SLEZSKÁ
UNIVERZITA
FILOZOFICKO-
PŘÍRODOVĚDECKÁ
FAKULTA V OPAVĚ

Šárka Vavrečková

Skripta do předmětu

Praktikum z logického programování

Ústav informatiky
Filozoficko-přírodovědecká fakulta v Opavě
Slezská univerzita v Opavě

Opava, poslední úpravy 27. 12. 2023

Anotace: Tento dokument je určen pro studenty předmětu *Praktikum z logického programování* vyučovaného na Ústavu informatiky Slezské univerzity v Opavě. Navazujeme na předmět Logika a logické programování, tedy předpokládáme, že se studenti už alespoň trochu orientují v logickém programování. V tomto předmětu se zaměřujeme na programovací jazyk Prolog. Studenti se naučí používat běžné programové konstrukce v tomto jazyce, včetně seznamů.

Praktikum z logického programování

RNDr. Šárka Vavrečková, Ph.D.

Ústav informatiky
Filozoficko-přírodovědecká fakulta v Opavě
Slezská univerzita v Opavě
Bezručovo nám. 13, Opava

Sázeno v systému LaTeX

Předmluva

Co najdeme v těchto skriptech

 *Rychlý náhled:* Tento text je určen studentům předmětu *Praktikum z logického programování* na Ústavu informatiky Slezské univerzity v Opavě. Předpokládá znalosti z předmětu *Logika a logické programování*, základní pojmy a metody přednášené v tomto úvodním předmětu jsou připomenuty v kapitole 1.

Programovací jazyk Prolog je založen na matematické logice a jedná se o deklarativní programovací jazyk. Je poměrně specifický v postupech při programování algoritmů, a taky specifický v tom, kde je vhodné tento jazyk použít. Studenti v tomto předmětu procházejí základní programovací techniky a vzorové úlohy k řešení.

Některé oblasti jsou „navíc“ (jsou označeny ikonami fialové barvy), ty nejsou probírány a ani se neobjeví na zkoušce – jejich úkolem je motivovat k dalšímu samostatnému studiu či pokusům nebo pomáhat v budoucnu při získávání dalších informací. Pokud je fialová ikona před názvem kapitoly (sekce), platí pro vše, co se v dané kapitole či sekci nachází.

Značení

Ve skriptech se používají následující barevné ikony:

-  *Rychlý náhled* (skript, kapitoly), ve kterém se dozvíme, o čem to bude.
-  *Klíčová slova* kapitoly.
-  *Cíle studia* pro kapitolu nám řeknou, co nového se v dané kapitole naučíme.
-  Nové *pojmy*, značení apod. jsou značeny modrým symbolem, který vidíme zde vlevo.
-  Konkrétní *postupy* a nástroje, způsoby řešení různých situací, do kterých se může správce počítačového vybavení dostat, atd. jsou značeny také modrou ikonou.
-  ,  Některé části textu jsou označeny fialovou ikonou, což znamená, že jde o *nepovinné úseky*, které nejsou probírány (většinou; studenti si je mohou podle zájmu vyžádat nebo sami prostudovat). Jejich účelem je dobrovolné rozšíření znalostí studentů o pokročilá témata, na která obvykle při výuce nezbývá moc času.

-  Žlutou ikonou jsou označeny odkazy, na kterých lze získat *další informace* o tématu. Nejčastěji u této ikony najdeme webové odkazy na stránky, kde se dané tématice jejich autoři věnují podrobněji.
-  Červená je ikona pro *upozornění* a poznámky.

Pokud je množství textu patřícího k určité ikoně větší, je celý blok ohraničen prostředím s ikonami na začátku i konci, například pro poznámku:

 **Poznámka:**

V takovém prostředí uvádíme doplňující poznámku k předchozímu textu, může zde být třeba upřesnění, rozvedení myšlenky, upozornění na vedlejší důsledky postupu apod.

Podobně může vypadat prostředí pro delší postup nebo více odkazů na další informace. Mohou být použita také jiná prostředí:

 **Příklad 0.1**

Takto vypadá prostředí s příkladem, obvykle nějakého postupu. Příklady jsou obvykle komentovány, aby byl jasný postup jejich řešení.

 **Úkol**

Otázky a úkoly, náměty na vyzkoušení, které se doporučuje při procvičování učiva provádět, jsou uzavřeny v tomto prostředí. Pokud je v prostředí více úkolů, jsou číslovány.

Obsah

Kapitola 1

Logické programování

Rychlý náhled: Tato kapitola je výběrem nejdůležitějších témat o logickém programování z předmětu Logika a logické programování. Je zde proto, aby si studenti připomněli základy, na kterých budeme v dalším textu stavět – tedy opakování.

Většina této kapitoly je převzata z poslední kapitoly skript pro předmět Logika a logické programování, aby studenti tápající v základech měli k těmto základům snadnější přístup (také pokud se najde někdo, kdo ještě předmět Logika a logické programování neabsolvoval).

Klíčová slova: Logické programování, Prolog, program, pravidlo, fakt, dotaz, cílová klauzule, anonymní proměnná, predikát rovnosti, rekurze, rezoluce.

Cíle studia: Po prostudování této kapitoly se seznámíte se základy programování v programovacím jazyce Prolog. Naučíte se sestavit program, tedy znalostní bázi, a pokládat dotazy, které mají být podle programu vyhodnoceny.

1.1 Pár slov k programovacím jazykům

Jak už víme, k nejběžnějším paradigmatům pro programování patří procedurální a deklarativní programování. Běžné programovací jazyky jako C, C++, Java, C#, Python a další jsou procedurální, což znamená, že program tvoříme jako posloupnost kroků, které je třeba udělat (tj. přímo sdělujeme, jak se má postupovat).

Deklarativní programování spočívá v tom, že netvoříme přímo postup, ale v programu sdělujeme, čeho má být dosaženo. Jedním z programovacích jazyků postavených na deklarativním paradigmatu je Prolog, k dalším patří například jazyk SQL (ten známe z databází).

K dalším paradigmatům patří funkcionální programování (kde vše je funkce) – např. Haskell, Scala, F# a Dart, známým paradigmatem je také objektové programování, paralelní a distribuované programování, datové programování (kdy je účelem zpracovávání velkého množství dat).

Zaměřit se na konkrétní paradigma můžeme volbou programovacího jazyka, ale v některých jazycích lze programovat podle různých paradigmat a do určité míry je kombinovat (ne vždy je to vhodné). Například jazyk Java je obvykle považován na procedurální objektový jazyk, a dokonce se v něm dá programovat i funkcionálně.

