Slovensko,Rakousko),
    sousede(Slovensko,Madarsko),
    sousede(Polsko,Nemecko),
    sousede(Rakousko,Nemecko),
    sousede(Rakousko,Madarsko).

sousede(X,Y) :- barvy(X,Y).
sousede(X,Y) :- barvy(Y,X).

barvy(cervena,modra).
barvy(cervena,zluta).
barvy(modra,zluta).
```

Komentář

V predikátu *obarvi* použijeme šest proměnných pro hledaných šest barev. Predikátem *sousede* definujeme mapu - musíme vypsat všechny dvojice spolu sousedících států. Přitom každé dvojic sousedů má odpovídat některá dvojice barev.

Cvičení

1. Napište alespoň jednu další variantu řešení zadaného problému.
2. Upravte stávající program tak, aby bylo možné brát v úvahu doplňující podmínky typu Německo bude červené, Slovensko nebude modré atd.

Pythagorejská čísla

Napište program, který vypíše všechna pythagorejská čísla a,b,c taková, že největší číslo z trojice (délka přepony pravoúhlého trojúhelníka) je menší nebo rovné předem zadanému číslu.

Řešení

```
pyth(N) :-

    N > 0,
    gen(A,1,N),
    gen(B,A,N), % A < B
    gen(C,B,N), % B < C
    C * C =:= A * A + B * B ,
    write(A:B:C), nl,
    fail.

pyth(_) :-

    write('Konec vypisu !'), nl.

gen(M,M,N) :- M=<N.
gen(X,M,N) :- M=<N, M1 is M+1, gen(X,M1,N).
```

Komentář

- Program spustíme predikátem *pyth*. Bude probíhat hledání a vypisování trojic Pythagorejských čísel. Každá trojice musí být vzestupně uspořádána a musí splňovat podmínku Pythagorovy věty. Po nalezení vyhovující trojice Prolog narazí na nesplnitelný predikát fail, takže se vrátí a bude hledat jinou trojici. Aby program po dohledání všech trojic neskončil hlášením "no", připojujeme druhou část predikátu *pyth* s výpisem řetězce a odřádkováním. Tato část zároveň řeší požadavek *pyth(0)*.
- Vlastní generování čísla provádí predikát *gen*. Kromě proměnné pro hledané číslo do něj předáváme zarážky pro omezení čísla zdola (jde o předchozí číslo v trojici: A při generování čísla B a B při generování čísla C) a shora - zarážka N. Tím, že si přejeme trojici A,B,C uspořádat podle velikosti, zabráníme výpisům stejných, jen jinak uspořádaných trojic čísel.

Cvičení

1. Vygenerujte všechna přirozená čísla a,b,c taková, že trojúhelník o stranách délky a,b,c má obvod menší než zadané číslo N.
2. Vygenerujte všechna přirozená čísla a,b,c taková, že kvádr o hranách délky a,b,c má stejný povrch i objem.

Magický čtverec

Napište program, který umístí do mřížky o rozměru 3x3 buňky čísla 1 až 9 tak, že součet ve všech sloupcích, řádcích i úhlopříčkách bude stejný. Zkuste řešit úlohu dvěma způsoby:

1. pomalejším postupem, kdy pouze nadefinujete, které součty hodnot proměnných se sobě mají rovnat
2. a rychleji, s využitím úvahy, že součet čísel ve sloupci, řádku i úhlopříčce je přesně...víte kolik?

Řešení 1. - pomalejší

```
ctverec1 :- % pomalejsi algoritmus

    Cisla = [1,2,3,4,5,6,7,8,9],
    vytkni(A,Cisla,Zb1),
    vytkni(B,Zb1,Zb2),
    vytkni(C,Zb2,Zb3),
    A < C, % nechci pootocena reseni
    S is A+B+C, % 1. radek
    vytkni(D,Zb3,Zb4),
    vytkni(E,Zb4,Zb5),
    vytkni(F,Zb5,Zb6),
    S is D+E+F, % 2. radek
    vytkni(G,Zb6,Zb7),
    C < G, % nechci pootocena reseni
    vytkni(H,Zb7,Zb8),
    vytkni(I,Zb8,[]),
    S is G+H+I, % 3. radek
    S is A+D+G, % 1. sloupec
    S is B+E+H, % 2. sloupec
    S is C+F+I, % 3. sloupec
    S is A+E+I, % uhlopricka
    S is C+E+G, % uhlopricka
    write(A:B:C), nl,
    write(D:E:F), nl,
    write(G:H:I), nl,
    nl, fail.

ctverec1 :- write('Zadna dalsi reseni.'), nl.

vytkni(H,[H|T],T).
vytkni(X,[H|T],[H|T1]) :- vytkni(X,T,T1).
```

Řešení 2. - rychlejší

```
ctverec2 :-

    Cisla = [1,2,3,4,5,6,7,8,9],
    vytkni(A,Cisla,Zb1),
    vytkni(B,Zb1,Zb2),
    C is 15-A-B, % 1. radek
```

```
    vytkni(C,Zb2,Zb3),
    A < C, % nechci pootocena reseni
    vytkni(D,Zb3,Zb4),
    G is 15-A-D, % 1. sloupec
    C < G, % nechci pootocena reseni
    vytkni(G,Zb4,Zb5),
    E is 15-C-G, % uhlopricka
    vytkni(E,Zb5,Zb6),
    I is 15-A-E, % uhlopricka
    vytkni(I,Zb6,Zb7),
    F is 15-D-E, % 2. radek
    15 is C+F+I, % 3. sloupec
    vytkni(F,Zb7,Zb8),
    H is 15-G-I, % 3. radek
    15 is B+E+H, % 2. sloupec
    vytkni(H,Zb8,[]),
    write(A:B:C), nl,
    write(D:E:F), nl,
    write(G:H:I), nl,
    nl, fail.

ctverec2 :- write('Zadna dalsi reseni.'), nl.

vytkni(H,[H|T],T).
vytkni(X,[H|T],[H|T1]) :- vytkni(X,T,T1).
```

Komentář

- Řešení 1. - Hledáme čísla A,B,C,D,E,F,G,H,I, jimiž po řádcích vyplníme buňky čtverce. Čísla postupně vytýkáme ze seznamu čísel 1 - 9. Protože každé další číslo vytkneme z již zkráceného seznamu, je zajištěno, že čísla budou navzájem různá. Čísla do čtverce píšeme zleva doprava a shora dolů a současně (v rámci možností) podle velikosti - tím se vyhneme výpisu různě pootočených, ale jinak stejných řešení. Všechny řádky, sloupce i úhlopříčky mají dávat stejný součet S.
- Řešení 2. - Součet čísel 1 až 9 je 45. Součty ve třech různých řádcích mají být shodné. Proto každý řádek musí mít součet 15. Také součty ve sloupcích a v úhlopříčkách by měly být rovny 15. Do řešení tedy místo obecného požadavku na součet S píšeme rovnou požadavek na součet 15. Zároveň můžeme urychlit naplňování proměnných tím, že při znalosti A, B rovnou dopočítáme C jako 15-A-B a pod.

Cvičení

1. Vygenerujte magický čtverec o rozměru 4x4. Zjistěte kolik různých čtverců lze nalézt.
2. Představte si rovnostranný trojúhelník, který má v každém vrcholu a uprostřed každé hrany vepsánu jednu číslici z intervalu 1 až 6, přičemž každou z uvedených číslic je nutné použít právě jedenkrát. Napište program, který zjistí, kolik existuje navzájem různých trojúhelníků takových, že součet číslic na všech hranách je stejný.

Fibonacciho posloupnost

Napište program, který bude počítat N-tý člen Fibonacciho posloupnosti, zadané takto:

