Programování v Prologu

Co je to Prolog?

Programovací jazyk **Prolog** vznikl na začátku 70. let ve Francii. Alain Colmerauer a Philipp Roussel jej začali vyvíjet za účelem umožnění komunikace člověka s počítačem v přirozeném jazyce.

Prolog pracuje na bázi *predikátové logiky prvního řádu*. Jednoduše řečeno, je to odvozování poznatků na základě známých faktů a pravidel neboli *vyplývání*.

Imperativní a deklarativní programování

Moderní vysokoúrovňové jazyky (např. Java) jsou *imperativní*. Programy popisují algoritmus, kterým se řeší specifický (konkrétní) problém, a využívá se k tomu většinou *procedurální programování* a objekty.

Prolog je zkratka francouzského "**Pro**grammation en **log**ique" nebo anglického "**Pro**gramming in **log**ic". Ta značí, že se jedná o logické programování. Definujeme pouze, ***co*** se má vyhodnotit, a neřešíme způsob ***jak***, což Prolog řadí do kategorie *deklarativních programovacích jazyků*.

Deklarativní (logické) programování

Vnitřní algoritmy vyhodnocování dotazů (cílů) jsou pro nás skryté. Tím se jazyk stává přívětivější i pro laiky, neboť není třeba znát (naučit se) architekturu a princip fungování počítače.

Pro snadnou orientaci v kódu je důležité ovšem znát (naučit se) základy ***predikátové logiky***. Programátor vytváří posloupnost tzv. *predikátů*, které dále dohromady v daném sledu a kontextu vytvářejí tzv. Prologovskou databázi (nespojovat s klasickou databází !) pro reprezentaci znalostí.

Jazyk *Prolog* je interpretovaný, čili nemáme předem určený postup řešení, ale zadáváme dotazy, které chceme vyhodnotit. K tomu interpretátor využívá ***rekurzi, unifikaci a backtracking***.

Prolog byl využit v roce 2005 ve vesmírné stanici agentury ***NASA***. V systémech umělé inteligence našel využití v nedávné době i při programování superpočítače ***Watson*** od IBM.

Stejně důležité jsou návrh znalostních bází, chytré vyhledávání ap.

Prolog přináší jiný pohled na programování a přechod z klasického přístupu může být pro některé programátory výzva.

V Prologu se dají napsat rychle *jednoduché a efektivní* aplikace, které jsou kódově *transparentní* a tedy *snadno udržitelné*, což je většinou naším cílem.

V kombinaci s *imperativním jazykem*, jako je *Java* nebo *C*, má Prolog ještě mnohem větší potenciál. Nejprve si řekneme, jak s programováním v Prologu vlastně začít.

Vývojové prostředí

Jazyk Prolog je jazykem interpretovaným, proto pro volání programů potřebujeme vhodný *interpretátor*.

První intepretátor naprogramoval P. Roussel společně s A. Colmerauerem v roce 1972. Postupně se rozšiřovaly další implementace a každá v sobě obsahovala trochu jinou sadu zabudovaných *predikátů*.

Kvůli přenositelnosti Prologovských programů bylo nutné zavést pevný *standard*. Návrh standardu sepsal v roce 1984 R. O'Keefe, přesto ho málokterá verze Prologu dodržovala.

V průběhu let se standard upřesňoval a v roce 1993 vyšla další neoficiální sumarizace návrhu. Až v roce 1995 byl standard *oficiálně zaveden*, ale až v posledních několika letech je implementacemi více dodržován.

Rozhraní každého interpretátoru je *odlišné*, podporuje jiné zápisy a stejné predikáty mohou v různých verzích fungovat trochu jinak. Nás budou zajímat pouze implementace podporující *ISO Prolog*. Záleží také na platformě, na které budeme pracovat. V praxi budeme chtít Prolog využít k efektivnímu řešení komplexnějších problémů a různé *implementace* se hodí na *odlišné typy úloh*.

Přehled interpretátorů s volnou licencí

tuProlog (GNU LGPL) – skládá se z minimalistického jádra, které zvládá základní práci s Prologem, a lze ho rozšířit dalšími sadami predikátů.

Jazyk byl původně vytvořen pro webové applety. Výsledné programy kompiluje do .jar formátu, takže jsou spustitelné napříč platformami s JVM. Existuje také verze pro Android a .NET. S těmito jazyky je potom kompatibilní.

Visual Prolog – podporuje pouze platformu Windows. Kombinuje logické, klasické i objektové programování. Je kompatibilní s jazyky C a C++ a může používat funkce Win32.

Prostředí podporuje i grafické vytváření a úpravy formulářů a oken. Pro soukromé účely je Visual Prolog zdarma, komerčně se může využívat až se zaplacenou licencí.

Ciao Prolog – v základu plně podporuje ISO standard, ale mohou se přidat pro každý projekt rozšiřující moduly. Poslední verze je ze srpna roku 2011 a funguje na všech nejpoužívanějších platformách.

Ciao Prolog se může propojit s jazyky C i Java anebo s relačními databázemi. Existují i moduly pro síťovou komunikaci a webové aplikace.

C#Prolog (GNU GPLv2) – stále aktualizovaná verze, podporuje jednoduché propojení s jazykem C# a tedy i databázemi, se kterými tento jazyk dokáže pracovat.

Má podporu pro formát XML a stále se rozšiřující struktury JSON. Funguje na platformě Windows i Linux.

GNU Prolog (GNU LGPL) – kromě standardu ISO obsahuje více než 300 dalších zabudovaných predikátů. Podporuje ladění a propojení s jazykem C.

Obsahuje optimalizovaný a rychlý kompilátor, který vytváří jednoduché spouštěcí soubory. Pracuje na všech základních platformách.

SWI Prolog (GNU GPL) – Standard ISO Prolog je rozšířen o robustní grafické rozhraní s možností ladění programů a přidání rozšiřujících balíčků.

Můžeme jej propojit s jazykem C a C++, databázemi, případně nahrát hotovou základní knihovnu pro rozpoznávání přirozeného jazyka apod. Samozřejmostí je podpora všech nejpoužívanějších platforem.

B-Prolog – implementace od spol. Afany Software, která také splňuje standard ISO Prolog, dokáže obousměrně spolupracovat s jazyky C, C++ a Java a pro osobní, nekomerční nebo akademické účely je zdarma.

Právě B-Prolog používám v dalších programových ukázkách.

(TIP: doporučuji vybrat si jiný interpretátor – vysvětlení viz závěr)

Interpretátor Prologu v praxi

Podívejme se nejprve na praktickou ukázku. Spustíme interpretátor, který nás vyzývá posloupností znaků „| *?-*“ k zápisu příkazů (anglicky ***prompt***), je to *interaktivní mód* interpretátoru.

Začneme klasicky, tedy výpisem textu *Ahoj, světe!* na standardní výstup neboli na obrazovku:

```
| ?- write('Ahoj, svete!').
Ahoj, svete!
yes
```

Prolog vyhodnocuje dotazy v závislosti na predikátech ve vnitřní *databázi*, kterou jsme ale zatím sami nevytvořili. Predikát *write* je součástí standardu ISO Prolog, který je v interpretátoru zabudován. Výpis textu „*Ahoj, svete!*“ na obrazovku je u vyhodnocení tohoto predikátu vlastně vedlejší efekt.

Slovo *yes* na konci výpisu znamená, že byl cíl vyhodnocen *úspěšně*. Pokud by se vyhodnocení nepovedlo, interpretátor vypíše *no*.

Ať už interpretátor vyhodnotí predikát úspěšně či neúspěšně, následuje opět interaktivní mód, takže je na obrazovku vypsán „prompt“ a čeká se na vstup od uživatele.

Syntax a terminologie

Řádek `write('Ahoj, svete!').` je z logiky *term*. Každý term je v Prologu ukončen *tečkou*. V tomto případě zastupuje *dotaz (cíl)*, který chceme vyhodnotit. Termy se v Prologu dělí na *konstanty* (čísla nebo atomy), *proměnné*, *struktury* a *pravidla*.

Struktura – část před závorkou *write* se nazývá hlava, přesněji funktor, který musí být obecně *atom*. Uvnitř závorek je také atom `'Ahoj, svete!'`, ale obecně to může být jakýkoliv jiný term.

Termům uvnitř závorek se říká *argumenty*. Až budeme později definovat vlastní fakta, argumentů může být libovolný počet a jsou odděleny čárkou. Když nepoužijeme žádný argument, vynecháme i závorky. Počet argumentů se označuje jako *arita*. Pokud mluvíme o konkrétní struktuře, popisujeme ji způsobem *funktor/arita*, konkrétně *write/1*.

Jestliže mají struktury totožný funktor, ale odlišnou aritu, jsou také odlišné. Podle arity nazýváme struktury *nulární* (bez argumentů, taktéž je nazýváme objekty), *unární* (1 argument), *binární* (2 argumenty) atd.

Atom – jinými slovy bezkontextový „popis“, neboli objekt.
Zapisujeme jej třemi způsoby:

1) Symbolické jméno začínající malým písmenem a jako další znaky jsou kromě malých a velkých písmen povoleny pouze číslice a podtržítka.
2) Libovolný řetězec symbolů uzavřený v jednoduchých uvozovkách (apostrofech), v němž můžeme používat i různé speciální znaky (např. `'Ahoj, svete!'`).
3) Jeden (nebo posloupnost) z povolených speciálních znaků (`+ - * /` apod.).