1.2 Logické programování v Prologu

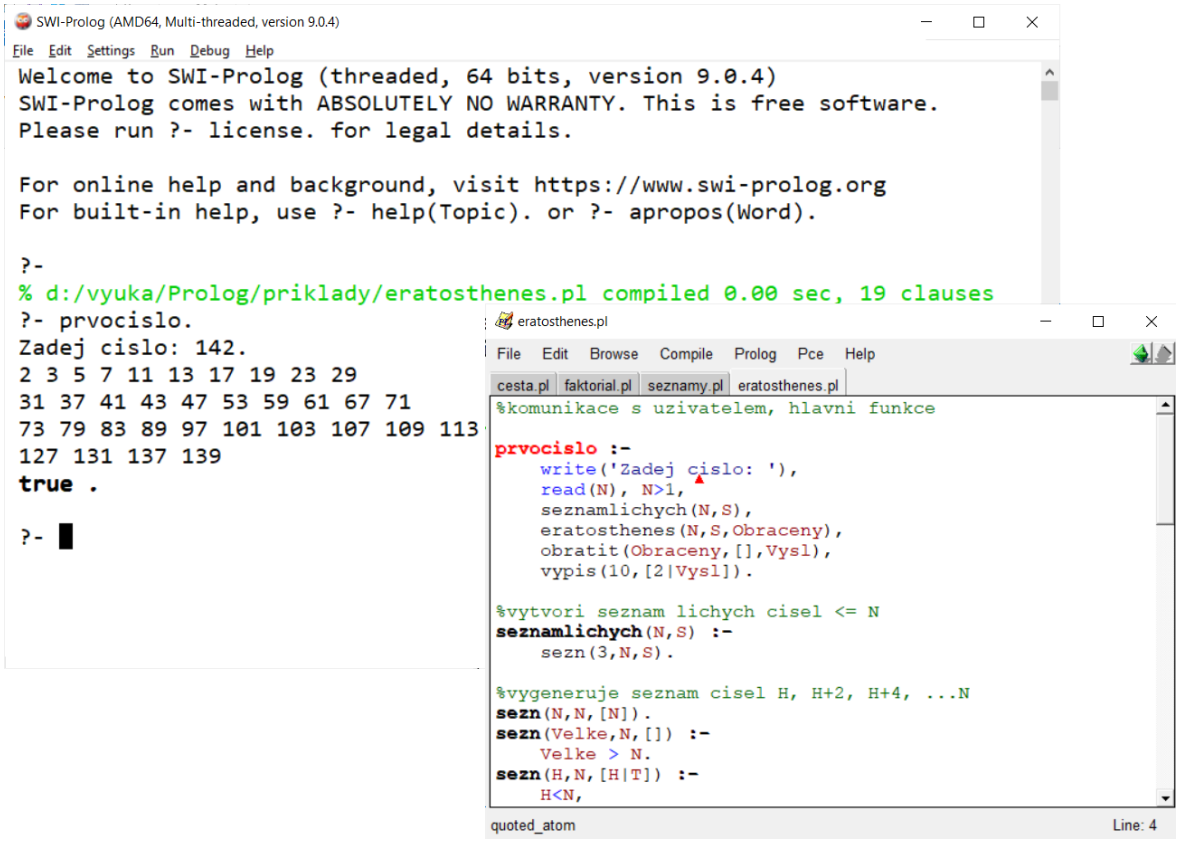
Prolog je jedním z jazyků pro logické programování. Vznikl ve Francii v roce 1973 (prof. A. Colmerauer) a jeho název je zkratka z francouzského PROgrammation à LOGique („programování v logice“). Je to deklarativní (neimperativní) jazyk. Většina překladačů je pouze interpretační nebo umožňují obojí, kompilace Prologu (GNU Prolog) obvykle znamená vytvoření jakéhosi mezikódu (například GNU Prolog generuje kód WAM – Warren Abstract Machine), který je pak přeložen na binární (spustitelný) soubor.

Existuje mnoho implementací Prologu. K nejznámějším patří SWI Prolog, GNU Prolog, LPA Win Prolog, Amzi! Prolog a další, zde budeme používat SWI Prolog šířený pod licencí GPL, případně jeho cloudovou variantu SWISH, která se nemusí instalovat.

Jednotlivé Prology se liší nejen licení, vzhledem a vybaveností svého editoru (většina Prologů má vlastní editor, se kterým je provázán), ale bohužel v některých případech také syntaxí programovacího jazyka. Rozdíly jsou například v práci se soubory.

Úkol

Pokud jste tak ještě neučinili, najděte si vhodnou implementaci Prologu. Můžete použít také online verzi SWI Prologu (SWISH), která je dostupná na https://swish.swi-prolog.org/.



Obrázek 1.1: SWI Prolog ve verzi 9.04 pro Windows

1.2.1 Zápis klauzulí v Prologu

Program v Prologu je vlastně báze znalostí, ze které Prolog odvozuje odpovědi na dotazy. Program je konečná neprázdná množina Hornových klauzulí, konkrétně jde o dva typy klauzulí:

- *pravidla* – obecná tvrzení ve tvaru „Závěr platí, pokud platí všechny jeho předpoklady zároveň."
- *fakty* – tvrzení bez předpokladů, je to obdoba toho, co jsme měli v předchozí kapitole jako speciální axiomy.

Používání programu spočívá v zadávání *dotazů* (cílových klauzulí) – Hornových klauzulí bez pozitivních literálů. Prolog dotazy vyhodnocuje podle programu a podle vnitřních pravidel.

	Klauzulární logika	**Predikátová klauzule**	**Zápis v Prologu**
Pravidlo	$B, C, D \rightarrow A$	$A \vee \neg B \vee \neg C \vee \neg D$	A :- B, C, D.
Fakt	$\rightarrow A$	A	A.
Dotaz	$B, C, D \rightarrow$	$\neg B \vee \neg C \vee \neg D$	?- B, C, D.

Tabulka 1.1: Zápis elementů v Prologu

Zápis jednotlivých elementů ukazuje tabulka 1.1. Každý element (příkaz, prologovskou klauzuli) vždy ukončíme tečkou. V pravidle rozlišujeme *tělo pravidla* (podle tabulky 1.1 to je B, C, D) a *hlavu pravidla* (A), tedy pravidlo je ve tvaru *hlava* :- *tělo*. Pro přehlednost se v delším pravidle cíl s oddělujícím dvojznakem zapisuje na samostatný řádek, tělo může být na více řádcích.

V případě dotazu dvojznak `?-` nezapisujeme, jde o prompt (výzvu) Prologu.

Příklad 1.1

Proměnné zapisujeme velkým počátečním písmenem, konstanty malým počátečním písmenem. Konstantou je i číslo nebo řetězec v uvozovkách či apostrofech. To před závorkou je predikát, v závorce jsou argumenty. Příklady faktů:

```
kocka(micka).     % Micka je kočka.
mys(jerry).      % Jerry je myš.
pes(zoubek).     % Zoubek je pes.
otec(jan, klara).   % Jan je otcem Kláry.
```

V pravidle píšeme nejdřív závěr a pak za dvojznakem `:-` předpoklady. Příklady pravidel:

```
lovi(Kdo, Koho) :-     % Kočky loví myši (když je „Kdo" kočka a „Koho" myš)
kocka(Kdo), mys(Koho).

prcha(Kdo, PredKym) :-      % Kočky prchají před psy.
kocka(Kdo), pes(PredKym).
```

Jako implikaci v predikátové logice bychom tato pravidla zapsali takto:

```
kocka(Kdo) & mys(Koho) -> lovi(Kdo, Koho)
kocka(Kdo) & pes(PredKym) -> prcha(Kdo, PredKym)
```

Pokud bychom se chtěli na něco dotázat, zápis bude následující:

```
prcha(micka, zoubek).      % Prchá Micka před Zoubkem?
prcha(X, zoubek).      % Kdo prchá před Zoubkem?
```

Pokud by nám bylo proti mysli, aby jména začínala malým písmenem, můžeme je napsat velkým, ale v uvozovkách či apostrofech (pozor, není to totéž, takže když se pro konkrétní zápis konstanty rozhodneme, musíme se ho držet). Takže pro Micku můžeme vybrat ze tří možností:

```
micka    "Micka"    'Micka'
```

Sestavený program se uloží do textového souboru s příponou .pl a konzultuje, tedy předá se překladači jazyka Prolog.