```
fib(0) = 1
fib(1) = 1
fib(N) = fib(N-1) + fib(N-2)
```

Řešení 1. - málo efektivní

```
fib(0,1).
fib(1,1).
fib(N,K) :-

    N > 1,
    N1 is N - 1,
    N2 is N - 2,
    fib(N1,L1),
    fib(N2,L2),
    K is L1 + L2.
```

Řešení 2. - vhodnější

```
fibo(N,X) :-

    N >= 0,
    fibo(0,1,1,N,X).

fibo(N,X,_,N,X) :- !.
fibo(I,J,K,N,X) :-

    I1 is I + 1, K1 is K + J,
    fibo(I1,K,K1,N,X).
```

Komentář

- Řešení 1. - Generování čísel probíhá shora. Ke zjištění N-tého prvku posloupnosti potřebujeme znát předchozí prvky L1 s indexem N1 = N-1 a L2 s indexem N2 = N-2. Sečtením prvků L1 a L2 dostaneme hledané K. Pro určení L1 a L2 voláme dvakrát predikát *fib*. Problém je, že i generování prvků L1 a L2 se rozdělí na hledání jejich dvou předchůdců, čili dalších čtyř čísel, k určení čtyř čísel je třeba znát dalších osm atd. Velmi rychle dojde k přeplnění zásobníku a program tedy lze prakticky použít nejvýš pro generování patnáctého členu Fibonacciho posloupnosti.
- Řešení 2. - Zde naopak postupujeme zdola. Do dalších volání predikátu předáváme vždy dva prvky posloupnosti a současně zvyšujeme hodnotu indexu. V predikátu *fib* vystupuje pět proměnných pro:
  1. počítadlo I, začínající na hodnotě 0, čili u nultého prvku posloupnosti
  2. prvek J posloupnosti, jehož index je roven hodnotě I
  3. prvek K posloupnosti, jehož index je roven hodnotě I+1
  4. zarážka N pro ukončení generování prvků posloupnosti
  5. proměnná X pro uložení výsledku

  Při prvním zavolání predikátu dojde ke zvýšení hodnoty počítadla o 1 (vzniká nové I1 = I+1) a k sečtení dvou prvků posloupnosti (K1 = J+K). Je znovu volán predikát *fib*, avšak s hodnotami I1, K, K1. Hledání končí, jestliže se shoduje index

se zarážkou. Pak lze naplnit proměnnou X.

Cvičení

Napište analogické programy pro výpočet hodnoty jiné rekurzivně definované posloupnosti, například:

f1(0) = 2
f1(1) = 3
f1(N) = f1(N-1) * f(N-2)

nebo

f2(0) = 1
f2(1) = 2
f2(2) = 3
f2(N) = f2(N-1) + f2(N-2) + f2(N-3)

Faktoriál

Napište program, který bude počítat faktoriál daného čísla N.

Řešení 1. - shora, bez použití řezu

```
fakt(1,1).
fakt(N,X):-

    N>1,
    N1 is N-1,
    fakt(N1,X1),
    X is N*X1.
```

Řešení 2. - shora, s použitím řezu

```
fakt2(1,1):-!.
fakt2(N,X):-

    N1 is N-1,
    fakt2(N1,X1),
    X is N*X1.
```

Řešení 3. - zdola

```
fakt3(N,X):-

    fakt(1,1,N,X).

fakt(N,X,N,X).
fakt(Poc,Dosud,Zarazka,Vysledek):-

    Poc2 is Poc+1,
    Dosud2 is Dosud*Poc2,
    fakt(Poc2,Dosud2,Zarazka,Vysledek).
```

Cvičení

Změňte program, aby počítal součiny všech lichých / sudých čísel od 1 do N.

Histogram

Vytvořte program, který na základě zadaného číselného seznamu vypíše odpovídající histogram.

Řešení

```
histogram([]) :- nl.
histogram([A|T]) :-

    pis(A),
    histogram(T).

pis(N) :-

    N > 0,
    write('x'),
    N1 is N - 1,
    pis(N1).

pis(0) :- nl.
```

Komentář

Predikátu *histogram* předáme seznam, z nějž chceme histogram vytvořit. V rekurzi je seznam rozdělen na hlavu A a tělo T. Pro hlavu A je predikátem *pis* vypsán řádek znaků "x", pro tělo T je znovu volán predikát histogram. Predikát *pis* postupně snižuje hodnotu argumentu N a při každé dekrementaci vypíše znak "x". Až je argument snížen na 0, dojde k odřádkování.

Cvičení

1. Upravte stávající program tak, aby bylo možné volit vypisovaný grafický symbol.
2. Upravte stávající program tak, aby bylo možné volit vypisovaný grafický symbol pro každou hodnotu zvlášť.
3. Vytvořte program, který bude jednotlivé hodnoty histogramu zobrazovat vertikálně.

Inzerát

Alena si podala inzerát k seznámení. Nabídek přišlo obrovské množství a Alena nyní vybírá vhodné kandidáty. Nejprve si stanovila podmínku, že vyhovujícími budou pouze nekuřáci. Poté ale dospěla k názoru, že by touto restrikcí vyloučila některé velmi zajímavé nabídky, a rozhodla se kouření tolerovat v případě, že daný kandidát je vysokoškolák s vlastním rodinným domkem. Pomozte Aleně vybrat vhodné nápadníky (pokud kandidát splňuje obě podmínky, vypište jej pouze jedenkrát). Pracujte s následující databází:

```
kandidat(adam).
kandidat(pavel).
kandidat(david).
kandidat(ivo).
kandidat(tomas).
kandidat(karel).
kurak(ivo).
kurak(david).
kurak(karel).
vysokoskolak(adam).
vysokoskolak(david).
vysokoskolak(karel).
vlastni_dum(tomas).
vlastni_dum(ivo).
vlastni_dum(david).
```

Řešení

```
partner :-

    kandidat(X), splnuje_podminku(X),
    write('mozny partner pro alenu je '),
    write(X), nl, fail.

splnuje_podminku(X) :- not(kurak(X)), !.
splnuje_podminku(X) :- vysokoskolak(X), vlastni_dum(X).
```

Komentář

- Program voláme predikátem *partner*. Protože jeho cílem není naplnit žádnou proměnnou, musíme použít predikát fail k získání všech řešení.
- Za vhodného partnera se považuje kandidát X, splňující určitou podmínku. Podmínkou je buď nekuřáctví, nebo vysokoškolské vzdělání a vlastnictví domku. Za první část predikátu *splnuje_podminku* proto umístíme řez - tím zajistíme, že u nekuřáků nebudou prověřovány další podmínky.

Cvičení

1. Rozviňte uvedený příklad a zkoušejte implementovat komplikovanější podmínky výběru kandidátů.
2. Řešte analogický problém - výběr vozu v autobazaru. Vytvořte databázi nabízených aut (evidujte značku, stáří, najeté kilometry, barvu a cenu). Stanovte několik kritérií, podle kterých chcete auto vybírat, například "staré nejvýš pět let a zároveň modré", nebo "libovolně staré, avšak ne dražší, než 300 000 Kč".

Kvadratická rovnice

Napište program pro řešení kvadratické rovnice.

Řešení