Číslo – může být celé (integer) nebo s plovoucí desetinnou čárkou a není omezeno velikostí hodnoty. Celá čísla můžeme zapisovat v desítkové (*576*), osmičkové (*0o123*), hexadecimální (*0xC2A*) i binární (*0b011*) soustavě. Desetinné číslo zapisujeme pouze v desítkové soustavě a používáme *desetinnou tečku*. Zápisy mohou vypadat následovně:

```
5.33    -5.33    5.33E2    5.33E-5
```

kde *E* je exponent. Zápis *5.* nebo *.33* není povolen.

Proměnná – vždy se zapisuje s počátečním velkým písmenem nebo podtržítkem a mohou následovat další znaky. Můžeme použít i číslice.

Pro jednoduché programy používáme běžně jednopísmenné názvy, u složitějších bychom měli pojmenovávat proměnné dle kontextu *celými názvy*, abychom se v programu vyznali.

Pokud je proměnná zapsána pouze podtržítkem, říkáme, že je *anonymní*. Interpretátor jí přiřadí hodnotu, ale po vyhodnocení ji nezmiňuje ani s ní nemůžeme dále pracovat.

Proměnné se dosazují např. do struktur, které chceme určit *abstraktně*. Používají se místo konkrétních objektů.

Seznam – chová se obdobně jako v jazyce LISP – každý seznam má hlavu (první prvek) a tělo (orig. „tail“), což je zbytek seznamu. Výčet prvků seznamu se zapisuje do hranatých závorek `[a, b, c].`

Seznam se hodí převážně pro případy, kdy potřebujeme pro nějaký funktor *proměnlivý počet* argumentů. Nepotřebujeme předem vědět, kolik jich bude, a seznam se vždy *přizpůsobí*.

Může obsahovat atomy, struktury nebo další seznamy. Seznamy se mohou *vnořovat* neomezeně:

```
[prvek1, predikat(1, X), Y, [1,2,3]]
```

Souvislost s logikou

Algoritmus pro určení pravdivostní hodnoty formule je *rezoluční metoda*. K tvorbě jazyka autoři Prologu přesně vymezili tvar formulí, vstupujících do rezoluce.

Syntax Prologu je založena na *Hornových klauzulích*:

Klauzule je *disjunkce literálů*.

Formule $a \vee \neg b \vee c \vee d$ je klauzule.

Formule $(a \wedge \neg b) \Leftrightarrow (c \vee d)$ není klauzule.

Formule $\neg(a \vee \neg b \vee c \vee d)$ není klauzule.

Hornova klauzule je klauzule, která obsahuje *nejvýše jeden pozitivní literál*, ostatní jsou negované.

Mějme Hornovu klauzuli $\neg a \vee \neg b \vee \neg c \vee d$, což je ekvivalentní zápis implikace $(a \wedge b \wedge c) \Rightarrow d$, protože $f \Rightarrow g \Leftrightarrow \neg f \vee g$.

Tzn. platnost d vyplývá z platnosti všech literálů a, b, c.

Hornovy klauzule lze obecně zapsat jako *implikace*.

V predikátové logice můžeme kromě prostých faktů používat i *proměnné*, *predikáty* a *kvantifikátory* proměnných – ∀ a ∃ .

Disjunkci literálů lze zapsat jako množinu literálů:

$$p(X,Y) \Leftarrow q(X),\ r(X,Y,a),\ s(Y,b)$$

Obecně:

$$b \Leftarrow a_1,\ a_2,\ \ldots,\ a_{n-1},\ a_n$$

Při *procedurální interpretaci* Hornovy klauzule chápeme b jako proceduru, jejíž tělo sestává z procedur a_1, …, a_n, které se postupně volají. Tato úvaha je ekvivalentní postupu zpracování v počítači a v Prologu.

V Prologu Hornovu klauzuli zapíšeme:

```
b :- a1, …, an.
```

Tento zápis nazýváme *pravidlo*. Jako symbol implikace je použita posloupnost znaků `:-`, tedy dvojtečka bezprostředně následovaná pomlčkou. Nesmíme zapomenout ukončit klauzuli tečkou.

Vlastní program a ukázkové příklady

Když už známe základní *syntax* a stavební prvky Prologu, vytvoříme si *vlastní program*, kterým definujeme prologovskou *relační databázi*. K tomu potřebujeme libovolný *textový editor*.

Vytvoříme prázdný textový soubor, který *vhodně pojmenujeme*. Jako přípona souboru se dle konvencí používá *.pl*. V některých případech se může přiřazení této koncovky v systému krýt se zdrojovými soubory programů jazyka Perl. V těchto případech tvůrci SWI Prologu (ale jen v SWI Prologu !) v dokumentaci doporučují používat příponu *.pro*.

Zápis kódu

Pro ukázku si navrhneme malou databázi domácích mazlíčků:

```
% Psi
pes(alik). pes(rony). pes(besi).
% Kocky
kocka(micka). kocka(mnauka). kocka(andy).
% Kralici
kralik(ferda). kralik(matilda).
```

Podívejme se blíže na význam tohoto kódu – viz následující fólie.

```
% Psi
pes(alik). pes(rony). pes(besi).
% Kocky
kocka(micka). kocka(mnauka). kocka(andy).
% Kralici
kralik(ferda). kralik(matilda).
```

Definovali jsme celkem 8 faktů pomocí tří unárních predikátů *pes*, *kocka* a *kralik*. Predikáty nám určují *relace* mezi termy. Zápis faktu `pes(rony)` můžeme „přeložit“ jako „Rony je pes“.

Na prvním řádku znakem % začíná jednořádkový *komentář* a až do odřádkování můžeme psát jakékoliv znaky, které nebudou interpretátorem zpracovány. Pokud chceme psát komentář na více řádků, ohraničíme text posloupností znaků /* na začátku komentáře a */ na jeho konci.

Za každým termem musí být napsána *tečka*. Ta slouží k odlišení začátku zápisu dalšího termu stejně jako středníky v jazyce Java. Proto můžeme v programech zapsat více termů na jednu řádku. Tečkou je třeba ukončit i *vstup* od uživatele nebo ze souboru!

Čistota kódu

Komentáře jsou u rozsáhlejších programů nutností. Někteří programátoři v praxi namítají, že nejlepším komentářem má být kód samotný. Vzhledem k transparentnosti Prologovských zápisů bychom se toho mohli držet, ale jestliže píšeme složitější pravidla anebo nejasné či na první pohled nelogické zápisy, je dobré komentáře používat.

K přehlednosti patří i správné a logické pojmenování predikátů a proměnných – raději použijte *víceslovný název* než zkratku.

Použití pravidel a proměnných

Rozšíříme si naši databázi mazlíčků. Definujeme nový predikát *zvire/1*. Můžeme tvrdit, že všechna jména v naší databázi představují jména zvířat.

Abychom nemuseli definovat pro každý atom nový fakt (například že Alík je zvíře, Micka je zvíře atp.), použijeme proměnnou a zapíšeme následující pravidlo:

```
zvire(X) :- pes(X); kocka(X); kralik(X).
```

```
zvire(X) :- pes(X); kocka(X); kralik(X).
```

Středník v tomto zápisu znamená *disjunkci* neboli *„nebo“*. Čteme *„X je zvíře, KDYŽ X je pes NEBO kočka NEBO králík“*. Kromě disjunkce můžeme použít ještě logickou konjunkci *„a zároveň“*. K tomu slouží operátor *,/2*.

Definujeme si pro ukázku několik nových pravidel. Přidáme do databáze ještě křečka a řekneme, že psi, kočky a křečci žerou maso a králíci i křečci žerou býlí. Tím dokážeme odlišit skupiny masožravců, býložravců a všežravců:

Programování v Prologu

```
pes(alik). pes(rony). pes(besi). % psi
kocka(micka). kocka(mnauka). kocka(andy). % kocky
kralik(ferda). kralik(matilda). % kralici
krecek(tony). % krecci
% Strava
zere_maso(X) :- pes(X); kocka(X); krecek(X).
zere_byli(X) :- kralik(X); krecek(X).
% Potravinove strategie
masozravec(X) :- zere_maso(X), not zere_byli(X).
bylozravec(X) :- zere_byli(X), not zere_maso(X).
vsezravec(X) :- zere_byli(X), zere_maso(X).
```

V pravidlech se objevuje predikát *not/1*, který má význam negace predikátu.

U námi vytvořených programů velice *záleží na pořadí* termů. Prolog při vyhodnocování postupuje v databázi *shora dolů* a v těle klauzule *zleva doprava*.

Podívejme se, jak s naším programem můžeme pracovat. Když spustíme interpretátor klasickou cestou, jeho vnitřní databáze obsahuje pouze *základní zabudované predikáty*. O našem kódu se musí nějakým způsobem „dozvědět“. K tomu slouží predikát *consult/1*, jehož argumentem je absolutní či relativní cesta k našemu souboru.