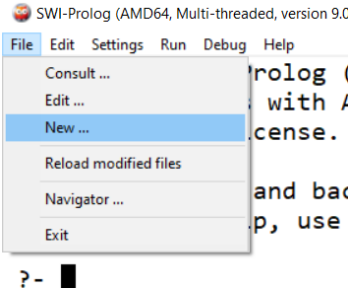
Příklad 1.2

Převedeme zadané klauzule klauzulární logiky na prologovské klauzule:

1. „Jahoda je červená."
   Klauzulární logika: $\rightarrow barva(jahoda, cervena)$
   Prolog:
   ```
   barva(jahoda,cervena).
   ```
2. „Ferda je dítě."
   Klauzulární logika: $\rightarrow dite(ferda)$
   Prolog:
   ```
   dite(ferda).
   ```
3. „Psi mají čtyři nohy."
   Klauzulární logika: $pes(X) \rightarrow pocet\_nohou(X, 4)$
   Prolog:
   ```
   pocet_nohou(X,4) :- pes(X).
   ```
4. „Školáci mají v létě prázdniny."
   Klauzulární logika: $skolak(X), obdobi(leto) \rightarrow ma\_prazdniny(X)$
   Prolog:
   ```
   ma_prazdniny(X) :-
     skolak(X),
     obdobi(leto).
   ```
5. „Děti mají rády sladká jídla."
   Klauzulární logika: $dite(X), jidlo(Y), chut(Y, sladky) \rightarrow ma\_rad(X, Y)$
   Prolog:
   ```
   ma_rad(X,Y) :-
     dite(X),
     jidlo(Y),
     chut(Y,sladky).
   ```

1.2.2 Ovládání aplikace SWI Prolog a webové aplipace SWISH

Nejdřív se zaměříme na (instalovanou) aplikaci. Po spuštění SWI Prologu se objeví konzola, což je okno, do kterého zadáváme dotazy (včetně požadavků na nápovědu nebo dotazů na dosud načtené klauzule).



Obrázek 1.2: Menu *File* ve SWI Prologu

V menu nás ze začátku budou zajímat především položky v části *File*, a to *New* (pro vytvoření nového programu), *Edit* (pro otevření existujícího programu v textovém editoru) a *Consult* (pro konzultování – načtení – programu do vnitřní databáze Prologu, jak vidíme na obrázku vpravo.

Vytvoření programu. Pokud zvolíme *New* nebo *Edit*, otevře se další okno, což je editor. Na obrázku 1.1 je vpravo dole. Jde o jednoduchý editor, který umí vysvicovat syntaxi Prologu. To

známe i z jiných vývojových prostředí, výhodou je, že na první pohled poznáme, co k čemu patří a co je případně chybné. Jednotlivé barvy mají tento význam:

- jasně červené jsou predikáty, které jsou použity pouze v klauzulích typu fakt nebo v hlavách klauzulí odpovídajících pravidlům,
- tučné černé jsou predikáty nacházející se v hlavách klauzulí, které se vyskytují i v tělech jiných klauzulí,
- netučné černé jsou predikáty v tělech klauzulí,
- modré jsou operátory a některé vestavěné predikáty (například write/1),
- červenohnědé jsou proměnné,
- tučné tmavé červené jsou chyby,
- zelené jsou komentáře (komentářovým symbolem je procento).

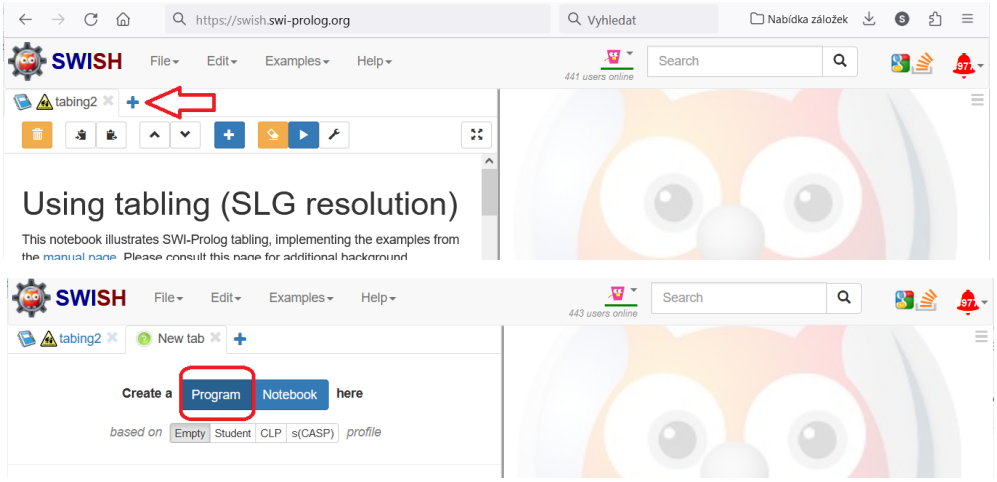
Proč Prolog zabarvuje predikáty v hlavách klauzulí (a ve faktech) některé jasně červenou a jiné černou barvou? Protože ty černé se v jiných klauzulích nacházejí na opačné straně implikace, a tedy je možné použít je jako „spojovací materiál“ při unifikaci a následné rezoluci, když z takových dvou klauzulí chceme něco odvodit. Atomy s predikáty zabarvenými jasně červenou barvou *nejsou chybné*, jen se Prologu moc nelíbí, že nebude možné využít je pro rezoluci s jinou klauzulí.

Načtení/konzultování programu. Pokud už máme program sestaven (například s využitím zmíněného editoru, nebo v jakémkoliv jiném textovém editoru), uložíme soubor s příponou .pl a pak v konzoli zvolíme v menu *File* → *Consult*, v běžném dialogovém okně najdeme soubor a potvrdíme. Alternativně můžeme v konzoli použít predikát `consult`:

```
consult("D:/Prolog/priklady/rodina.pl").
```

Tímto se soubor načte do interní databáze Prologu, Prolog si ho předzpracuje do binární podoby, aby se mu s klauzulemi lépe a rychleji pracovalo. Na výzvu Prologu (prompt, je to dvojznak `?-`, znamená „zadej dotaz“) zadáváme dotazy, Prolo vypisuje odpovědi.

Načtení (přeložení, konzultování) programu je nutné, protože Prolog si program udržuje v interním kódu (databázi) v binární podobě, se kterým se mu pracuje jednodušeji a především rychleji, navíc interní kód bývá obvykle bez syntaktických chyb (kontrola se provádí při načítání a sestavování interní reprezentace programu).



Obrázek 1.3: Prostředí webové aplikace SWISH

Při každé změně v souboru programu musíme (samozřejmě po uložení těchto změn) program znovu načíst, aby si Prolog mohl tento interní kód obnovit. Znovunačtení je jednodušší, nemusíme soubor hledat, stačí v konzoli napsat:

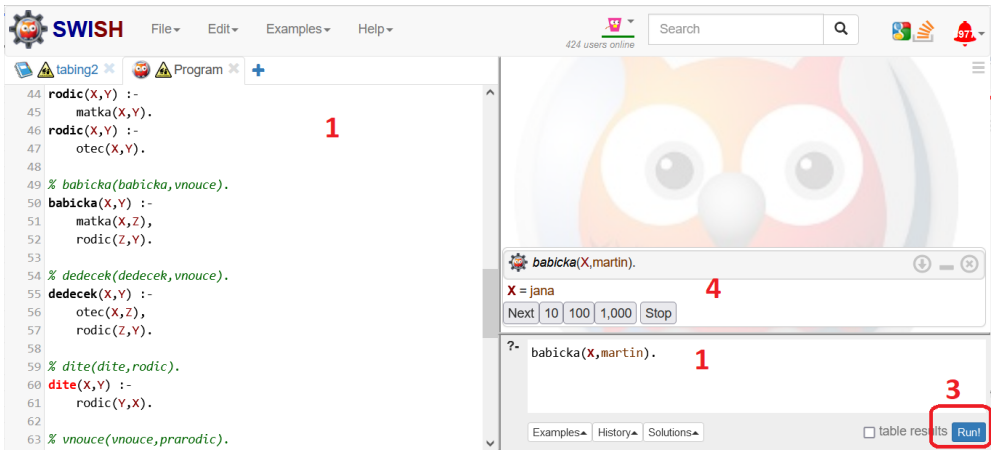
`make.`

(nezapomeňte tečku na konci).