```
rovnice(0,B,C) :-

    !, B \= 0, X is -C/B,
    write('Jedine reseni:'),
    write('x='),write(X), nl.

rovnice(A,B,C) :-

    Det is B*B-4*A*C,
    reseni(A,B,Det), nl.

reseni(_,_,D) :-

    D < 0, !,
    write('Nema reseni v oboru realnych cisel !').

reseni(A,B,D) :-

    D = 0, !, X is -B/(2*A),
    write('Jedine reseni:'),
    write('x='), write(X).

reseni(A,B,D) :-

    Odm is sqrt(D),
    X1 is (-B + Odm)/(2*A),
    X2 is (-B - Odm)/(2*A),
    write('x1 = '), write(X1), nl,
    write('x2 = '), write(X2).
```

Komentář

- Program voláme predikátem *rovnice* a předáváme mu tři koeficienty kvadratické rovnice.
- Pokud koeficient kvadratického členu je roven nule, je rovnice lineární, uplatní se první část predikátu *rovnice* a ihned je vypočteno řešení. Řez zajistí, že druhá část predikátu *rovnice* už nebude zkoumána.
- Jestliže je koeficient kvadratického členu rovnice různý od nuly, lze vypočítat determinant Det.
- První dva koeficienty A,B a determinant Det se předávají predikátu *reseni*. Ten má tři části, opět oddělené pomocí řezů tak, aby se vždy vykonala jen jedna z nich.
  - Při záporném determinantu rovnice nemá řešení v oboru reálných čísel.
  - Jestliže se determinant rovná nule, existuje jedno řešení.
  - Kladný determinant znamená, že řešení jsou dvě.

Cvičení

Implementujte některý z dalších jednoduchých matematických algoritmů (např. nejmenší společný násobek, největší společný dělitel atd.).

Palindrom

Napište program, který bude o předkládaných slovech rozhodovat, zda jsou či nejsou palindromy. Program skončí po zadání slova "stop".
Poznámka: Palindrom je slovo, věta nebo verš, jejichž znění se při čtení od konce nemění (např. krk, pop, rotor, v elipse spi lev, kobyla ma maly bok, ...).

Řešení

```
palindrom :-

        write('Tento program testuje, zda '),
        write('zadane slovo je palindrom. '), nl,
        write('Zadavane slovo je nutno ukoncit teckou. '), nl,
        write('Po zadani slova "stop" program '),
        write('ukonci cinnost...'),nl,
        write('-----------------------------------------'),
        nl,nl,
        repeat,
        nl, write('Zadej slovo > '),
        read(X),
        ( X=stop; test(X),fail),
        !.

test(X):-

        write('Slovo '), write(X),
        name(X,L),
        otoc(L,L), !,
        write(' je palindrom'), nl.

test(X) :-

        write(' neni palindrom'), nl.

otoc([],[]).
otoc([A|X],Y) :- otoc(X,Z), spoj(Z,[A],Y).

spoj([],S,S).
spoj([A|X],Y,[A|Z]) :- spoj(X,Y,Z).
```

Komentář

- Program v cyklu (predikát repeat) načítá jednotlivá slova do proměnné X. Podle obsahu proměnné X dojde k rozvětvení:
  - když X = stop, Prolog přejde přes řez a skončí,
  - jinak (když X je různé od "stop") se načtené slovo otestuje, potom Prolog narazí na predikát fail a vrací se k načítání.
- Predikát *test* má dvě části, provést se ale má vždy jen jedna z nich, proto jsou odděleny řezem. První část predikátu je splněna, když zkoumané slovo je palindrom. Druhá část reaguje tehdy, když první se nepodaří splnit, tedy když slovo palindromem není.
- K rozpoznání palindromu je třeba převést načtený řetězec na seznam ASCII znaků (predikátem name), otočit jej (predikátem *otoc*) a zjistit, jestli se otočený seznam shoduje s původním.

Cvičení

1. Rozšiřte možnosti programu tak, aby bylo možné jeho činnost ukončit více možnostmi "posledního slova" (např. "stop" nebo "konec" nebo "dost" nebo ...).
2. Vylepšete stávající program tak, aby při zadávání celých vět (viz. příklady uvedené výše) ignoroval mezery mezi jednotlivými slovy (tj. namísto "kobylamamalybok" bude možné zadat "kobyla ma maly bok" a věta bude stejně vyhodnocena jako palindrom).

Synonyma a antonyma

Jsou dány seznamy navzájem synonymních slov a dvojice antonym:

```
synonyma([velky, veliky, rozlehly, rozsahly, rozmerny, obrovsky]).
synonyma([maly, malicky, nepatrny, drobny]).
synonyma([svetly, bily, jasny]).
synonyma([cerny, tmavy, temny]).
antonyma(cerny,bily).
antonyma(velky,maly).
```

Napište predikáty, kterými

- vypíšete všechna synonyma k zadanému slovu (zadané slovo ale znovu nevypisujte!)
- vypíšete všechna antonyma k zadanému slovu

Příklad: při zadání slova *rozlehlý* budou vypsána synonyma *velký, veliký, rozsáhlý, rozměrný, obrovský* a antonyma *malý, maličký, nepatrný, drobný*

Řešení

```
ant(A,B):-antonyma(A,B).
ant(A,B):-antonyma(B,A).

najdi-synonyma(X):-

    synonyma(S),clen(X,S),
    write('Nalezena synonyma ke slovu '),
    write(X),
    write(' jsou: '),
    vypis(X,S).

vypis(X,[]):-nl.
vypis(X,[X|S]):-vypis(X,S).
vypis(X,[A|S]):-X\=A,write(A),write(', '),vypis(X,S).

clen(X,[X|_]).
clen(X,[_|Zb]):-clen(X,Zb).

najdi-antonyma(X):-

    synonyma(S1),
    clen(X,S1),
    ant(Y,Z),
    clen(Y,S1),
    synonyma(S2),
    clen(Z,S2),
    write('Nalezena antonyma ke slovu '),
    write(X),
    write(' jsou: '),
    write(S2),nl.
```

Komentář

- Predikátem *ant* popíšeme skutečnost, že na dvojice antonym lze nahlížet z obou stran.
- Při hledání synonym (*najdi-synonyma*) stačí vyhledat v databázi ten seznam synonym, který obsahuje zadané slovo. Nalezený seznam S však nelze vypsat celý. Proto voláme predikát *vypis*, který se slovu X umí vyhnout.
- Predikát pro hledání antonym je trochu složitější. K danému slovu X najdeme synonyma S1. Potom hledáme dvojici antonym Y,Z, z nichž jedno (Y) je obsaženo v seznamu S1 a druhé (Z) patří do jiného seznamu S2. Seznam S2 je řešením.

Cvičení

Napište program, který nahradí zadaný seznam několika slov jejich synonymy / antonymy.

Jednoduchý kalkulátor

Napište program, který bude vyhodnocovat zadávané aritmetické výrazy.

Řešení