Implementace B-Prolog 8.1 na Linuxu hledá soubory relativně od umístění, ze kterého byla spuštěna, tedy z aktuální otevřené cesty v Shellu. Předpokládejme, že náš soubor, pojmenovaný *Mazlicci.pro*, je umístěn ve složce, ze které interpretátor spouštíme. Stačí tedy použít následující zápis:

```
|?- consult('Mazlicci.pro').
consulting::Mazlicci.pro
yes
```

V B-Prologu lze použít i jednodušší formu `['Mazlicci.pro']`, zastupující právě predikát *consult.*

Interpretátor potvrdil syntaktickou správnost a přidání našich termů do databáze. Abychom viděli, co všechno interpretátor dosud nahrál do své vnitřní databáze, použijeme predikát *listing/0*:

```
| ?- listing.
krecek(tony).
kralik(ferda).
kralik(matilda).
pes(alik).
pes(rony).
…
```

A další…

Jak je vidět, interpretátor seřadil fakta tak, aby bylo vyhledávání optimalizované. Pokud jednotlivé termy v souboru *Mazlicci.pro* promícháme, intepretátor vypíše na obrazovku informaci:

*** Warning: Predicate is not defined contiguously: pes/1*

a obdobně i pro ostatní predikáty.

Přesto jsou v databázi vnitřně seřazeny stejně jako u předchozího programu. Přehazují se mezi sebou ale pouze celé *bloky* spolu souvisejících predikátů. Stejná fakta nebo pravidla mezi sebou kvůli závislosti na pořadí interpretátor prohodit samozřejmě *nesmí*.

Na takové databázi si vyzkoušíme zadávání *cílů*:

```
| ?- pes(rony).
yes
| ?- pes(micka).
No
| ?- bylozravec(ferda).
yes
| ?- bylozravec(tony).
no
| ?- masozravec(tony).
no
| ?- vsezravec(tony).
yes
```

Činnost interpretátoru a standardní predikáty

Zkusme nyní v aktuálně nahraném programu *Mazlicci.pro* vyhodnotit cíl `zvire(tony).` Ze zápisu víme, že tony je křeček a v reálném světě je to jistě zvíře. Prolog nám ale odpoví *negativně*. Prologovská databáze je totiž *uzavřený svět*, kde platí právě taková pravidla a fakta, která si my *sami definujeme*. Proto, aby interpretátor věděl, že křeček je zvíře, musíme upravit zápis predikátu zvire/1 následovně:

```
zvire(X) :- kocka(X); pes(X); kralik(X); krecek(X).
```

Tedy přidat křečka do výčtu zvířat. Potom je vše v pořádku. Musíme si ale dávat velký pozor na souvislosti.

Vyhodnocování cílů

Vraťme se k prvnímu příkladu této prezentace. Abychom vypsali na obrazovku text *„Ahoj, světe!“*, využili jsme k tomu predikát *write/1*. Ten je navržen tak, aby jeho vyhodnocení *vždy uspělo*.

Vypsání atomu uvnitř predikátu na obrazovku není při vyhodnocení cíle standardní chování, proto můžeme říci, že je to *vedlejší efekt*.

Sami můžeme definovat takový predikát, který při vyhodnocení vypíše libovolný text. K tomu využijeme právě již existující predikát *write/1*. Vytvoříme dva predikáty pozdrav, jeden nulární a jeden s argumentem, a predikát *vypis_pozdrav/0*:

```
pozdrav(Osloveni) :- write('Ahoj '), write(Osloveni).
pozdrav :- pozdrav(lidi), nl, write('(Obecny pozdrav)').
vypis_pozdrav :- write('Prolog zdravi'), nl, pozdrav.
```

Záleží zde na pořadí. Nejprve se kontroluje hlava predikátu, následně arita a nakonec se provádí tělo. Na predikátu *pozdrav* si ukážeme, jak funguje „přetěžování“:

```
| ?- pozdrav('Pepo').
Ahoj Pepo
yes
| ?- vypis_pozdrav.
Prolog zdravi
Ahoj lidi
(Obecny pozdrav)
yes
```

Blíže se podíváme na vyhodnocení cíle *vypis_pozdrav/0*:

Volání `write('Prolog zdravi')` jistě uspěje.

Další predikát v řadě je *nl/0*. To je také zabudovaný predikát, který na aktuální výstup vypíše *novou řádku* a také se ho vždy podaří vyhodnotit. Interpretátor pokračuje voláním predikátu *pozdrav/0*. Ten jsme si sami definovali, takže interpretátor vyhodnotí jeho tělo a zjevně uspěje. Vyhodnocování se dokončí výpisem textu na novou řádku.

Cíl byl vyhodncen v *přímém chodu* a nebylo třeba žádných dalších mechanismů. Podívejme se ale, co se stane, když upravíme predikát *pozdrav/0* takto:

```
pozdrav :- pozdrav(lidi), fail.
```

Vyhodnocení zabudovaného predikátu *fail/0* je z návrhu *neúspěšné*, tedy uměle určíme, že predikát *pozdrav/0* bude také vždy neúspěšný.

V okamžik, kdy některý z posloupnosti predikátů je neúspěšně vyhodnocen, přichází na řadu *backtracking*.

Jak funguje backtracking?

Český význam slova *backtracking* je *„zpětný chod“*. Pokud má interpretátor na výběr více možností vyhodnocení predikátu, *vždy* vybere první možnost.

Pokud se nepodaří vyhodnotit další část posloupnosti procedur, vrátí se a zkusí použít jiné řešení, pokud existuje.

Jestliže další řešení neexistuje, vrátí se od tohoto místa o jeden krok zpět a postup opakuje – postupuje zprava doleva.

Přidáme si ještě jeden predikát pro pozdrav pro názornou ukázku backtrackingu. Nyní máme definována tato pravidla:

```
pozdrav(Osloveni) :- write('Ahoj '), write(Osloveni).
pozdrav :-
   pozdrav(lidi), nl, write('(Obecny pozdrav)'), fail.
% zachyceni chyby predikatu pozdrav/0
pozdrav :- write('Pozdrav se nezdaril').
vypis_pozdrav :- write('Prolog zdravi'), nl, pozdrav.
```

Podívejme se na vyhodnocení predikátu *vypis_pozdrav/0*:

```
| ?- vypis_pozdrav.
Prolog zdravi
Ahoj lidi
(Obecny pozdrav)
Pozdrav se nezdaril
yes
```

Po neúspěchu u predikátu *fail/0* se interpretátor vrátí zpět na predikát *write/1*, který se při zpětném chodu nikdy nevyhodnotí kladně a znovu nic nevypíše, stejně jako *nl/0*.

Dále interpretátor pokračuje na `pozdrav(lidi)` a zkusí najít jinou možnost řešení nebo jinou definici tohoto pravidla.

Žádná jiná možnost není, proto vstoupí zpět tam, odkud vyhodnocování začalo, tedy do těla struktury *vypis_pozdrav/0*. Interpretátor zkusí vyhledat další výskyty predikátu *pozdrav/0*.

Ten *nalezne* a při vyhodnocování *uspěje* s výpisem *„Pozdrav se nezdaril“* a celé vyhodnocení je úspěšné, proto je konečná odpověď *yes*.

Proměnné a unifikace

U pravidla *pozdrav/1* vidíme použití proměnné *Osloveni.*

```
pozdrav(Osloveni) :- write('Ahoj '), write(Osloveni).
```

Proměnná v Prologu může být v jednom z následujících stavů:

- *Volná proměnná* – výchozí stav všech proměnných.
- *Vázaná proměnná* – proměnná se naváže na libovolný term. Tomuto procesu říkáme *unifikace*.

Unifikace je úspěšná v těchto případech:

- Stejný term je unifikován *sám se sebou*.
- Obě proměnné jsou vázány na *stejný term*.
- Unifikujeme jednu *volnou* a jednu *vázanou* proměnnou nebo *jiný term* – pak je volná proměnná navázána na term.
- Pokud se ho účastní *dvě volné proměnné*. Proměnné, které byly úspěšně ztotožněny, se potom chovají jako stejná proměnná – navázáním jedné z nich je na stejný term navázána i druhá.

Navázání nelze zrušit nebo změnit s výjimkou zpětného chodu.

Proměnná existuje jako volná jen do chvíle, než je *poprvé unifikována* s termem, který není volnou proměnnou, a potom už je *trvale vázána* na tento term.

Datové typy

Prolog rozlišuje datové typy termů, ale jedná se o *slabě typovaný jazyk*. Proměnné nedeklarujeme a většinou ani nepotřebujeme znát typ termu, se kterým interpretátor proměnnou unifikoval.

V případě potřeby lze typ ověřit následujícími predikáty:

`atom(X)`, `atomic(X)` (včetně čísel), `float(X)` (nebo `real(X)`), `integer(X)`, `number(X)`, `nonvar(X)`, `var(X)`, `compound(X)`, `ground(X)`, `callable(X)`.

S *pravdivostní hodnotou* (neboli boolean) *nelze* v Prologu pracovat např. za pomoci proměnných. Zápisem `X = true` *neunifikujeme* *X* s pravdivostní hodnotou, ale s termem.

Unifikaci si ukážeme konkrétně na programu *Mazlicci.pro*. Budeme se chtít například seznámit se jmény zvířat v databázi. Položíme následující dotaz:

```
| ?- zvire(Zvire).
Zvire = micka ?
```

Interpretátor prohledává databázi *shora dolů* a hledá predikát *zvire* s aritou 1. Ten máme definovaný pouze jednou jako pravidlo, jehož hlava je `zvire(X)`. Aby byly predikáty shodné, přiřadí se proměnné *Zvire* prázdná proměnná *X*.