V Prologu totiž obvykle vše končí tečkou: fakty, pravidla, dotazy, odpovědi. Všimněte si, že i dotaz (defacto příkaz) `make.` končí tečkou.

Zadávání dotazů. Dotazy píšeme do konzoly, každý dotaz taktéž končí tečkou. Pokud na dotaz Prolog vyhledává více než jednu odpověď, pak je vypisuje jednotlivě a po každé čeká na pobídnutí, jestli chceme další možnou odpověď: pokud má hledat další, stiskneme klávesu ;, pokud ne, tak cokoliv jiného, třeba tečku nebo Enter.

Používání online varianty SWISH. Na adrese https://swish.swi-prolog.org/ najdeme online variantu SWI Prologu. Obrázek 1.3 ukazuje prostředí po otevření odkazu a níže po otevření nové záložky pro vlastní program (můžeme mít otevřeno více záložek s různými programy), klepneme na tlačítko Program.



Obrázek 1.4: Program a dotazy ve SWISH

Na obrázku 1.4 je pak situace po napsání nebo vkopírování programu (bod 1). Zde není nutno konzultovat, program je ve stejném okně jako konzola. Zadáme dotaz (vpravo dole, bod 2) a klepneme na tlačítko Run (zcela dole, bod 3). Dotaz je vyhodnocen a výsledek se zobrazí o něco výše (bod 4).

Rozdíl oproti instalované aplikaci je hlavně v tom, že program se zde nekonzultuje a za dotazem se neklepe na Enter, ale na tlačítko Run, a odpovědi na dotaz se zobrazují nahoře, nikoliv pod dotazem. Jinak je to podobné.

Příklad 1.3

Sestavíme program obsahující zadané klauzule.

Zadání programu: Petr má rád květiny, Ivanu a televizi.
Jan má rád jitrnice a televizi.
Věra má ráda všechno, co má rád Jan.

V klauzulární logice:

$$\rightarrow ma\_rad(petr, kvetiny)$$
$$\rightarrow ma\_rad(petr, ivana)$$
$$\rightarrow ma\_rad(petr, televize)$$
$$\rightarrow ma\_rad(jan, jitrnice)$$
$$\rightarrow ma\_rad(jan, televize)$$
$$ma\_rad(jan, X) \rightarrow ma\_rad(vera, X)$$

Program v Prologu:

```
ma_rad(petr,kvetiny).
ma_rad(petr,ivana).
ma_rad(petr,televize).
ma_rad(jan,jitrnice).
ma_rad(jan,televize).
ma_rad(vera,X) :- ma_rad(jan,X).
```

Program, který jsme takto vytvořili, uložíme do souboru s příponou PL. Tento soubor pak načteme (konzultujeme) do Prologu a můžeme zadávat dotazy.

Dotazy mohou obsahovat jeden nebo více atomů, argumenty predikátů mohou být i proměnné. Pokud jsou všechny atomy dotazu bázové, Prolog odpoví pouze `yes` nebo `no`, podle toho, zda klauzule dotazu vyplývá z programu (znalostní báze), jestliže však jsou použity proměnné, Prolog vypíše postupně všechny možné hodnoty proměnné (kombinace proměnných), pro které je dotaz splnitelný.

Příklad 1.4

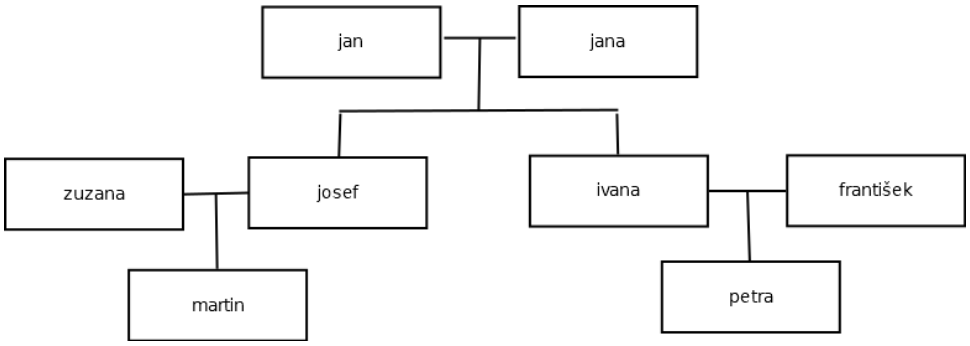
Po načtení programu z předchozího příkladu zadáme následující dotazy a získáme uvedené odpovědi.

```
?- ma_rad(petr,televize).                    Má rád Petr televizi?
    yes
?- ma_rad(jan,kvetiny).                      Má rád Jan květiny?
    no
?- ma_rad(jan,X).                            Co má rád Jan?
    X = jitrnice ;
    X = televize ;
    no
?- ma_rad(X,jitrnice).                       Kdo má rád jitrnice?
    X = jan ;
    X = vera ;
    no
?- ma_rad(petr,X),ma_rad(jan,X)              Co má rád Petr a zároveň Jan?
    X = televize ;
    no
?- ma_rad(Kdo,Co).                           Kdo má co (koho) rád?
    Kdo = petr , Co = kvetiny ;
    Kdo = petr , Co = ivana ;
    Kdo = petr , Co = televize ;
    Kdo = jan , Co = jitrnice ;
    Kdo = jan , Co = televize ;
    Kdo = vera , Co = jitrnice ;
```

```
    Kdo = vera , Co = televize ;
    no
```

Příklad 1.5

Napíšeme program v Prologu, který ve faktech a pravidlech zachytí rodinnou strukturu, která je znázorněna na obrázku 1.5 (část klauzulí v dále sestaveném programu chybí).



Obrázek 1.5: Rodinná struktura pro příklad

Nejdřív pomocí predikátů `muz/1` a `zena/1` vytvoříme fakty o tom, kdo je muž a kdo žena:

- jan, josef, František a martin jsou muži,
- jana, zuzana, ivana a petra jsou ženy.

```
muz(jan).
zena(jana).
...
```

Připíšeme predikát `manzele/2`, který určí, kdo jsou manželé. Pozor, musí fungovat „obousměrně“, tedy nesmí záležet na tom, jestli jako první parametr uvedeme ženu nebo muže. Rekurzi se vyhneme použitím pomocného predikátu (třeba `manzelepom/2`).

```
manzelepom(jan,jana).
manzelepom(josef,zuzana).
manzelepom(ivana,frantisek).
manzele(X,Y) :-
    manzelepom(X,Y).
manzele(X,Y) :-
    manzelepom(Y,X).
```

Následně dodáme pravidla určující predikáty `manzelka/2` a `manzel/2`.

```
% manzelka(zena,muz).
manzelka(Zena,Muz) :-
    manzele(Zena,Muz),
    zena(Zena),
    muz(Muz).

% manzel(muz,zena).
manzel(Muz,Zena) :-
    manzele(Muz,Zena),
    muz(Muz),
    zena(Zena).
```

Vytvoříme množinu faktů určující, kdo je čí matkou a otcem, a obecně rodičem a dítětem:

```
% matka(matka,dite).
matka(jana,josef).
matka(jana,ivana).
matka(zuzana,martin).
matka(ivana,petra).

% otec(otec,dite).
otec(X,Y) :-
    manzel(X,Z),
    matka(Z,Y).

%rodic(rodic,dite).
rodic(X,Y) :-
    matka(X,Y).
rodic(X,Y) :-
    otec(X,Y).

% dite(dite,rodic).
dite(X,Y) :-
    rodic(Y,X).
```

Další bude babička, dědeček, vnuk, vnučka a sourozenec, přičemž musíme zajistit, aby nikdo nebyl sourozencem sám sobě:

```
% babicka(babicka,vnouce).
babicka(X,Y) :-
    matka(X,Z),
    rodic(Z,Y).

% dedecek(dedecek,vnouce).
dedecek(X,Y) :-
    otec(X,Z),
    rodic(Z,Y).

vnuk(X,Y) :-
    muz(X),
    babicka(Y,X).
vnucka(X,Y) :-
    zena(X),
    dedecek(Y,X).
vnucka(X,Y) :-
    zena(X),
    babicka(Y,X).
vnouce(X,Y) :-
    vnuk(X,Y).
vnouce(X,Y) :-
    vnucka(X,Y).

% sourozenec(X,Y).
sourozenec(X,Y) :-
    matka(Z,X),
    matka(Z,Y),
    X\=Y.
```

Program konzultujeme přes *File* a pak se můžeme dotazovat. Položíme následující dotazy:

- Jsou Zuzana a Josef manželé?
- Jsou Jan a Ivana manželé?
- Kdo je Martinův dědeček?
- Jak se jmenují Janova vnoučata?