```
start :-

    write('Tento program vyhodnocuje'),
    write('zadane aritmeticke vyrazy.'), nl,
    write('Program ukoncite zadanim '),
    write('slova stop.'), nl,
    write('_____________________________'),
    write('_____________________________'), nl,
    write(' [platny aritmeticky vyraz '),
    write('musi byt ukoncen teckou]'), nl,
    repeat,
    nl, write('Zadej vyraz > '),
    read(X),
    pocitej(X),
    !, nl, nl.

pocitej(stop) :- write('Program byl ukoncen.').
pocitej(X) :- Vys is X, write(X = Vys), fail.
```

Komentář

- Program spustíme predikátem *start*. Po úvodních výpisech je požadováno zadání z klávesnice. Každý načtený vstup je předán predikátu *pocitej*.
- Pokud je načteno slovo "stop", Prolog úspěšně splní predikát *pocitej*, v predikátu *start* přejde přes řez a dvě odřádkování a skončí u tečky.
- Při zadání vstupu různého od "stop" má být vyhodnocen výraz. Do proměnné *Vys* se přiřazuje výsledek výpočtu. Výpočet se provádí zpracováním proměnné X, která obsahuje čísla, aritmetické znaky a závorky. Vypočtený výsledek je vypsán a predikát *pocitej* potom selže, takže je možné se vrátit k načítání dalšího výrazu.

Cvičení

Upravte stávající program tak, aby při zadání chybného výrazu, který nelze vyhodnotit, na tuto skutečnost uživatele upozornil a pokračoval v činnosti.

Hanojské věže

Hanojské věže jsou prastará hra, která se hraje s N disky a třemi tyčemi. Tyče jsou upevněny na podložce, každý disk má uprostřed otvor dovolující jeho nasunutí na tyč. Jednotlivé disky mají navzájem různý průměr. Ve výchozí situaci je všech N disků na pravé tyči tak, že jsou uspořádány dle velikosti - největší disk je dole. Cílem hry je přesunout všechny disky na prostřední tyč. Přitom je nezbytně nutné přemisťovat vždy pouze jediný disk a zároveň nesmí nikdy nastat situace, ve které by disk o větším průměru ležel na disku menším.

Řešení

```
start :-

    write('Zadej pocet disku > '),
    read(N),
    presun(N,leva,prostredka,prava).

presun(0,_,_,_) :- !.
presun(N,A,B,C) :-

    N1 is N-1,
    presun(N1,A,C,B),
    write('presun disk z '),
    write(A), write(' do '), write(B), nl,
    presun(N1,C,B,A).
```

Komentář

Zde vycházíme důsledně z metody rozkladu úlohy na podproblémy. Dle této metody je k vyřešení úlohy třeba přesunout N-1 disků z pravé tyče na tyč levou, přičemž lze používat jako pomocnou tyč prostřední, a poté přesunout poslední (největší) disk z pravé tyče rovnou na tyč prostřední. Nakonec ještě přesuneme N-1 disků, které jsou nyní vlevo na prostřední tyč.
Při optimálním postupu je třeba učinit 2N-1 tahů.

Cvičení

Zkuste zadanou úlohu vyřešit jiným způsobem (např. prohledáváním stavového prostoru).

Mastermind

Napište program, se kterým bude možné hrát hru "Mastermind".
Pravidla: Počítač zvolí pětici čísel z intervalu 1..9. Cílem hry je, aby protihráč co nejdříve tuto pětici uhodl. Nabídne vždy počítači pětici čísel a ten odpoví, kolik číslic je shodných s jeho pěticí a kolik z těchto číslic je i na správných pozicích. Hra končí ve chvíli, kdy hráč uhodne celou pětici včetně správného pořadí.
Začněte nejprve s jednodušší variantou hry, kdy je třeba uhodnout pouze správné číslice, nikoliv i správné pořadí.

Řešení

```
hra :-

        write(' Tato hra se jmenuje Mastermind.'), nl,
        write('-------------------------------------'),nl,
        write('Vasim ukolem je uhodnout petici cislic,'),
        write(' ktere si myslim.'), nl,
        write('V uvahu pripadaji cislice 1 az 9. '), nl,
        write('Hadate pomoci seznamu delky pet '),
        write('(napr. [1,3,3,4,6]).'), nl,
        write('Vzdat se muzete kdykoliv odpovedi []. '), nl,
        write('-------------------------------------'), nl,
        write('Muzeme zacit hrat ? (a/n):'),
        getb(Ch), Ch = 97, % Ch = a
        nl,
        Petice = [A,B,C,D,E],
        vytvor(Petice),
        repeat,
        write(' > '), read(X),
        vyhodnoceni_odpovedi(Petice,X), !.

vyhodnoceni_odpovedi(Petice,[]) :-

        !,
        write('Vzdavate se prilis snadno ... '), nl,
        write('Spravna odpoved':Petice), nl.

vyhodnoceni_odpovedi(Petice,X) :-

        X = [A,B,C,D,E], !,
        porovnej(Petice,X).

vyhodnoceni_odpovedi(Petice,_) :-

        write('Nekorektni odpoved !!!'), nl,
        !, fail.

porovnej([A|S],Odp) :-

        smaz(A,Odp,L), porovnej(S,L).

porovnej([],[]) :-
```

```
        !, nl,
        write(' Vyborne ! Prave jste vyhral !!! '), nl.

porovnej([],Odp) :-

        length(Odp,X), Y is 5 - X,
        write(Y), write(' krat spravna odpoved.'), nl,
        !, fail.

smaz(X,[],[]).
smaz(X,[X|T],T) :- !.
smaz(X,[Y|T],[Y|U]) :- smaz(X,T,U).

vytvor([]).
vytvor([X|T]) :- vytvor(T), X is int(rand(9)) + 1.
```

Komentář

- Predikát *hra* spouštíme hru. Po výpisu pravidel čeká Prolog na stisknutí klávesy "a". Je-li stisknuta jiná klávesa, predikát *hra* není splněn a program končí. Pokud hráč chce hrát, je třeba vygenerovat pětici čísel predikátem *vytvor*. Následuje cyklus, v němž se načítají a vyhodnocují odpovědi hráče.
- *K vyhodnoceni_odpovedi* je třeba předat načtenou odpověď uživatele a také Pětici čísel, s níž se hraje. Vyhodnocení může mít tři různé výsledky: buď hráč vzdal, nebo zadal odpověď ve správném formátu, nebo udělal chybu. V prvním případě hra končí, ve třetím selhává predikát *vyhodnoceni_odpovedi*. Je-li jako odpověď načtena pětice čísel, může dojít k jejímu porovnání s původní pětící.
- Predikát *porovnej* se snaží odstraňovat dvojice stejných čísel z původní a z hádané pětice. Pokud výsledkem porovnání jsou dva prázdné seznamy, znamená to, že hráč vyhrál (uhodl všechna čísla). Když z Pětice ještě něco zůstane, dozví se hráč, kolikrát se trefil a predikát *porovnej* selže, tím selže i *vyhodnoceni_odpovedi* a lze načíst další hráčův tip.

Cvičení

1. Upravte program tak, aby bylo možné hrát "Mastermind" dle výše uvedených pravidel (tj. hádá se i správné pořadí číslic).
2. Napište program, který umožní počítači hrát tuto hru v opačné roli (tj. počítač hádá číslice).

Hra se zápalkami

Na stole leží 21 zápalek. Dva hráči, kteří se pravidelně střídají, mohou v jednom tahu odebrat nejméně jednu a nejvýše čtyři zápalky. Hra končí ve chvíli, kdy jsou všechny zápalky rozebrány a hráč, který vzal poslední zápalku prohrává.

Řešení