Následně se vyhodnocuje tělo pravidla *zleva doprava*. První je uveden predikát `kocka(X)`.

Proměnná *X* stále nemá žádnou hodnotu a hledá se predikát *kocka/1* v databázi opět shora dolů.

První je nalezen fakt `kocka(micka)` a aby byl shodný s `kocka(X)`, musí platit *X* = *micka*. Je to pouze fakt, proto vyhodnocení uspěje a kladná odpověď se vrací zpět do těla predikátu *zvire/1*. Žádné další predikáty již vyhodnocovat nemusíme (ty jsou za středníkem, tedy volitelné), proto i celé pravidlo uspěje.

Nyní již máme přiřazení *Zvire* = *X* = *micka* a cíl je vyhodnocen celý. Pokud jsme s výsledkem spokojeni, *potvrdíme* klávesou enter, interpretátor odpoví *yes* a jsme zpět v interaktivním uživatelském režimu. My se ale s takovou odpovědí nespokojíme.

Řízení backtrackingu

Ze zápisu víme, že predikát *zvire/1* jistě splňuje více mazlíčků v databázi. Pro nalezení další možnosti použijeme středník a potvrdíme klávesou enter. Přiřazení proměnné se v průběhu celého procesu změní na *mnauka* a znovu se vypíše. V praxi to vypadá takto:

```
| ?- zvire(Zvire).
Zvire = micka ?;
Zvire = mnauka ?;
Zvire = andy ?;
Zvire = alik ?
```

Postupně se vypíšou všechna jména zvířat. Vypadá to jednoduše, ale v interpretátoru je pod tím skryt poněkud složitější proces.

Středníkem řekneme interpretátoru, že poslední přiřazení do proměnné, tedy *micka*, není správné a tím zařídíme, že poslední vyhodnocený predikát v řadě neuspěje. Konkrétně to byl fakt `kocka(micka)`.

Proces se vrátí opět až do těla pravidla s hlavou `zvire(X)` s tím, že proměnná X je nyní opět *nepřiřazena*, a interpretátor se pokouší najít další výskyt predikátu *kocka/1*.

Ten nalezne, uspěje, přiřadí proměnnou *Zvire = X = mnauka* a vypíše.

Po třetím opakování již nenalezne ani další predikát *kocka/1*, ale stále v těle následuje za středníkem volitelná podmínka `pes(X)`, kterou se povede vyhodnotit. Takto můžeme projít všechna zvířata a u posledního výskytu interpretátor automaticky vyhodnocování ukončí s kladnou odpovědí.

Proměnnou můžeme před voláním ještě *přiřadit*. To bývá někdy zdrojem chyb v programu. Položíme například takovýto dotaz:

```
| ?- X = tony, zvire(X).
X = tony
yes
```

Vyhodnocení je stejné, ale do pravidla již vstupuje definovaná proměnná, proto se unifikuje odpovídající proměnná v jeho těle. Stejně pro tento případ funguje i dotaz `zvire(tony)`.

Když budeme chtít zjistit, která zvířata jsou býložravci, použijeme následující dotaz:

```
| ?- bylozravec(Zvire).
Zvire = ferda ?;
Zvire = matilda ?;
no
```

Vyhodnocování probíhá totožně s předchozím příkladem.

Jelikož jsou pravidla složitější, interpretátor *dopředu neví*, jestli je *matilda* poslední možnost a ještě čeká na interakci od uživatele.

Chceme-li vyvolat další backtracking, cíl se již vyhodnotit *nepovede* a dostaneme negativní odpověď.

Řízení v pravidlech

Backtracking je pro nás velice užitečný, ale někdy nám dokáže přidělat vrásky. Proto potřebujeme mít možnost jej nějakým způsobem kontrolovat i při definování pravidel.

Řekněme, že všechna zvířata mají srst, až na Besi, která je naháč. Definujeme následující pravidla:

```
% Vlastnosti
ma_srst(besi) :- fail.
ma_srst(X) :- zvire(X).
```

Když načteme program do databáze a zadáme cíl `ma_srst(besi)`, dostaneme přesto *kladnou* odpověď. To zapříčinil proces *zpětného chodu*, který po neúspěchu unifikoval cíl s následujícím predikátem *ma_srst/1*, jehož tělo *besi* splňuje.

Pro zabránění backtrackingu využijeme predikát *!/0* nazvaný *cut* neboli volně přeloženo *„řez“*. S výhodou jej využijeme např. pro definování početních funkcí, zabránění nekonečné rekurze, ale i v mnoha dalších případech. Řez zde použijeme následovně:

```
ma_srst(besi) :- !, fail.
ma_srst(X) :- zvire(X).
```

Pro konkrétní dotazy nyní interpretátor reaguje správně:

```
| ?- ma_srst(besi).
no
| ?- ma_srst(micka).
yes
```

Zápisu `!, fail` se říká *„cut with failure“.*

Obecně predikát *cut* interpretátoru *zabrání změnit unifikaci*, kterou učinil před jeho vyhodnocením.

V podstatě odřízne příslušné větve rozhodovacího stromu.

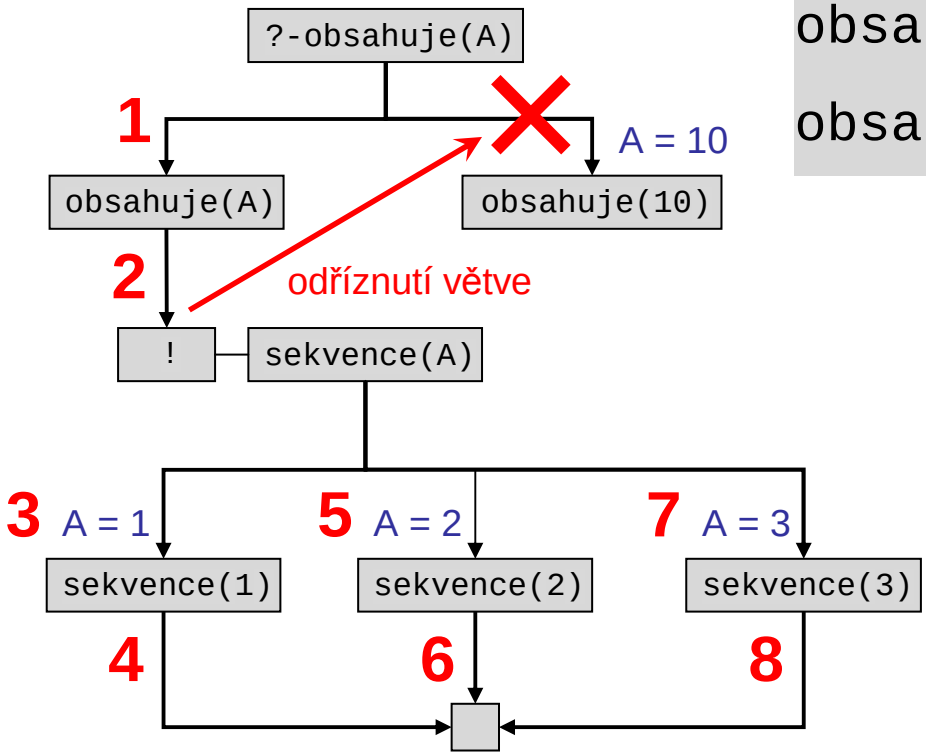
Podívejme se, co se stane, pokud v následujícím programu použijeme či nepoužijeme řez:

```
sekvence(1). sekvence(2). sekvence(3).
obsahuje(A) :- !, sekvence(A), !.
obsahuje(10).
```

```
sekvence(1). sekvence(2). sekvence(3).
obsahuje(A) :- !, sekvence(A), !.
obsahuje(10).
```

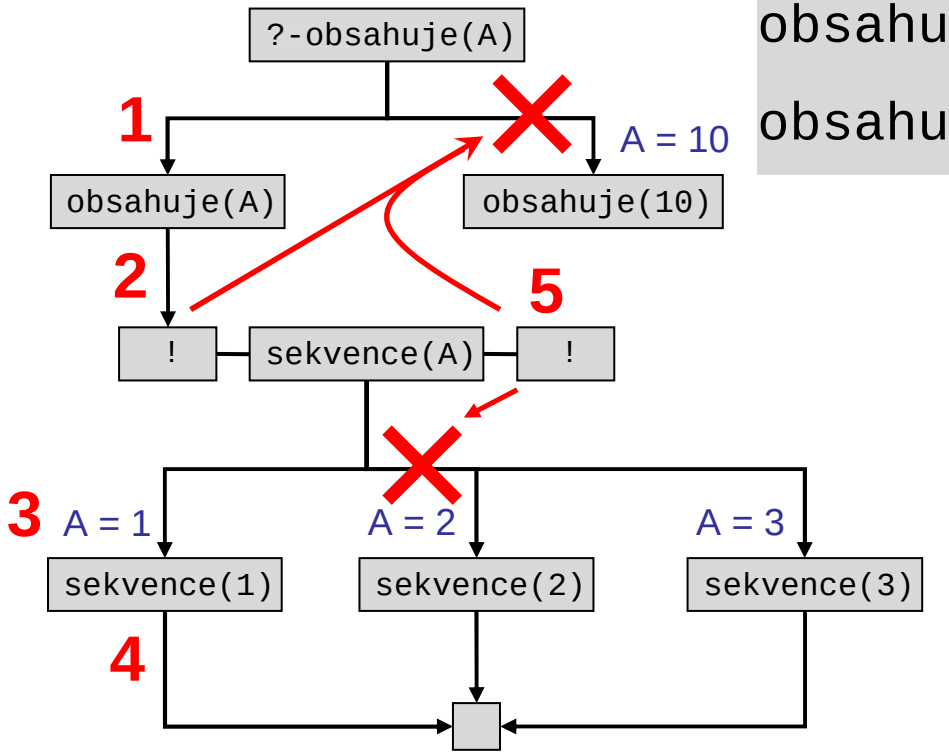
Pokud řezy vůbec nepoužijeme a zkusíme dotaz `obsahuje(A)`, dostaneme se k odpovědím – *A = 1*, *A = 2*, *A = 3* a *A = 10*.