Celá dotazovací komunikace bude vypadat takto (píšeme to, co je **za promptem** „`?-`“):

```
?- manzele(zuzana,josef).
true.

?- manzele(jan,ivana).
false.

?- dedecek(X,martin).
X = jan ;
false.

?- vnouce(X,jan).
X = martin ;
X = petra ;
false.
```

Zatím jsme v parametrech predikátů používali buď konstanty (začínající malým písmenem) nebo proměnné (začínající velkým písmenem). Do proměnné Prolog postupně doplňuje a vypisuje nalezené možnosti. Ale co když nás ty možnosti ve skutečnosti nezajímají? Můžeme se zeptat třeba takto:

- Má Jan nějaká vnoučata?
- Je Jana něčí babičkou?

V dotazech pak nepoužijeme běžnou proměnnou, ale místo ní tzv. anonymní proměnnou. Anonymní proměnná je vázaná existenčně (v predikátové logice bychom před ní použili symbol zavináče). Tady používáme podtržítko:

```
?- vnouce(_,jan).
true .

?- babicka(jana,_).
true .
```

Úkol

Podle následujících vět sestavte soubor s příponou .PL s klauzulemi:

- Hektor a Lea jsou lvi, Zulejda je zajíc.
- Lvi jsou silní a velcí, mají žlutou barvu.
- Zajíci jsou rychlí a mají hnědou barvu.
- Kdo je silný a velký, je králem zvířat.

Dbejte na to, aby opravdu šlo o Hornovy klauzule a nezapomeňte, v jakém pořadí je antecedent/konsekvent v prologovské klauzuli. Nezapomeňte, že konstanty začínají malým písmenem.

.PL soubor uložte a konzultujte (načtěte) do některého Prologu. Dále zadávejte dotazy:

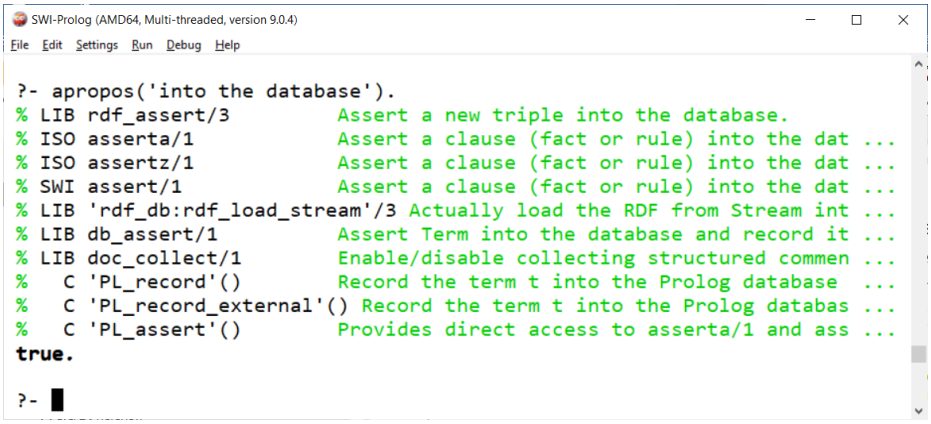
- Je Hektor lev?
- Je Zulejda lev?
- Je Lea králem zvířat?
- Kdo je lev? (tj. chceme, aby Prolog vypsal všechny lvy)
- Kdo je rychlý?
- Kdo je hnědý?
- Kdo má jakou barvu? (tj. chceme uspořádané dvojice [*jméno*, *barva*])

1.2.3 Nápověda

Nápovědu k vestavěným predikátům a dalším mechanismům získáme příkazem `help`, případně pokud neznáme název predikátu, ale známe klíčové slovo z jeho popisu, tak `apropos` (pracujeme v hlavním okně Prologu, tedy tam, kde zadáváme i dotazy).

```
apropos('into the database').
```

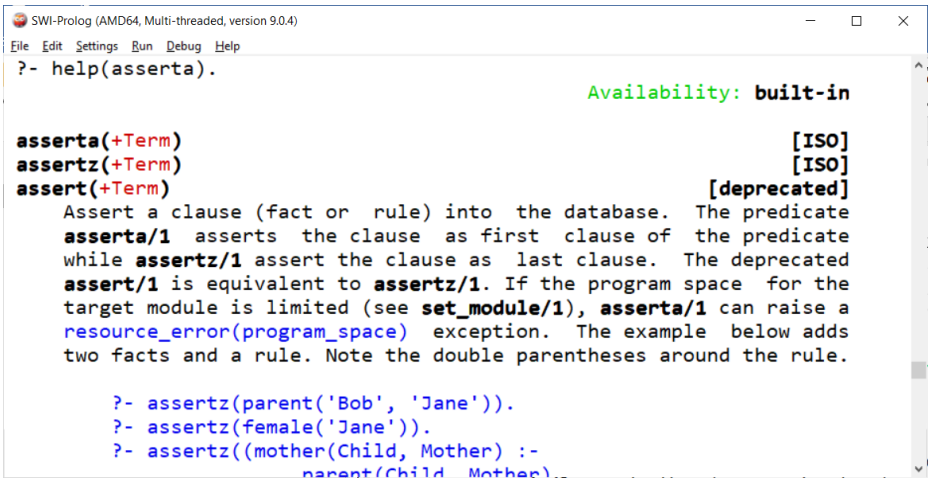
Zadaný dotaz žádá o seznam vestavěných predikátů, které ve svém popisu v nápovědě mají vkládání faktů a pravidel do databáze. Na obrázku 1.6 je výstup, kde chceme zjistit, které predikáty (příkazy) slouží k přístupu do databáze.



Obrázek 1.6: Nápověda pro případ, že hledám název příkazu

Zajímá nás druhý a třetí nalezený řádek, jejichž popis začíná stejně, tedy použijeme příkaz help, abychom se dozvěděli podrobnosti:

```
help(asseta).
help(assetz).
```



Obrázek 1.7: Nápověda pro konkrétní predikát/příkaz

Vlastně bude stačit jeden z nich, protože oba tyto predikáty mají společnou nápovědu:

Takže je jasné, že takto můžeme vkládat do báze ručně další klauzule, aniž bychom je museli vepisovat do programu. Zatímco `asserta/1` vloží novou klauzuli před již načtené klauzule pro daný predikát, `assertz/1` vloží novou klauzuli za již načtené klauzule pro daný predikát. Tyto vestavěné predikáty používáme v dotazovacím prostředí, pokud se dodatečně rozhodneme načtenou bázi obohatit.

Úkol

Zjistěte pomocí příkazu `apropos/1`, jestli existuje funkce (predikát) zjišťující podřetězec z daného řetězce (podřetězec se anglicky řekne substring). Dále zjistěte, jestli existuje funkce (predikát) zřetězující, tedy spojující řetězce (zřetězit se anglicky řekne concatenate).