```
start :- hrac(21).
% zacina hrat uzivatel a ve hre je 21 zapalek

hrac(1) :-

    !, write('Prohral jsi, beres posledni !'), nl.

hrac(Poc) :-

    write('Pocet zapalek na stole:'),
    write(Poc), nl,
    cti(X), nl,
    Poc1 is Poc-X,
    pocitac(Poc1).

pocitac(1) :-

    !, write('Prohral jsem, beru posledni !').

pocitac(Poc) :-

    write('Pocet zapalek na stole:'),
    write(Poc), nl,
    X is (Poc-1) mod 5,
    write('Ja beru:'), write(X), nl, nl,
    Poc1 is Poc - X,
    hrac(Poc1).

cti(X):-

    repeat,
    write('Kolik beres zapalek ? '),
    read(X),
    integer(X),
    X < 5, X > 0, !.
```

Komentář

- Hru spustíme predikátem *start*. Následně se střídavě volají predikáty *hrac* a *pocitac*, podle toho, kdo je právě na řadě. Predikátům se jako argument předává počet zápalek, zbývajících na stole.
- Predikát hráč má dvě části. Jestliže hráči zbývá jen jedna zápalka, prohrál. Má-li hráč více než jednu zápalku, znamená to, že si může vybrat, kolik zápalek odebere. Spustí se tedy predikát *cti.*
- Predikát *cti* pracuje v cyklu a čeká na zadání čísla z intervalu 1 až 4.
- Také tah počítače je rozdělen do dvou částí. Zbude-li počítači jedna zápalka, prohrál. (V této variantě programu ale taková situace nemůže nastat.) Když je

zápalek více, lze provést tah. V takovém případě je počet odebíraných zápalek vypočítán tak, aby počítač mohl vyhrát. Prozkoumejte, jak tato strategie funguje!

Cvičení

1. Skutečnost, že neustále vyhrává počítač je jistě nudná (navíc příslušná strategie je zcela průhledná). Upravte program tak, aby počet zápalek ve hře byl volitelný z rozumného intervalu.
2. Obohaťte program o možnost volby "Začíná počítač" / "Začíná uživatel".
3. Zamlžte vyhrávající strategii podle níž hraje počítač (např. nechť se počítač pokouší hrát podle této strategie až když počet zápalek ve hře klesne pod určitou hodnotu a do této doby nechť odebírá náhodně zvolený počet zápalek).

Sněhová kalamita

Představte si, že jste pracovníkem zodpovědným za sjízdnost silnic v dané oblasti (silniční síť je popsána predikátem sousedi(Místo1, Místo2)). Celou oblast náhle postihla sněhová kalamita, která zapříčinila neprůjezdnost některých obcí (jsou zadány predikátem neprujezdne(Seznam_obci)), a značně zkomplikovala celkovou dopravní situaci. Vaším cílem je napsat program, který usnadní návrh nouzových tras mezi místy, která nejsou ve výše uvedeném seznamu (např. zodpovědět dotaz, zda je možné se dostat z Orlice do Vysoké a případně jak?).

Řešení

```
sousedi(jamne,hrusova).
sousedi(jamne,knirov).
sousedi(jamne,orlice).
sousedi(kvetna,knirov).
sousedi(lhota,jablunkov).
sousedi(lhota,zleb).
sousedi(knirov,vysoka).
sousedi(knirov,lhota).
sousedi(knirov,dlouha).
sousedi(dlouha,kvetna).
sousedi(dlouha,jablunkov).
sousedi(hrusova,jablunkov).
sousedi(hrusova,orlice).
sousedi(javornik,jablunkov).
sousedi(javornik,kvetna).
sousedi(javornik,zleb).

neprujezdne([jamne,lhota,kvetna]).

navrhni_cestu(Odsud,Cil) :-

        neprujezdne(Neprujezdne),
        not(clen(Odsud,Neprujezdne)),
        not(clen(Cil,Neprujezdne)),
        cesta(Odsud,Cil,[Cil],Neprujezdne),
        fail.
        navrhni_cestu(_,_) :-
        write('Zadna dalsi cesta ! '), nl.

cesta(Odsud,Odsud,Navrh,_) :-

        write(Navrh), nl.

cesta(Odsud,Nyni,Dosud,Neprujezdne) :-

        silnice(Nyni,Dalsi),
        not(clen(Dalsi,Dosud)),
        not(clen(Dalsi,Neprujezdne)),
        cesta(Odsud,Dalsi,[Dalsi|Dosud],Neprujezdne).

silnice(X,Y) :- sousedi(X,Y).
silnice(X,Y) :- sousedi(Y,X).
```

```
clen(X,[X|_]).
clen(X,[_|H]) :- clen(X,H).
```

Komentář

- Program spustíme predikátem *navrhni_cestu,* kterému předáme počátek a cíl cesty. Predikát ověří, že ani jeden z těchto bodů se nevyskytuje v seznamu neprůjezdných obcí. Potom je spuštěno vyhledávání cesty predikátem *cesta*.
- Predikát *cesta* využívá algoritmus prohledávání stavového prostoru do hloubky. Obec, o kterou má být stávající cesta prodloužena, nesmí být obsažena v seznamu neprůjezdných obcí (Neprujezdne), ani se nesmí vyskytovat v dosavadní cestě (Dosud). Pokud se první bod cesty (Odsud) shoduje s poslední položkou, přidanou do seznamu, je vypsáno řešení.
- Po splnění cíle *cesta* narazí predikát *navrhni_cestu* na fail, vrátí se a hledá další řešení.

Cvičení

Modifikujte program tak, že neprůjezdné nebudou celé obce, ale pouze konkrétní silnice

Nejkratší cesta

Prostřednictvím predikátu c(Odkud,Kam,Km) je zadána silniční síť v určité oblasti včetně kilometráže. Vaším úkolem je napsat program, který bude vyhledávat nejkratší možné spojení mezi dvěma zadanými uzly ve smyslu co nejmenšího počtu najetých kilometrů - nikoliv nejmenšího počtu použitých silnic (hran).

Řešení

```
c(a,b,68).
c(a,c,75).
c(a,e,45).
c(b,d,43).
c(c,d,31).
c(c,e,28).
c(c,f,63).
c(d,f,28).
c(e,g,115).
c(f,g,16).

najdi_min_cestu(Zacatek,Konec) :-

    hledej(Konec,cesta(0,[Zacatek]),[]).

hledej(Konec,cesta(Cena,Cesta),_) :-

    Cesta = [Konec|_],
    write(Cesta), write(' : '), write(Cena), nl.

hledej(Konec,cesta(Cena,Cesta),Ostatni_cesty) :-

    findall(R,najdi_pokracovani(cesta(Cena,Cesta),R),Seznam),
    spoj(Ostatni_cesty,Seznam,NovySeznam),
    najdi_min(NovySeznam, Min, Ostatni),
    hledej(Konec,Min,Ostatni).