Pokud ponecháme pouze první řez, odpovědi budou pouze *A = 1*, *A = 2* a *A = 3*. Vlivem řezu Prolog poslední klauzuli nevyhodnotí.




```
obsahuje(A) :- !, sekvence(A).
obsahuje(10).
```

Pokud ponecháme pouze druhý řez, odpověď bude právě *A = 1*.
Totožné chování bude i při použití obou řezů najednou:




```
obsahuje(A) :- !, sekvence(A), !.
obsahuje(10).
```

Bohužel ani takové ošetření nefunguje ve všech případech správně. Položme obecný dotaz „kdo má srst?“:

```
| ?- ma_srst(X).
no
```

Negativní vyhodnocení nastane proto, že interpretátor unifikuje proměnnou *X* jako první s atomem *besi*, který skončí s chybou a zpětný chod jsme řezem *zakázali*.

Lepší přístup je takový, jaký jsme si již ukázali u potravinových strategií. Definujeme si *pomocný predikát* *nahac/1* a budeme tvrdit, že srst má zvíře, které není naháč.

Takto bude vypadat upravená část:

```
% Vlastnosti
nahac(besi).
ma_srst(X) :- zvire(X), \+ nahac(X).
```

Nyní když zkusíme vyhodnotit cíl `ma_srst(X)`, dostaneme správné výsledky.

V kódu vidíme nový predikát (operátor) *\+/1*. Ten má stejnou funkci jako *not*.

Návratová hodnota

Jako např. v jazyce C máme funkce, které nám vracejí po provedení hodnotu daného typu (chybový kód apod.), můžeme podobně získat *návratovou hodnotu* z vyhodnocení predikátu. Přidáme si dva nové predikáty do programu *Mazlicci.pro*:

```
dlouhe_usi(X) :- kralik(X). dlouhe_usi(rony).
dlouhe_usi(alik).

vzhled_zvirete(Zvire, chlupate) :- ma_srst(Zvire).
vzhled_zvirete(Zvire, bez_srsti) :- nahac(Zvire).
vzhled_zvirete(Zvire, usate) :- dlouhe_usi(Zvire).
```

Při definování predikátu nemůžeme nijak explicitně určit návratovou hodnotu, proto je predikát binární a *návratovou hodnotu* představuje *druhý argument*.

Nyní program opět načteme a vyzkoušíme funkčnost. Když chceme vědět, jak vypadá *matilda*, položíme dotaz tímto způsobem:

```
| ?- vzhled_zvirete(matilda, X).
X = chlupate ?;
X = usate ?
yes
```

Návratové hodnoty využijeme hojně, např. při definování matematických operací jako třebas porovnávání dvou hodnot:

```
vetsi_cislo(Vetsi, Mensi, Vetsi) :- Vetsi > Mensi, !.
vetsi_cislo(_, Vetsi, Vetsi).
```

Tento příklad bude fungovat správně. Jako třetí argument zadáme v dotazu *nedefinovanou proměnnou*, do které je předáno větší číslo. Všimněme si při této příležitosti použití *řezu*.

Kdybychom jej vynechali, může nastat následující problém:

```
| ?- vetsi_cislo(8, 5, X).
X = 8 ?;
X = 5
```

V programu se můžeme ptát i na *konkrétně definované číslo*, o kterém si myslíme, že je větší:

```
| ?- vetsi_cislo(8, 5, 8).
yes
```

Takto položený dotaz bez proměnných se úspěšně unifikuje se zadaným pravidlem `vetsi_cislo(X, Y, X).` V tomto případě by ale interpretátor vyhodnotil kladně i dotaz:

```
| ?- vetsi_cislo(8, 5, 5).
yes
```

Proto *nesmíme* spoléhat na postupné vyhodnocování, protože mohou nastat případy, kdy se bude chovat program jinak, než si představujeme.

Raději budeme přidávat do pravidel kontroly navíc, abychom zajistili robustnost řešení.

Standard ISO Prolog

Část standardních zabudovaných predikátů jsme si již představili na předchozích stránkách, ale stále existuje hodně zabudovaných predikátů, které by se nám v další práci mohly hodit.

Kompletní seznam najdeme v příloze A knihy Prolog Programming in Depth od M. Covingtona. Ukážeme si zde na příkladech pro nás ty nejzajímavější.

Operátory

Kromě standardních struktur se v Prologu setkáme i s *operátory*. Ty se používají nejen pro numerické operace, ale můžeme si definovat i *vlastní* s požadovanou funkcí.

Zavedeny jsou proto, abychom například operaci sčítání nemuseli zapisovat ve funktorové notaci, tedy `+(4, 7)`, ale mohli jsme použít přirozenější zápis `4 + 7`.

Existují *tři druhy* operátorů:

- **Prefixové** – jejich zástupcem je predikát *not/1*. Je to unární predikát a píše se vždy *před* term, který představuje jeho argument.

- **Infixové** – jsou všechny binární operátory. Zapisují se *mezi* dva termy, které představují jeho argumenty. Je to např. oprátor disjunkce *(,)/2* nebo matematických operací jako sčítání *(+)/2* apod.

- **Postfixové** – píší se *za* argument, vyskytují se velice zřídka.

Zkusme nyní vyhodnotit cíl `5+3`:

```
| ?- 5+3.
***  Undefined procedure: (+)/2
```

Výjimka nastane proto, že operátor + není určen pro unifikaci. Aby mohl interpretátor unifikaci použít, naskytne se nám použití *proměnné*. Zkusíme položit dotaz takto:

```
| ?- X = 5 + 3.
X = 5+3
yes
```

Jistě víme, že Prolog v tomto případě zafungoval správně, tedy přiřadil do proměnné *X* predikát *(+)/2*. Je to stejné, jako bychom proměnné přiřadili např. strukturu `krecek(tony)` nebo atom. Aby se *vyhodnotily* aritmetické operace, musíme použít zabudovaný infixový operátor *is/2*, a to následovně:

```
| ?- X is 5 + 3.
X = 8
yes
```

Za *is* můžeme psát jakékoliv složité rovnice, bohužel ale nemůžeme používat nedefinované proměnné. Zatím nám bude stačit základní práce s operátory.

Definice vlastních operátorů

Jak již bylo řečeno, je možné definovat si vlastní operátory. Pro ukázku si vytvoříme do souboru *Osoby.pro* jednoduchou databázi osob, které jsou společně ve vztahu. Dále program rozšíříme:

```
% Muzi
muz(jan). muz(roman). muz(ales).
% Zeny
zena(stela). zena(ema). zena(lucie). zena(petra).
% Vztahy
manzele(jan, stela). manzele(ales, petra).
miluje(X, Y) :- manzele(X, Y).
```

V našem jednoduchém světě jsou tři muži a čtyři ženy, z toho dva páry jsou v manželském vztahu. Tvrdíme, že manželé se navzájem milují. Můžeme položit několik dotazů:

```
| ?- miluje(jan, stela).
yes
| ?- miluje(X, petra).
X = ales
yes
```

Ovšem přirozenější zápis by byl např. `jan miluje stela`. To se dá zařídit pomocí zabudovaného predikátu *op/3:*

```
:- op(priorita, specifikator, nazev_operatoru)
```

Priorita je celé číslo od nuly výše a znamená, v jakém pořadí se bude operátor vyhodnocovat před ostatními.

Čím vyšší číslo, tím nižší priorita, tedy pozdější vyhodnocení. Například násobení má prioritu 400, sčítání a odčítání 500 apod.

Specifikátor určuje *asociativitu* a zda bude operátor prefixový, infixový nebo postfixový. Asociativita určuje, z jaké strany se zápisy vyhodnocují. Specifikátor se skládá z určené posloupnosti znaků (atom):

Atom	Notace	Asociativita
fx	prefixová	neasociativní
fy	prefixová	asociativní zprava
xfx	infixová	neasociativní
xfy	infixová	asociativní zprava
yfx	infixová	asociativní zleva
xf	postfixová	neasociativní
yf	postfixová	asociativní zleva

Operátor *miluje/2* definujeme bez asociativity jako infixovou notaci, protože je to původně binární predikát. Definovat funktor jako operátor musíme před jeho prvním použitím v programu následovně:

```
?- op(200, xfx, miluje).
```

I v kódu musí být na začátku řádky zapsány znaky *?-*, jimiž začíná výzva v uživatelském režimu, event. znaky *:-*. Tomuto zápisu se říká *direktiva*.

Predikát *miluje/2* můžeme používat dále libovolně i ve funktorovém zápisu.