V nápovědě (či referenční příručce a různých manuálech) se u argumentů predikátů či termů používá toto značení – zajímá nás symbol před označením argumentu:

- `+argument` musí být instanciovaný, tedy mít už předem přiřazenu nějakou hodnotu
- `-argument` zde má být proměnná
- `?argument` instanciovaný parametr nebo proměnná
- `@argument` parametr nebude vázán unifikací
- `:argument` jedná se o název predikátu

Příklad 1.6

V předchozím úkolu jsme měli zjistit, jak se nazývá predikát pro nalezení podřetězce. Pokud jsme postupovali správně, zjistili jsme, že to je predikát `sub_string/5` nebo `sub_string/3` (a pár dalších možností). Zobrazíme si k němu nápovědu:

```
?- help(sub_string).
sub_string(+Table, +Sub, +String)
    Succeeds if Sub is a substring of String using the named Table.
```

```
                                                    Availability: built-in

sub_string(+String, ?Before, ?Length, ?After, ?SubString)
    This predicate is functionally equivalent to sub_atom/5, but operates on
    strings. Note that this implies the string input arguments can be either
    strings or atoms. If SubString is unbound (output) it is unified with a
    string. The following example splits a string of the form <name>=<value>
    into the name part (an atom) and the value (a string).

        name_value(String, Name, Value) :-
            sub_string(String, Before, _, After, "="),
            !,
            sub_atom(String, 0, Before, _, Name),
            sub_string(String, _, After, 0, Value).
```

Jak vidíme, před každým argumentem v závorkách je některý z výše zmíněných znaků. Tento predikát můžeme v dotazu využít třeba takto:

```
?- sub_string("zadanretezec123", 5, 7, Zbytek, Sub).
Zbytek = 3,
Sub = "retezec".
```

První argument je konstantní řetězec v uvozovkách, může být i bez uvozovek (protože začíná malým písmenem a neobsahuje mezery) nebo v apostrofech – fungovalo by to stejně. Musí jít o konstantu nebo unifikovanou proměnnou, protože v předpisu máme před tímto argumentem znak +.

Druhý argument je pozice, od které chceme získat podřetězec (pozor, počítá se od nuly), následuje počet požadovaných znaků podřetězce, potom kolik má zbývat za podřetězcem, poslední parametr je ten podřetězec. Za poslední dva parametry jsme dosadili proměnnou, což můžeme, to odpovídá symbolu ?.

Kdyby nás nezajímal předposlední parametr, mohli bychom použít anonymní proměnnou:

```
?- sub_string("zadanretezec123", 5, 7, _, Sub).
Sub = "retezec".
```

Ale tento predikát může být použit i jinak než na zjištění podřetězce na dané pozici a v dané délce. Například můžeme vyhledat pozici a délku zadaného podřetězce:

```
?- sub_string("zadanretezec123", Poz, Delka, _, "adan").
Poz = 1,
Delka = 4
```

V tomto případě by byla irelevantní i délka, tedy místo proměnné Delka bychom klidně mohli dát také anonymní proměnnou.

Pokud bychom chtěli najít všechny výskyty určitého podřetězce v daném řetězci, můžeme predikát použít takto:

```
?- sub_string("vstup123 x2123yt", Poz, _, _, "123").
Poz = 5 ;
Poz = 11 ;
false.
```

Prolog najde postupně všechny výskyty, pokud po každém nalezeném budeme poctivě zadávat středník. Když už žádný další nenajde, vypíše false.

Úkol

Vypište si nápovědu k predikátu `between/3`. Zjistěte, jak se dá používat, kde lze použít proměnnou a kde musí být instanciovaná hodnota, vyzkoušejte různé varianty.

1.2.4 Základní práce s databází

V Prologu máme i „manažerské“ nástroje – speciální predikáty či příkazy sloužící pro práci s interní databází. Již dříve byl zmíněn postup konzultování programu přes menu, totéž se dá provést pomocí predikátu `consult/1`, kterému jako parametr dodáme název souboru, ideálně s celou cestou k němu.

Protože obvykle s tímtéž zdrojovým souborem pracujeme opakovaně (hlavně když ladíme program) a potřebujeme ho často načítat, může se hodit také predikát `make/0`, který jednoduše znovu konzultuje veškeré programy, které zatím byly konzultovány. Stačí napsat:

```
make.
```

Dále se nám může hodit například predikát `listing/1` – jako parametr uvedeme predikát, jehož definici chceme vypsat. Například podle předchozího příkladu můžeme napsat:

```
listing(manzele).
```

Tím zjistíme definici predikátu `manzele/2`.

1.2.5 Anonymní proměnná

Existenční termy v Prologu nepoužíváme, místo nich máme k dispozici *anonymní proměnnou* (zapisuje se znakem podtržítka). Pro argument, ve kterém je použita, existuje hodnota, kterou tam lze dosadit, ale tato hodnota nás nezajímá (kdyby nás zajímala, pak bychom použili proměnnou).

Příklad 1.7

Máme následující program:

```
lovi(liska,zajic).                    Liška loví zajíce.
lovi(orel,mys).                       Orel loví myš.
lovi(orel,vrabec).                    Orel loví vrabce.
lovi(honza,ryba).                     Honza loví rybu.
dravec(X) :- lovi(X,_).               Kdo někoho loví, je dravec.
```

Budeme zadávat dotazy:

```
?- dravec(_).                         Existují nějací dravci?
   yes
?- lovi(liska,_).                     Loví někoho liška?
   yes
?- lovi(X,_).                         Kdo někoho loví?
   X = liska ;
   X = orel ;
   X = honza ;
   no
```

Anonymní proměnnou také použijeme místo „běžné“ proměnné, pokud se tato proměnná vyskytuje v těle pravidla pouze jednou.

Obecně platí, že anonymní proměnná *začíná* symbolem podtržítka. To znamená, že za podtržítkem může následovat řetězec písmen a číslic. Pokud do Prologu přepisujeme klauzuli, ve které se některá existenční konstanta vyskytuje více než jednou, měli bychom za podtržítko přidat ještě identifikační řetězec, který zajistí vazbu mezi těmito různými výskyty.

Příklad 1.8

V úkolu na straně 10 jsme vytvořili .PL soubor s klauzulemi o dvou lvech a jednom zajíci. Podle této báze je král zvířat každý, kdo je zároveň silný a velký. Do báze přidáme tyto dvě klauzule:

```
existuje_rychly_kral1 :- rychly(_),kral_zvirat(_).
existuje_rychly_kral2 :- rychly(_1),kral_zvirat(_1).
```

Po uložení a rekonzultování zadáme dotaz „Existuje rychlý král?“:

1. Nejdřív použijeme první přidaný predikát:
   ```
   ?- existuje_rychly_kral1.
       yes
   ```
   Ovšem tato odpověď ve skutečnosti není správná. V definici predikátu jsme použili dvě anonymní proměnné a nedefinovali jsme mezi nimi žádnou vazbu, tedy Prolog pouze zjistil, zda existují hodnoty, které lze na tato místa dosadit, považoval je za různé proměnné.
2. Použijeme druhý přidaný predikát:
   ```
   ?- existuje_rychly_kral2.
       no
   ```
   Dostali jsme správnou odpověď, protože přidáním „`1`“ za podtržítko jsme určili vazbu a Prolog do obou míst dosazuje tutéž hodnotu.

Úkol

Vytvořte soubor s bází prologovských klauzulí podle těchto vět:

- Pepa jedl mrkev, Jana jedla zákusek, Pepa pil pivo, Honza pil vodu.
- Každý, kdo něco jedl, je sytý.
- Vše, co někdo jedl, je snědeno.
- Vše, co někdo pil, je vypito.
- Co bylo snědeno *nebo* vypito, se musí koupit.
- Každý, kdo něco jedl *a* pil (obojí zároveň), odchází.