najdi_pokracovani(cesta(Cena,Cesta),cesta(Y,[H2|Cesta])) :-

    Cesta = [H1|_],
    silnice(H1,H2,C1),
    nevznikne_cyklus(H2,Cesta),
    Y is C1 + Cena.

najdi_min([A|X],Min,Zbytek) :- minimum(X,A,[],Min,Zbytek).

minimum([],Min,Zbytek,Min,Zbytek).
minimum([A|L],B,X,Min,Zbytek) :-

    A = cesta(C1,_), B = cesta(C2,_),
    C1 > C2,
    !,
    minimum(L,B,[A|X],Min,Zbytek).

minimum([A|L],B,X,Min,Zbytek) :-
```

```
    minimum(L,A,[B|X],Min,Zbytek).

spoj([],L,L).
spoj([A|L1],L2,[A|K]) :- spoj(L1,L2,K).

nevznikne_cyklus(X,[]).
nevznikne_cyklus(X,[Y|T]) :-

    X \= Y, nevznikne_cyklus(X,T).

silnice(A,B,X) :- c(A,B,X).
silnice(A,B,X) :- c(B,A,X).
```

Komentář

- Predikátu *najdi_min_cestu* zadáme počátek a konec cesty. Cesta se ukládá do struktury s aritou 2: prvním argumentem je délky cesty, druhým seznam uzlů.
- Predikát *hledej* má dvě části. První vypisuje řešení, druhá je hledá. Hledání je založeno na algoritmu prohledávání stavového prostoru do šířky. V každém volání predikátu jsou vyhledána všechna pokračování stávající cesty a uložena do seznamu. Ze seznamu možných pokračování se vybere to nejvýhodnější, které se pak dále rozvíjí.
- Predikát *najdi_pokracovani* prodlužuje cestu o jeden nový uzel H2. Přitom kontroluje, zda H2 již není součástí cesty (*nevznikne_cyklus*). Cena cesty se zvyšuje o Y, což je délka použité silnice.
- Predikát najdi_min hledá v seznamu (první argument) jeho minimum (Min) a zbytek seznamu odkládá do proměnné Zbytek. K vlastnímu rozkladu seznamu na minimum a ostatní prvky slouží predikát *minumum*.

Cvičení

1. Upravte stávající program tak, aby hledal naopak co nejdelší cestu mezi zadanými místy.
2. Napište program, který bude hledat nejkratší cestu mezi dvěma místy ve smyslu nejmenšího počtu použitých cest (hran).

Koza, vlk, zelí

Převozník, koza, vlk a zelí jsou na pravém břehu řeky. Musí se přepravit na břeh levý, přičemž mají k dispozici malou loďku, ve které může převozník vézt kromě sebe nejvýše jeden další objekt. Pokud převozník zanechá na některém břehu bez dozoru kozu s vlkem, vlk ji sežere. Analogická situace nastane v případě, že bez dozoru zůstane koza a zelí. Je možné přepravit celý převozníkův majetek bezpečně na opačný břeh ? (Poznámka: Je zřejmé, že loďka nesmí jet nikdy bez převozníka.)

Řešení

```
% stav(Prevoznik,Koza,Vlk,Zeli)
% - lodka musi byt tam, kde je prevoznik
% pocatecni stav
pocatecni_stav(stav(leva,leva,leva,leva)). % vsichni vlevo

% koncovy stav
koncovy_stav(stav(prava,prava,prava,prava)). % vsichni vpravo

% MOZNE ZMENY STAVU
% prevoznik jede sam
zmena(stav(P,K,V,Z),stav(P1,K,V,Z)) :- opacny_breh(P,P1).
% prevoznik a koza
zmena(stav(P,P,V,Z),stav(P1,P1,V,Z)) :- opacny_breh(P,P1).
% prevoznik a vlk
zmena(stav(P,K,P,Z),stav(P1,K,P1,Z)) :- opacny_breh(P,P1).
% prevoznik a zeli
zmena(stav(P,K,V,P),stav(P1,K,V,P1)) :- opacny_breh(P,P1).

opacny_breh(leva,prava).
opacny_breh(prava,leva).

najdi_reseni :-

    pocatecni_stav(Pocatecni_stav),
    koncovy_stav(Koncovy_stav),
    hledej(Pocatecni_stav, [Koncovy_stav], Reseni),
    vypis_reseni(Reseni), nl,
    fail.

najdi_reseni.

% hledej(Poc_stav, Potencialni_reseni, Reseni)
hledej(X,[X|T],Plan) :- Plan = [X|T]. % mam reseni
hledej(X,[S|T],Plan) :-

    zmena(S,S1),
    bezpecny(S1),
    nevznika_cyklus(S1,[S|T]),
    hledej(X,[S1,S|T],Plan).

% stav je bezpecny, pokud prevoznik a koza jsou spolu
bezpecny(stav(X,X,_,_)) :- !.
% stav je bezpecny, pokud prevoznik, vlk a zeli jsou spolu
```

```
bezpecny(stav(X,_,X,X)).

% zabranuje zacykleni programu
nevznika_cyklus(X,[]).
nevznika_cyklus(X,[Y|Z]) :- X \= Y, nevznika_cyklus(X,Z).