Problém symetrických relací

Zkusíme zadat dotaz za pomoci operátoru a ukážeme si nedostatek v aktuálním programu:

```
| ?- ales miluje X.
X = petra
yes
| ?- stela miluje X.
no
```

Odpověď na druhý dotaz je negativní, protože jsme definovali tělo pravidla *miluje* pouze jednostranně.

Aby pravidlo platilo *symetricky*, definujeme nový predikát *chot/2* a upravíme pravidlo následujícím způsobem:

```
chot(X, Y) :- manzele(X, Y); manzele(Y, X).
miluje(X, Y) :- chot(X, Y).
```

S touto symetrickou realcí bude Prolog vyhodnocovat předchozí dotazy správně. Pokud ale položíme obecný dotaz `chot(X,Y).`, lidé se při backtrackingu budou *opakovat*.

Tento problém se objevuje u každé symetrické relace. Vhodné řešení může být v různých případech *individuální* a neexistuje žádné univerzální pravidlo.

V našem případě stačí říci, že symetrické řešení chceme hledat pouze tehdy, když neexistuje řešení původní. K tomu použijeme nový zápis – podmínku.

Podmínka

Upravíme predikát *chot* následujícím způsobem:

```
chot(X,Y) :- \+ manzele(X,Y) -> manzele(Y,X) ;
  manzele(X, Y).
```

Zápis *predikat1 -> predikat2 ; predikat3* je podmínka „if“, jak ji známe z jiných programovacích jazyků.

První se vyhodnotí *predikat1*; pokud uspěje, zavolá se *predikat2*, v opačném případě se zavolá *predikat3* – ten je ale v zápisu nepovinný.

Pokud neuspěje *predikat2*, jako další se *predikat3* vyhodnocovat nebude kvůli závislosti implikace.

Ani teď bohužel není řešení ideální pro všechny případy. Podívejme se proto na následující dotazy:

```
| ?- Y = ales, X miluje Y.
Y = ales
X = petra
yes
| ?- X miluje Y, Y = ales.
no
```

Další problém je symetričnost u dotazu `X miluje Y.`, jelikož je výčet interpretován pouze *jednostranně*, i když chceme, aby v našem programu byl tento vztah *symetrický*.

U deklarativního programování někdy *nelze* jednoduše pokrýt všechny možnosti zadaných dotazů, ale měli bychom se soustředit hlavně na *požadovanou funkčnost*.

Vstup od uživatele

Jestliže vytváříme program, ve kterém chceme nechat uživatele *zadat libovolný vstup*, aniž by pro něj byly viditelné vnitřní struktury, použijeme k tomu predikát *read/1*. Jeho argumentem je proměnná, ve které bude následně přiřazen zadaný vstup.

Predikát si můžeme vyzkoušet i v interaktivním módu interpretátoru:

```
| ?- read(X), write('napsal(a) jste '), write(X), nl.
| ahoj.
napsal(a) jste ahoj
X = ahoj
yes
```

Můžeme napsat pouze term dle konvencí jazyka Prolog ukončený tečkou a nakonec potvrdíme klávesou enter.

Nyní víme, jak se vypisuje na obrazovku a čte z klávesnice, ale stejně jednoduchým způsobem lze pracovat se soubory.

Práce se soubory

Pro *čtení* ze souboru využijeme predikát *see/1*, který jako argument přijímá cestu k souboru. Tím určíme, že *všechny* vstupní operace budou pracovat právě s uvedeným souborem.

Predikátem *seeing/1* ověříme, s jakým vstupem interpretátor pracuje. Potom už používáme *stejné* predikáty jako pro čtení ze standardního vstupu a platí pro ně stejná pravidla.

Pro *ukončení* práce se souborem použijeme predikát *seen/0*.

Při *zápisu* do souboru postupujeme *podobně jako při čtení*. Také stačí pouze intepretátoru *přesměrovat* standardní *výstup* na náš soubor.

Jako u čtení jsou pro nás pro zápis důležité tři predikáty *tell/1*, *telling/1* a *told/0*.

Jako ekvivalent k *tell/1* je v některých verzích prologu predikát *append/1*, který umožní zápis na konec zvoleného souboru, zatímco *tell/1* aktuální obsah přepíše nebo vytvoří soubor nový, pokud neexistuje.

Cyklus

Vytvoříme jednoduchý postup, kterým přečteme všechny termy ze souboru A a zapíšeme je do souboru B:

```
cti_a_zapis_soubor(A, B) :-
  tell(B), cti_soubor(A), told.
cti_soubor(X) :-
  see(X), repeat, read(Term), \+ zpracuj(Term), seen.

zpracuj(-1) :- !, fail.
zpracuj(end_of_file) :- !, fail.
zpracuj(X) :- write(X), nl.
```

Ze zápisu už snadno poznáme, jak se interpretátor bude chovat. V pravidle `cti_soubor(X)` určíme predikátem *see/1* zdrojový soubor, následuje predikát *repeat/0*, který v přímém chodu neudělá nic, jeho vyhodnocení ale uspěje. Následuje samotné čtení a odeslání na zpracování.

Chceme přečíst *celý* soubor, tedy pokud se povede zpracovat term, necháme pravidlo neuspět pro *vyvolání backtrackingu*.

Predikát read/1 neudělá znovu nic, ale *repeat/0* znovu uspěje. Před tento predikát se backtrackingem již *nedostaneme* a program se chová, jako kdyby *našel další možné řešení*.

Po jeho použití *musí* existovat možnost, jak cíl splnit, jinak se bude cyklus opakovat donekonečna.

Konec souboru je značen buď *-1* nebo atomem *end_of_file*. Proto ve zpracování po dosažení konce souboru vyvoláme neúspěch a tím zapříčiníme splnění těla pravidla *cti_soubor/1*, jelikož *seen/0* jistě uspěje.

Pro zápis do jiného souboru pouze čtení „obalíme“ mezi predikáty *tell/1* a *told/0*.

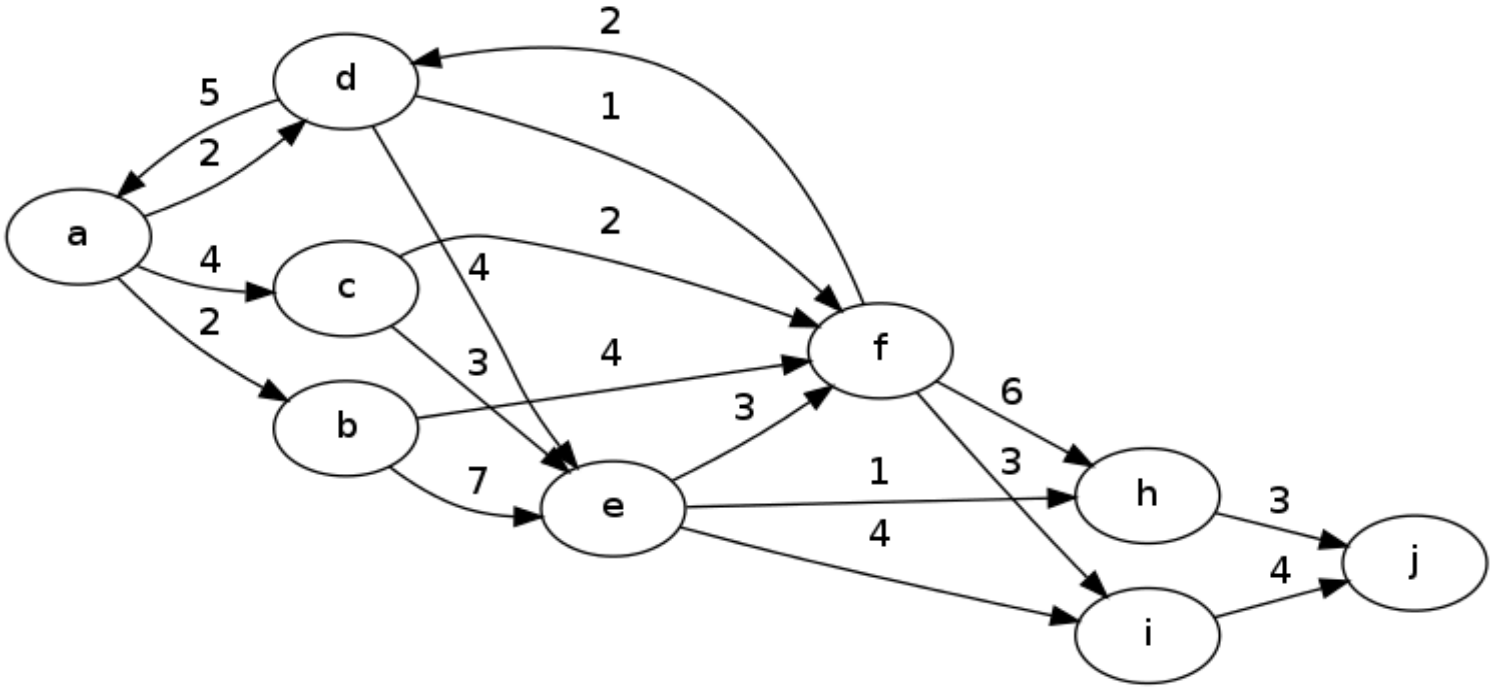
Ke čtení *jednoho* znaku slouží predikát *get/1*, pro čtení včetně „white space“ (bílých znaků, např. *mezer*) slouží *get0/1*.