Použijte následující predikáty:

`jedl(`$\langle kdo, co\rangle$`)`	`snedeno(`$\langle co\rangle$`)`	`koupit(`$\langle co\rangle$`)`
`pil(`$\langle kdo, co\rangle$`)`	`vypito(`$\langle co\rangle$`)`	`odchazi(`$\langle kdo\rangle$`)`
`syty(`$\langle kdo\rangle$`)`		

Uložte, konzultujte do Prologu a položte následující dotazy (s využitím stejných predikátů):

- Kdo je sytý?

- Je někdo systý? (chceme odpověď Yes/No)
- Co je třeba koupit?
- Je třeba něco koupit? (chceme odpověď Yes/No)
- Kdo zároveň jedl i pil?
- Kdo odchází?
- Odchází někdo?

1.3 Průběh výpočtu v Prologu

Prolog je deklarativní jazyk, tedy programátor určuje, *co* se má provést a ne *jak* se to má provést a kam ukládat mezivýsledky výpočtu. Data a program splývají, nerozlišují se. Řízení výpočtu je tedy na Prologu samotném, my mu pouze sdělíme, co má zjistit (odvodit, dokázat, vypsat).

Přesto je užitečné mít alespoň základní přehled o tom, jak výpočet probíhá, a to proto, abychom se dokázali vyhnout zbytečné, třeba i nekonečné rekurzi, optimalizovat program, a také co nejlépe využít prostředky, které nám jazyk nabízí.

1.3.1 Rezoluce a výpočetní strom

Prolog používá nepřímou rezoluci k odvozování důsledků z báze (kterou dostane od uživatele coby program), samozřejmě s unifikací. Protože jde o nepřímou rezoluci, úspěch (nalezení řešení) pro daný dotaz (který taktéž obdržel od uživatele) znamená, že Prolog dospěl k prázdné klauzuli – klauzuli s významem „false“.

 Průběh výpočtu zde nebudeme podrobně probírat, je to téma pro předmět Logika a logické programování. Na postup se podíváme podrobněji v následující sekci, zde jen stručně:

1. Prolog zneguje zadaný dotaz, získá *cílovou klauzuli pro první krok*
2. začne tvořit výpočetní strom pro daný dotaz takto:
   - vyhledá v bázi takovou klauzuli, která se dá unifikovat s cílovou klauzulí, tedy jejíž hlava obsahuje stejný predikát jako první atom cílové klauzule (uvědomte si, že hlavy klauzulí jsou vlastně konsekventy, tedy části „za šipkou“, kdežto negovaný dotaz má všechny atomy v antecedentu, tedy „před šipkou“)
   - Prolog provede unifikaci (použije vhodnou substituci) a následně použije rezoluční pravidlo
   - výsledkem je *cílová klauzule pro další krok*
   - pokud je takových klauzulí pro unifikaci v bázi více, výpočetní strom se rozdělí do větví pro různé nalezené klauzule, v jednotlivých větvích jsou různá řešení
   - pokud při unifikaci lze dosadit různé možnosti, opět to znamená větvení výpočetního stromu a možnost dojít k různým řešením
3. popsaný krok výpočtu provádí Prolog tak dlouho, dokud nedojde na konec větve
4. na konci větve vypíše výsledek (true/false, příp. hodnoty proměnných získané během unifikací cestou po větvi)

Výpočetní strom se sice větví (při více použitelných klauzulích pro unifikaci a při více hodnotách, které se v substituci dají dosadit), ale Prolog je negeneruje (neprohledává) paralelně. Používá se metoda prohledávání do hloubky, tedy když vyřeší první větev – dojde na její konec, pak teprve se navrací a prochází další větev, atd.

Definice 1.1 (Lineární výpočetní strom)

Lineární výpočetní strom klauzule pro znalostní bázi je takový strom, kde:

- všechny uzly jsou ohodnoceny klauzulemi,
- kořen je ohodnocen cílovou klauzulí,
- pro všechny uzly stromu platí: jestliže je uzel ohodnocen klauzulí C, pak každý jeho potomek je ohodnocen klauzulí vzniklou uplatněním rezolučního odvozovacího pravidla na klauzuli C a některou klauzuli znalostní báze.

Příklad 1.9

Vytvoříme lineární výpočetní strom pro nepřímý důkaz důkaz klauzule $D \rightarrow$ ze znalostní báze

1. $A, B \rightarrow C$
2. $D \rightarrow B$
3. $\rightarrow A$
4. $C \rightarrow$

Při konstrukci nepřímého důkazu budeme vždy rezoluční odvozovací pravidlo používat na poslední klauzuli, kterou jsme přidali k posloupnosti důkazu (v prvním kroku to je klauzule popírající množiny) a některou další klauzuli, tedy budeme postupovat *lineární metodou*.

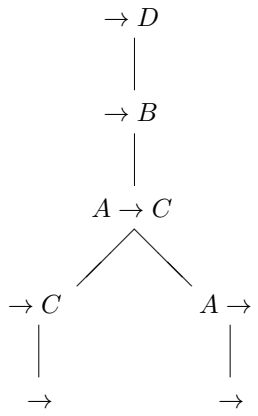
V tomto příkladu existují dvě takové posloupnosti důkazu:

5.	$\rightarrow D$	PM	5.	$\rightarrow D$	PM
6.	$\rightarrow B$	R(2,5)	6.	$\rightarrow B$	R(2,5)
7.	$A \rightarrow C$	R(1,6)	7.	$A \rightarrow C$	R(1,6)
8.	$\rightarrow C$	R(3,7)	8.	$A \rightarrow$	R(4,7)
9.	$\rightarrow$	R(4,8)	9.	$\rightarrow$	R(3,8)

Výpočetní strom konstruujeme tak, že kořen stromu ohodnotíme první klauzulí popírající množiny a dále podle každé vytvořené důkazní posloupnosti vytvoříme větev stromu. Strom pro uvedený příklad je na obrázku 1.8.

Programovací jazyky pro logické programování obvykle používají lineární metodu s prohledáváním do hloubky. Nebezpečí zacyklení výpočtu lze pak předejít vhodným pořadím klauzulí ve znalostní bázi a pořadím atomů v jednotlivých klauzulích. Zacyklení většinou předejdeme, pokud dodržujeme tato pravidla:

1. Ve znalostní bázi *nejdřív uvádíme fakty, pak pravidla*. Protože překladač při prohledávání báze postupuje shora dolů, docílíme tím používání takových unifikací klauzulí pro rezoluci, při kterých nedojde ke zbytečnému opakování výpočtu a tím někdy i k rekurzivnímu vyhodnocování klauzulí.



Obrázek 1.8: Výpočetní strom nepřímého důkazu klauzule

2. Jestliže je v těle klauzule (v antecedentu) *atom se stejným predikátem jako atom v hlavě klauzule*, třeba i s jinými argumenty, pak takový atom *umístíme až na konec těla klauzule*. Opět tím zamezíme nadbytečnému rekurzivnímu volání klauzule.

Zopakujme si nyní všechna pravidla, která bychom měli dodržovat při sestavování znalostní báze (programu v Prologu):

- předem si promyslíme názvy predikátů a konstant tak, aby byly čitelné pro uživatele, případně můžeme přidávat komentáře,
- pokud se proměnná vyskytuje v klauzuli pouze jednou, použijeme anonymní proměnnou,
- funktory používáme jen tehdy, když je to opravdu nutné a bereme na vědomí, že funktor lze používat spíše jen jako argument predikátu,
- v bázi uvedeme nejdřív fakty a pak pravidla, zvláště v případě, že některá pravidla mají v hlavě shodný predikát s příslušným faktem,
- jestliže se v pravidle vyskytuje rekurze, toto pravidlo uvedeme jako poslední ze všech klauzulí, které mají v hlavě tentýž predikát,
- pokud je v klauzuli atom negovaný predikátem `not`, pak tento atom uvádíme v klauzuli jako poslední,
- můžeme si také všímat nepřímé rekurze, nad tou se zamyslíme v následujícím úkolu.