vypis_reseni([]).
vypis_reseni([H|T]) :- write(H), nl, vypis_reseni(T).
```

Komentář

- V programu jsou nejprve nadefinována fakta o úloze: počáteční a koncový stav problému a přípustné změny stavů, odpovídající pohybům převozníka a jeho majetku.
- Predikát *najdi_reseni*, kterým program spustíme, vyhledá v databázi počáteční_stav a koncový_stav problému a naplní jimi stejnojmenné proměnné. Následně začne hledat vyhovující posloupnosti změn, jimiž lze změnit Počáteční_stav na Koncový_stav. Nalezenou posloupnost uloží do seznamu zvaného Reseni. Zjištěné řešení je vypsáno. Poté Prolog narazí na nesplnitelný predikát fail, vrátí se a hledá jiná řešení.
- V predikátu *hledej* se uplatňuje algoritmus prohledávání stavového prostoru do hloubky. Začíná se od Koncového_stavu, před nějž se postupně připojují přípustné předcházející stavy. Pokud se připojený stav shoduje s požadovaným počátečním stavem, hledání končí (viz. první řádek predikátu). V druhém části predikátu je vidět připojování stavů: před stav S lze připojit stav S1, pokud
  - lze změnit S na S1
  - stav S1 je bezpečný, čili není při něm ohroženo ani zelí, ani koza
  - nevznikne cyklus, tedy stav S1 ještě nebyl použit v sestavovaném řešeni

  Při splnění všech těchto podmínek je stav S1 připojen před S a znovu se spustí predikát *hledej*.
- Definice *bezpecnych* stavů je rozdělena do dvou částí, oddělených řezem.
- Predikát *nevznika_cyklus* je splněn, pokud X není prvkem seznamu, který byl předán predikátu jako druhý argument.
- Predikát *vypis_reseni* vypisuje na jednotlivé řádky stavy, které jsou součástí řešení.

Cvičení

1. Napište analogický program, který bude řešit následující problém: Na pravém břehu řeky jsou tři misionáři a tři kanibalové. Všichni se potřebují dostat na opačný břeh, přičemž mají k dispozici pouze malou loďku, do které se vejdou nejvýše dva pasažéři. V případě, že na některém břehu bude v libovolném okamžiku dosažen takový stav, že kanibalů bude více než misionářů, budou misionáři bez otálení snědeni.
2. Zobecněte program tak, aby bylo možné řešit problém N misionářů a N kanibalů.

Vzorové zadání zkouškové písemky

Písemná práce obsahuje čtyři příklady. Toto je vzorové zadání:

1. Napište program zdvoj(Sez, Sez2), který každý prvek seznamu Sez vloží do Sez2 dvakrát (Sez=[1,2,5] -> Sez2=[1,1,2,2,5,5]).
2. Napište jakýkoliv predikát pro setřídění seznamu čísel vzestupně.
3. Napište program, který rozhodne, zda zadané číslo je prvočíslo.
4. Pyramida z kostek má v nejvyšší vrstvě 1 kostku, v další 9 (3x3), potom 25 (5x5) kostek atd. Napište predikát spocti(N,Kolik), který vypočte celkovou spotřebu kostek pro pyramidu výšky N.

Na písemku máte 40 minut, následuje ústní část zkoušky, při které máte možnost vysvětlit řešení a postupy, které jste použili v písemce.

Hodnocení:

- všechny čtyři příklady správně - výborně
- tři příklady správně - velmi dobře
- dvě příklad správně - dobře

ČKD čili často kladené dotazy

1. Co znamená chybové hlášení “predicate_protected“?

Pravděpodobně jste pojmenovali svůj predikát názvem, který je rezervovaný pro vestavěné predikáty dané implementace Prologu: read, write, repeat atd. Přejmenujte svůj predikát a program znovu přeložte.

2. Co znamená chybové hlášení “predicate_not_defined“?

Prolog nenalezl predikát požadovaného názvu. Zkontrolujte, že definice predikátu je v programu obsažena, že jste program zkompilovali a že jste v dotazu neudělali překlep.

3. Proč se program s rekurzí cyklí…?

Programy, obsahující rekurzivní volání, se mohou zacyklit velmi snadno. Příčinou je většinou to, že zapomenete správně definovat zarážku pro rekurzi anebo ji omylem zapíšete až za řádek s rekurzivním voláním.

Správně definovaný predikát pracující v rekurzi vypadá takto:

```
vypis(0).
vypis(N):-N>0,write(N),nl,N1 is N-1,vypis(N1).
```

Tento predikát vypisuje na zvláštní řádky sestupně čísla od N do 1. V první řádku je definována zarážka, která zastaví výpočet, pokud argument bude mít hodnotu nula. Pokud tento řádek nevyhovuje, Prolog zkouší řádek druhý – vypíše číslo a v rekurzi volá stejný predikát znovu s o 1 nižším argumentem.

Zacyklená, čili chybná verze predikátu by byla tato:

```
vypis(N):-write(N),nl,N1 is N-1,vypis(N1).
vypis(0).
```

Zde Prolog vypíše číslo, sníží hodnotu argumentu a provede rekurzivní volání. Protože v predikátu chybí podmínka, že N>0, může Prolog zpracovávat první řádek predikátu donekonečna, k druhému řádku nikdy nedojde.

4. Proč program jmena(A):-kral(A,B,C),fail. nevypisuje žádné řešení, ačkoliv databáze je správně definována?

Fail na konci predikátu není nikdy splněný. Prolog se přes fail nedostane k tečce a nemůže tedy vracet hodnotu v proměnné A. Ve spojení s fail je vždy nutné řešení vypisovat pomocí příkazu write. Správně tedy definujeme:

```
jmena(A):-kral(A,B,C),write(A),fail.
```

5. Výsledkem generování sestupně uspořádaného seznamu pěti čísel [5,4,3,2,1|_34789]. Kde je chyba?

Pravděpodobně byl použit predikát v této podobě:

```
seznam(0,A).
seznam(N,[N|A]):-N>0,N1 is N-1,seznam(N1,A).
```

Zarážka výpočtu ale není dobře nastavená. Správně má být

```
seznam(0,[]).
seznam(N,[N|A]):-N>0,N1 is N-1,seznam(N1,A).
```

Pamatujte, že řádky programu musí vyjadřovat logicky pravdivá tvrzení. Logicky pravdivé je to, že seznam délky 0 je prázdný seznam [], nikoli libovolný seznam A!

6. Co znamená chybové hlášení _58932 is _88201 + 3?

Prologu se nepodařilo přiřadit hodnoty některým proměnným a nemůže proto provést výpočet. Zkontrolujte, že všechny proměnné jsou specifikované dříve, než se objeví ve výpočtech.

7. Proč musíme k predikátu, zakončeným příkazem fail, připojovat ještě prázdný řádek?

Prázdný řádek zajistí celkové kladné ukončení vyhodnocování predikátu.

Napíšeme-li:

```
nas-predikat:-nejaka-akce(A),write(A),nl,,fail.
```

Prolog vyhledá všechny přípustné hodnoty A, ale potom kvůli neúspěšnému fail skončí i celý *nas-predikat* neúspěšným hlášením "no". Chceme-li *nas-predikat* uplatnit v delším programu, musíme definovat.

```
nas-predikat:-nejaka-akce(A),write(A),nl,,fail.
nas-predikat.
```

Po výpisu všech možných A přejde Prolog přes prázdný řádek a *nas-predikat* bude považovat za úspěšně zpracovaný.

8. Musíme v dotazech používat stejné názvy proměnných, jaké byly použity v definicích predikátů?

Ne. Prolog dokáže navázat proměnné zadané v dotazu na proměnné v predikátech, takže můžeme používat libovolné značení. Ostatně uživatel programu nemusí vůbec znát názvy jednotlivých proměnných v kódu!

Při psaní programu je vhodné nešetřit námahou a volit co nejvýstižnější značení proměnných, třeba i složené z několika slov. Později se v programu snadněji zorientujete. Oč přehlednější je

```
fibonnaci(Pocitadlo,Zarazka,Clen1,Clen2,Vysledek)
```

než

```
fibonnaci(A,B,C,D,E)
```

9. Existují v Prologu globální proměnné?

Ne! Každý predikát pracuje se svými vlastními proměnnými. Při volání predikátu:

- buď předáme konkrétní hodnoty, které Prolog přiřadí odpovídajícím proměnným,
- nebo předáme názvy proměnných, jejichž prostřednictvím chceme získat výstupy predikátů.

S proměnnou, použitou v dřívějších výpočtech, kterou dalšímu predikátu nepředáme, nemůže Prolog pracovat.

Uvažujme o programu, který by uměl převzít seznam jako vstup, načíst číslo N, smazat N-tý prvek ze seznamu a vrátit zbytek seznamu jako výstup. Toto je správná verze programu:

```
uprava(A,B):-read(N),smazani(N,A,B)
```

A je na počátku specifikované, B nespecifikované. Načtením se do N přiřadí hodnota. Smazani přebírá hodnoty proměnných N, A a proměnnou B, kterou naplní výsledkem.

Varianta

uprava(A,B):-read(N),smazani(A,B)

by nefungovala, protože mazací predikát by neobdržel index prvku, který má být odstraněn.

10. Jak poznat, má-li být hlavička seznamu vložena do výsledku na levé, nebo na pravé straně pravidla?

Toto je správně napsaný program, který vyměňuje v binárním seznamu jedničky za nuly a opačně:

```
change([],[]).
change([0|T1],[1|T2]):-change(T1,T2).
change([1|T1],[0|T2]):-change(T1,T2).
```

Chybou by bylo napsat:

```
change([],[]).
change([0|T1],T2):-change(T1,[1|T2]).
change([1|T1],T2),:-change(T1,[0|T2]).
```

Je dobré si vždy uvědomit, jestli zarážka rekurze vypadá logicky nebo ne. Vyměňujeme-li prvky v prázdném seznamu, výstupem bude opět prázdný seznam. Kdybychom ale hlavičky připojovali na pravé straně pravidla, k řádku, kde jsou oběma argumenty prázdné seznamy, bychom se nikdy nedostal, protože zatímco první seznam by se v průběhu rekurze zpracoval, druhý by se prodlužoval.

11. Jak probíhá zkouška z předmětu LP1?

Zkouška je písemná a ústní. Písemka se skládá ze čtyř úloh, které musíte vyřešit během 60 minut písemně na papír, bez použití jakýchkoli podkladů a bez počítače. Při ústní části zkoušky máte možnost vysvětlit svá řešení, případně okomentovat chyby, kterých jste se dopustili. Hodnocení je jednoduché – za čtyři vyřešené úlohy dostanete jedničku, za tři dvojku, za dvě úlohy trojku.

Glossary

Search

Search Results:

A	
abolish(H)	Vestavěný predikát pro smazání těch klauzulí, která se dají unifikovat s hlavou H. V některých implementacích (např. Arity Prolog) se používá predikát retractall(H).
anonymní proměnná	Proměnná, jejíž hodnota nás nezajímá. Značí se podtržítkem.
arg(N,S,X)	Vestavěný predikát, který vybere N-tý argument ze struktury a naváže jen na proměnnou X.
argumenty	Souhrnný název pro jména objektů a proměnných v relaci.
arita struktury	Počet argumentů ve struktuře.
assert(Klauzule)	Vestavěný predikát pro přidávání do databáze.
asserta(Klauzule)	Vestavěný predikát pro přidávání na začátek databáze.
assertz(Klauzule)	Vestavěný predikát pro přidávání na konec databáze.
atom	Znaková konstanta.
atom(X)	Predikát, který je splněn, pokud X je atom.
atomic(X)	Predikát, který je splněn, pokud X je term.

B	
backtracking	Zpětné procházení. Vestavěný mechanismus, který zajišťuje, že po úplném skončení vyhodnocování podcíle se Prolog vrátí na předcházející podcíl a zkusí jej splnit s jinými vstupními údaji.
bagof(T,C,S)	Vestavěný predikát, který vytvoří seznam S ze všech termů T, pro něž je splněn cíl C.

C	
call(C)	Vestavěný predikát, který je splněn tehdy, když je splněn cíl C.
consult(Soubor)	Vestavěný predikát pro načítání klauzulí ze souboru.

D	
dotazovací režim	Režim, v němž lze klást dotazy.

F	
fail	Vestavěný, nikdy nesplněný predikát.
findall(T,C,S)	Vestavěný predikát, který vytvoří seznam S ze všech termů T, pro něž je splněn cíl C.
functor(S,J,A)	Vestavěný predikát, který vrací jméno a aritu struktury, nebo vytvoří a strukturu daného jména a arity.
funktor	Název struktury spolu s aritou struktury.

G	
get(X)	Vestavěný predikát pro načtení znaku z klávesnice.

H	
hlava pravidla	Levá část pravidla před znaky :-.
hlava seznamu	První prvek seznamu.

I	
integer(X)	Predikát, který je splněn, pokud X je celé číslo.
is	Operátor pro vyčíslení.

K	
klauzule	Fakt nebo pravidlo.
konstanta	Pojmenování konkrétního objektu nebo relace. Číslo, posloupnost znaků a čísel, nebo řetězec v uvozovkách.
konzultační režim	Režim, v němž lze zadávat fakta a pravidla.

L	
listing	Vestavěný predikát pro výpis databáze.
listing(P)	Vestavěný predikát pro výpis všech klauzulí P z databáze.
listing(P/A)	Vestavěný predikát pro výpis klauzulí P dané arity A z databáze.

N	
name(T,S)	Vestavěný predikát. Převádí term T na seznam S znaků ASCII, nebo opačně.
nl	Příkaz pro odřádkování.
not	Označení negace.

O	
operátor	Znak relace nebo operace.

P	
pravidlo	Vztah mezi skutečností a seznamem podcílů, které musí být splněny, aby skutečnost byla pravdivá.
predikát	Popis vztahu mezi objekty.
proměnná	Posloupnost znaků a čísel, začínající velkým písmenem nebo podtržítkem.
put(X)	Vestavěný predikát, který vypíše ASCII znak odpovídající číslu X.

R	
read(X)	Vestavěný predikát pro načtení vstupu do proměnné X.
reconsult(Soubor)	Vestavěný predikát pro načtení klauzulí ze souboru.
rekurze	Způsob definování objektu pomocí sebe sama.
relace	Jméno, popisující způsob, jakým je soubor objektů spolu vázán.
repeat	Vestavěný, vždy splněný predikát cyklu.
retract(C)	Vestavěný predikát po smazání klauzule, kterou lze unifikovat s C.
retractall(H)	Vestavěný predikát pro smazání všech klauzulí, které lze unifikovat s H. V některých implementacích se používá predikát abolish(H).
řez	Vestavný predikát, označený vykřičníkem. Program se před něj nemůže vracet. V některých případech řez zajišťuje správné fungování programu (červený řez), jindy řez pouze program zefektivní (zelený řez).

S	
setof(T,C,S)	Vestavěný predikát, který vytvoří seznam S ze všech termů T, pro něž je splněn cíl C. Každý term se v seznamů může vyskytovat nejvýše jednou.
složená struktura	Struktura, jejímž některým termem je opět jiná struktura.
struktura	Název spolu s libovolným počtem termů.

struktura=..Seznam	Konverze struktury na seznam, nebo naopak.

T	
tab(N)	Vestavěný predikát pro výpis N mezer.
tělo pravidla	Pravá část pravidla za znaky :-.
tělo seznamu	Všechny prvky seznamu s výjimkou prvního.
term	Libovolný objekt jazyka Prolog.

U	
unifikace	Ztotožnění. Proces, pomocí kterého se Prolog pokouší vyhovět podcíli vzhledem ke skutečnostem a levé straně pravidla tím, že buď splní tento podcíl, nebo určí jeden nebo více dalších podcílů, nutných pro vyhodnocení původního podcíle.

V	
vázaná proměnná	Proměnná, které byla přiřazena konkrétní hodnota.
volná proměnná	Proměnná, ke které se zatím neváže konkrétní hodnota.

W	
write	Vestavěný predikát pro výpis textu na výstupní zařízení.

Others	
,	Logická konjunkce (AND).
;	Logická disjunkce (OR).
=	Porovnání.
==	Operátor porovnání bez přiřazení.
\=	Opak operátoru =.
\==	Opak porovnání bez přiřazení.