Zkuste sami na konci prezentace vytvořit predikát, který přečte *libovolný* řádkový vstup bez nutnosti ukončení tečkou nebo uvození jednoduchými uvozovkami. Důležitá je znalost práce s řetězci a seznamy. **(Řešení najdete v programu *Vstup.pro*)**

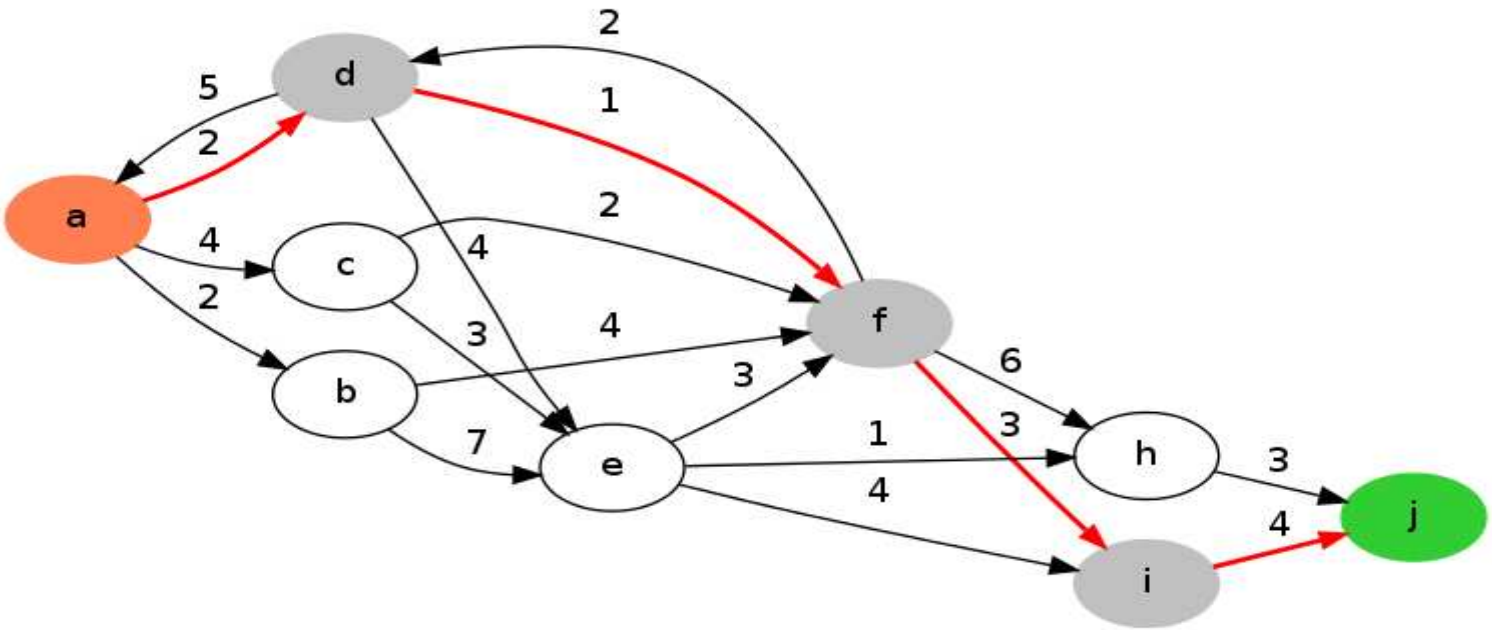
Víme, že konec řádky v Unixovém systému je určený netisknutelným znakem s označením *LF* (ASCII 10), neboli anglicky *„Line Feed“*. V systému Windows se používají dva znaky v pořadí *CR+LF*, kde *CR* (ASCII 13) znamená v angličtině *„Carriage Return“*.

Rekurze

Rekurzi si ukážeme na problému hledání *nejkratší cesty* v *orientovaném a ohodnoceném grafu*:



Na obrázku libovolným postupem vyznačíme nejkratší cestu z vrcholu *a* do vrcholu *j* a pokusíme se stejnou najít za pomoci Prologu.



Nejprve vytvoříme množinu *vrcholů* a *ohodnocených hran* mezi nimi. Jelikož nemusíme deklarovat *typy* proměnných, stačí definovat pouze hrany:

```
hrana(a, b, 2).
hrana(a, c, 4).
hrana(a, d, 2).
hrana(b, e, 7).
hrana(b, f, 4).
hrana(c, e, 3).
hrana(c, f, 2).
```

```
hrana(d, e, 4).
hrana(d, a, 5).
hrana(d, f, 1).
hrana(e, h, 1).
hrana(e, i, 4).
hrana(e, f, 3).
hrana(f, h, 6).
```

V dalším souboru načteme hrany a definujeme predikáty pro procházení grafu a hledání cesty mezi dvěma vrcholy:

```
?-consult('graf.pro').
% Cesta z A do B existuje, jestliže vede z vrcholu A hrana přímo do vrcholu B
cesta(A, B, Navstivene, Cena, Cesta) :-
 hrana(A, B, Cena), append(Navstivene, [A, B], Cesta).
% Cesta z A do C existuje, jestliže existuje spojení z A do B a z B vede cesta do C (rekurze)
cesta(A, C, Navstivene, Cena, Cesta) :-
 hrana(A, B, Cena_prechodu), \+ member(A, Navstivene),
 append(Navstivene, [A], N1), cesta(B, C, N1, Cena_cesty,Cesta),
 Cena is Cena_prechodu + Cena_cesty.
```

Vytvoříme predikát s uživatelsky přívětivějším vstupem i výstupem a také se naskýtá snadná možnost nalézt všechny cesty mezi A a B. Stačí použít zabudovaný *backtracking*:

```
% Nalezne první možnou cestu z vrcholu A do B, pokud existuje
cesta(A, B) :-
  cesta(A, B, [], C, P), format("Cesta přes vrcholy: ~w,
 cena cesty: ~w\n", [P, C]).

% Vyhledá všechny cesty z A do B
vsechny_cesty(A, B) :- cesta(A, B), fail.
vsechny_cesty(_, _).
```

Pro nalezení nejkratší cesty je nejjednodušší možností nalézt nejdříve *všechny* cesty a poté z nich vybrat nejkratší.

Prolog si ale standardně *nedokáže pamatovat* předchozí hodnoty, proto si můžeme pomoci právě vnitřní databází interpretátoru.

Úprava databáze

V intepretátoru *měníme obsah databáze* hojně, např. v průběhu testování nového programu za použití predikátu *consult/1* či *reconsult/1*. Když po několika takto nahraných programech vypíšeme obsah databáze voláním predikátu *listing/0*, zjistíme, že velká spousta nepotřebných predikátů zůstává v databázi. Predikáty i pravidla zde zůstávají do vypnutí interpretátoru.

Po spuštění interpretátoru můžeme do prázdné databáze přidat vlastní predikáty za pomoci dobře známého volání *consult/1*. Můžeme ale použít i jeden z predikátů *assert/1*, *asserta/1* nebo *assertz/1*. Jejich argumentem je libovolný predikát nebo atom. Takto můžeme přidávat predikáty i k již nahranému programu. Predikáty *assert* a *assertz* přidávají fakta na *konec*, predikát *asserta* přidává na *začátek*.

V případě, že chceme fakta z databáze odstranit, použijeme predikát *retract/1*. Jako argument zapíšeme predikát a první v databázi, *se kterým se tento predikát unifikuje*, bude z databáze vymazán.

Pro vymazání všech odpovídajících použijeme *retractall/1*. Pro vymazání všech definic konkrétního predikátu použijeme:

```
abolish(Funktor/Arita).
```

Predikát *abolish/0* maže *všechna* námi definovaná fakta.

Pozor na fakta a pravidla nahraná pomocí predikátu *consult*. Na jejich smazání *nemáme dostatečná oprávnění*.

A jak se nám tato funkcionalita hodí k hledání nejkratší cesty? Podívejte se sami:

```
% Vyhledá nejkratší cestu z vrcholu A do B a uloží do pomocných predikátů
nejkratsi_cesta(A, B) :-
   assert(min(100)), assert(nejkratsi([])), !,
   cesta(A, B, [], C, P),
   min(Cena), C<Cena, abolish, assert(min(C)),
   assert(nejkratsi(P)), fail.
```

Tímto způsobem prohledáme všechny cesty a cesta s nejmenší cenou bude následně uložena v databázi jako *nejkratsi/1* ve formátu seznamu. Cena této cesty bude v predikátu *min/1*.

Nyní stačí informaci přijatelně vypsat a „uklidit“ po sobě:

```
% Prohledá všechny cesty, vypíše údaje o nejkratší cestě z pomocných
predikátů a opět je smaže (pro jednoznačnost)
vypis_nejkratsi(A, B) :-
 \+ nejkratsi_cesta(A, B), nejkratsi(Cesta),
 write('Nejkratší cesta: '), write(Cesta), nl,
 write('Cena cesty: '), min(Cena), write(Cena), abolish.
```

Pozor ale na predikát abolish, který smaže všechny uložené predikáty nebo pravidla. Takový zápis je vhodné použít pouze pro jednoúčelové programy.

Seznamy

Velice užitečné jsou v Prologu *seznamy*, neboli listy. Zápis v hranatých závorkách `[a, b, c]` je zjednodušený zápis funktorové notace `.(a, .(b, .(c, [])))`.

Přidávání prvků

Jednou z možností, jak *přidávat* prvky do *existujícího seznamu*, je zápis `[termy | seznam]`, kde svislá čára zastupuje *konstruktor* seznamu. Nalevo je vypsán libovolný počet termů a napravo je seznam, do kterého se mají termy přidat:

```
| ?- List = [5, 10, 25], X is 25 - 3, Y = sqrt(121),
   List2 = [X, Y | List].
List = [5,10,25]
X = 22
Y = sqrt(121)
List2 = [22,sqrt(121),5,10,25]
yes
```

Jestliže chceme přidat prvky na konec seznamu, můžeme použít zabudovaný predikát *append/3*. Jako argumenty přijímá *výhradně listy*. Musíme si dávat pozor na přidání jednoduchého atomu:

```
| ?- X = 10, append([5, 10, 25], X, List).
X = 10
List = [5,10,25|10]
yes
```

Zápis *[5,10,25|10]* je sice dle konvencí správný, ale s takovou strukturou se dále špatně pracuje.

Řešením je vytvoření *jednoprvkového* seznamu:

```
| ?- X = 10, append([5, 10, 25], [X], List).
X = 10
List = [5,10,25,10]
yes
```

Predikát *append* využívá vnitřně konstruktor seznamu, proto stejného výsledku dosáhneme i zápisem:

```
X = 10, List = [5, 10, 25|[X]].
```

Zpracovávání prvků

Nejdůležitější část práce s prvky seznamu je jejich *zpracování*. Záleží, na co seznam využíváme, ale postup je většinou ekvivalentní.

Můžeme provést *dekompozici* seznamu na první prvek (*hlavu*) a zbytek seznamu (*tělo*). Využijeme unifikaci s pravidlem, které jako argument přijímá zápis `[Hlava|Telo]`.

Jako nejjednodušší příklad si ukážeme výpis všech prvků seznamu pod sebe:

```
vypis_prvky_seznamu(S) :- vypis_prvky_ocislovane(S, 1).
vypis_prvky_ocislovane([], _).
vypis_prvky_ocislovane([H|T], N) :-
  write(N), write('. prvek = '), write(H), nl, N1 is N+1,
  vypis_prvky_ocislovane(T, N1).
```

Vyzkoušíme si navržené řešení:

```
| ?- vypis_prvky([prvni,[1,2],3]).
1. prvek = prvni
2. prvek = [1,2]
3. prvek = 3
yes
```

Pokud bychom nedefinovali pravidlo

```
vypis_prvky_ocislovane([], _).
```

vyhodnocení by se provedlo, ale odpověď by byla negativní, jelikož *prázný seznam* by se s žádným predikátem již *neunifikoval*.

Hlava seznamu je vždy *první* term neprázdného seznamu a zbytek neboli *tělo* původního seznamu je *vždy seznam* bez prvního prvku, může být tedy i prázdný.

(Tip: zkuste si vytvořit predikát pro rekurentní vypsání prvků vnořených seznamů)

Řetězce

Libovolné textové řetězce *nejsou* jenom obyčejné *atomy*, ale je to speciální posloupnost znaků uvozená klasickými uvozovkami.

Jako v jazyce C se v Prologu řetězec reprezentuje pomocí *pole* (seznamu) jednotlivých *znaků* a každý znak má určen svůj číselný kód v ASCII tabulce:

```
| ?- X = "Retezec"
X = [82,101,116,101,122,101,99]
yes
```

Další užitečený zabudovaný predikát je *name/2*. Ten slouží k *převodu atomu na řetězec* reprezentující jeho název a naopak. *Směr* převodu určíme tím, do kterého z argumentů zadáme nepřiřazenou proměnnou. Můžeme řetězce i *porovnávat*:

```
| ?- name('Retezec', X).
X = [82,101,116,101,122,101,99]
yes
| ?- name(X, [82,101,116,101,122,101,99]).
X = 'Retezec'
yes
| ?- name('Retezec', [82,101,116,101,122,101,99]).
yes
```

Základy profilace kódu

Interpretátor dokáže pomocí predikátu *statistics/0* vypsat podrobnosti o využití paměti a další informace zaznamenané za jeho běhu. Pomocí *statistics/2* požádáme pouze o jednu konkrétní informaci.

Predikát *cputime/1* vypíše dobu uplynulou od posledního volání. Pokročilejší *time/1* zjišťuje dobu vyhodnocení volání predikátu předaného argumentem.

Při tvorbě programů využijeme *mód ladění* (anglicky ***„debugger“***), který nám může pomoci s odhalením chyb v návrhu pravidel a predikátů.

Mód v interpretátoru spustíme predikátem *trace/0*.

V tomto módu vidíme vyhodnocení jednotlivých cílů od začátku do konce, včetně veškeré unifikace.

Ladicí mód opustíme voláním *notrace/0*:

```
| ?- trace
yes
{Trace mode}
| ?- write(a).
   Call: (1) write(a) ?
a   Exit: (1) write(a) ?
yes
```

Vyhodnocování se hned zastaví a pomocí klávesy *enter* se můžeme posouvat *po krocích*, které interpretátor činí. Písmenem *s* přeskakujeme aktuálně volaný predikát a ladění se zastaví až po jeho úspěšném či neúspěšném vyhodnocení.

Písmenem *r* necháme ladicí mód vypsat všechny kroky až do konce vyhodnocení. V zastaveném ladění se dá vyvolat nápověda pomocí písmene *h* nebo znakem *?*.

V programu můžeme definovat pouze *konkrétní body*, na kterých se chceme zastavit a prozkoumat je při vyhodnocování. K tomu použijeme predikát `spy(Funktor/Arita)`. Ladící výpisy jsou poté od volání určeného predikátu stejné, podobně jako u *trace*. Jeden bod zastavení odebereme voláním *nospy/1* a všechny body voláním *nospy/0*.

Shrnutí

Vlastnosti Prologu:

- Prolog je *slabě typovaný* jazyk.
- *Proměnné* jsou v Prologu *trvale navazovány*.
- *Návratovou hodnotou* je v Prologu vždy *pravdivostní hodnota*.
- Proměnné v Prologu *nelze* navázat na pravdivostní hodnotu.
- Interpretátor hledá řešení prohledáváním do hloubky.

- Pravdivostní operátory pracují s termy, aritmetické operátory s jejich aritmetickými hodnotami.
- Prolog se používá především při *rozpoznávání přirozeného jazyka* a v *optimalizačních úlohách*.
- Obecné využití prologu je velké, přesto velmi specifické. Nevyužijeme jej například při složitých výpočtech.

Terminologie:

konstanta, proměnná, struktura, seznam = term

číslo, atom = konstanta

funktor(argumenty) = predikát

hlavička :- tělo. = pravidlo

hlavička. = fakt

?- tělo. = cíl

fakt, pravidlo, cíl = klauzule

běžné značení predikátů: funktor/arita, arita = počet argumentů

[hlavička|zbytek] = seznam; hlavička = termy oddělené čárkou; zbytek = seznam termů

"Toto je řetězec reprezentovaný seznamem znaků"

'Toto je term, který je dále nedělitelnou jednotkou'

Závěr

Interpretátor B-Prolog je malý, příjemný na ovládání a potěší svoji jednoduchou funkcionalitou.

Bohužel se v průběhu vytváření této práce *neosvědčil*. Přesto, že na oficiálních stránkách ke dni 1. 5. 2014 byla informace, že je interpretátor pro nekomerční účely zdarma, bylo mi sděleno p. Zhou, že *zdrojové kódy* potřebné k propojení s jazyky Java a C jsou přístupné až po zaplacení částky 2,980$. Tato skutečnost práci s B-Prologem *značně omezila*.

Další nedostatky jsou spojeny s neúplnou podporou nadstandardních predikátů, které i přesto, že jsou uvedeny v manuálu, v interpretátoru *nefungují*. S verzí 8.1 byla bez jakékoliv zmínky oproti verzi 7.8#5 *redukována* podpora kompilace a spouštění samostatných programů.

Navíc jeho podpora již nyní upadá a tvůrci se soustředí na vývoj nového interpretátoru pro logické programování – *Picat*.

Mohu doporučit již od začátku začít používat interpretátor podporující propojení s jazykem *Java*, *C* nebo *C#*. Právě v tom tkví velký potenciál Prologu.

Množina programů, které jsme si zde ukázali, a několik dalších je k dispozici na webových stránkách předmětu **KIV/UZI** spolu s touto prezentací.

Podrobnější popis B-Prologu, standardních predikátů a ukázky využití Prologu naleznete mimo jiné také v mé bakalářské práci.

Literatura:

Bramer, M.: Logic Programming with Prolog. University of Portsmouth, UK, 2005.
ISBN-10: 1-85233-938-1, 223 s.

Cigler, T.: Vytvoření výukového programu pro výuku programovacího jazyka Prolog.
Bakalářská práce, KIV ZČU v Plzni, 2014

Kryl, R.: Neprocedurální programování: Úvod do programovacího jazyka PROLOG.
KSVI MFF UK, Praha. Dostupné z: http://ksvi.mff.cuni.cz/ kryl/prolog.pdf

Neng-Fa Zhou: B-Prolog User's Manual. Version 7.8.
Dostupné z: www.probp.com/download/manual.pdf.

Sládek, T.: Výukový program pro předmět UIR. Bakalářská práce,
KIV ZČU v Plzni, 2014