Tento seznam rozhodně není vyčerpávající. Při používání pokročilejších programovacích technik například můžeme predikáty navrhovat tak, aby jejich argumenty mohly být zároveň vstupní i výstupní, apod. Mnohé nedostatky také zjistíme při ladění programu, kdy do dotazů dosazujeme různé možné hodnoty.

Úkol

Je dána znalostní báze šesti klauzulí klauzulární logiky:

1. $A \rightarrow B$
2. $E \rightarrow D$
3. $B, C \rightarrow D$
4. $\rightarrow C$
5. $\rightarrow A$
6. $A, D \rightarrow E$

Napište dvě různé posloupnosti nepřímého důkazu pro tvrzení $\rightarrow D$ (jde o nepřímý důkaz, nezapomeňte toto tvrzení předem negovat).

Dále vytvořte výpočetní strom pro toto tvrzení s délkami větví nejvýše 5. Zjistěte, která větev je rekurzivní a které klauzule rekurzi (nepřímou) způsobují. Zamyslete se nad tím, jak by bylo vhodné uspořádat klauzule v bázi, aby díky této větvi nedošlo k zacyklení už při hledání prvního řešení (používáme prohledávání do hloubky).

1.3.2 Podrobněji k výpočtu v Prologu

Prolog při uplatňování rezoluce používá *lineární metodu s prohledáváním do hloubky*, která byla popsána v předchozí sekci. Aby bylo možné používat rezoluční odvozovací pravidlo, musí být obvykle klauzule upraveny substitucí – unifikací. Pro unifikaci je použit algoritmus podobný algoritmu pro hledání nejobecnějšího unifikátoru.

Informace o použitých unifikačních substitucích se ukládají. Protože proměnné v Prologu jsou lokální pro danou klauzuli (dvě stejně pojmenované proměnné v různých klauzulích jsou ve skutečnosti různé proměnné) a globální proměnné neexistují, musí být v údaji o unifikaci explicitně odlišeny stejně pojmenované proměnné z různých klauzulí. Prolog toto zajišťuje připojením čísla klauzule k proměnné, a tím je odstraněno nebezpečí kolize.

Aniž si to většinou uvědomujeme, zadáváme dotaz v negovaném tvaru. Pokud jsou v dotazu použity proměnné, v původním (nenegovaném) tvaru jsou ve skutečnosti vázány existenčně, tedy ptáme se, zda existují nějaké hodnoty proměnných takové, že platí formule dotazu. V predikátové logice můžeme negovaný dotaz vyjádřit takto:

$$\begin{aligned} & \neg\exists u_1 \exists u_2 \ldots \exists u_r (A_1 \,\&\, A_2 \,\&\, \ldots \,\&\, A_p) \\ \Leftrightarrow \quad & \forall u_1 \forall u_2 \ldots \forall u_r (\neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_p) \\ \Leftrightarrow \quad & \forall u_1 \forall u_2 \ldots \forall u_r (true \rightarrow (\neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_p)) \end{aligned}$$

Poslední uvedená formule se do klauzulární logiky přepisuje jako

$$A_1, A_2, \ldots, A_p \rightarrow \tag{1.1}$$

Všechny proměnné u_i jsou vázány univerzálně, proto lze na klauzuli uplatňovat rezoluční pravidlo. Pokud jsme na některých místech zadali anonymní proměnné, je s nimi zacházeno jako s volnými proměnnými.

Samotný výpočet je rekurzivní proces, který se provádí tak dlouho, dokud je co počítat (u zacykleného výpočtu teoreticky i do nekonečna, prakticky se výpočet zastaví s chybovým hlášením o přetečení zásobníku).

V každém kroku zpracováváme klauzuli, kterou nazýváme *cílová klauzule*, její atomy nazýváme *cíle*, a hledáme klauzuli takovou, aby bylo možné na ni a na cílovou klauzuli uplatnit pravidlo rezoluce. Když se nám to podaří, rezolventa (výsledek uplatnění pravidla rezoluce) se stává novou cílovou klauzulí pro další krok výpočtu. Z toho, že v programu jsou pouze Hornovy klauzule (v konsekventu je nejvýše jeden atom), vyplývá, že všechny cílové klauzule, které během výpočtu získáváme, mají prázdný konsekvent (jsou to prologovské klauzule bez hlavy).

Při výpočtu používáme *zásobník*, do kterého při každém použití unifikace a rezoluce ukládáme údaje o této operaci. Uložíme vždy číslo klauzule, na kterou bylo spolu s cílovou klauzulí

uplatněno pravidlo rezoluce, a údaje o unifikační substituci použité pro přípravu na rezoluci, tedy údajem je uspořádaná dvojice $[i, \varphi]$, kde i je číslo klauzule a φ je unifikace.

Údaje se ze zásobníku vyjímají při každém ukončení výpočtu větve (ať úspěšném – `yes` nebo neúspěšném – `no`). Když byl tento údaj do zásobníku uložen, byla unifikace φ a rezoluce použita na klauzuli s číslem i. V případě úspěchu jsme již cíl, ke kterému se takto dalo dostat, zpracovali a potřebujeme najít další cíl, v případě neúspěchu tato cesta zklamala a potřebujeme najít další cestu ke splnění cíle, proto budeme pokračovat následující klauzulí (číslo $i + 1$) s tím, že jako cílovou klauzuli budeme mít tu, která byla cílovou klauzulí před krokem určeným údajem $[i, \varphi]$. Tato klauzule se dá zjistit „zpětným provedením" operací daných těmito údaji.

Vyjmutí údajů ze zásobníku je vlastně navracení se k předchozímu cíli, tato operace se nazývá *navracení (backtracking)*.

Postup (Výpočet v Prologu)

Protože pracujeme s Hornovými klauzulemi a navíc výpočet začíná u dotazu, který má všechny atomy v antecedentu, je *algoritmus postupu výpočtu* poměrně jednoduchý:

1. Na začátku výpočtu se cílovou klauzulí stane (negovaný) dotaz. Prvním cílem je nejlevější atom této klauzule. Od bodu 2 následuje rekurzivní algoritmus zpracování cílové klauzule.

2. Pokud cílová klauzule není prázdná klauzule, tento bod přeskočíme a pokračujeme bodem 3. Jestliže cílovou klauzulí je prázdná klauzule, *končíme výpočet větve s úspěchem* (`yes`). Jsou dvě možnosti:

   (a) V dotazu jsou proměnné: vypíšeme nalezené hodnoty těchto proměnných (poslední hodnoty ze zásobníku příslušející těmto proměnným), čekáme na stisk klávesy a pokud je to klávesa ;, provedeme navracení a pokračujeme bodem 3.

   (b) V dotazu nejsou proměnné: vypíšeme `yes`, ukončíme celý výpočet a smažeme obsah zásobníku.

3. Vezmeme nejlevější atom (cíl) cílové klauzule a hledáme v programu klauzuli, která

   - ještě pro tento cíl nebyla použita,
   - má ve své hlavě (tj. v konsekventu) tentýž predikát jako testovaný cíl
   - a je možné provést unifikaci přes atom v hlavě klauzule a testovaný cíl cílové klauzule.

   Jsou tři možnosti:

   (a) Takovou klauzuli najdeme: pokračujeme bodem 4.

   (b) Takovou klauzuli se nepodaří najít a zásobník není prázdný: provedeme navracení (vše, co bylo až do této pozice v programu provedeno, zrušíme a zkoušíme další cestu) a pokračujeme bodem 3.

   (c) Takovou klauzuli se nepodaří najít a zásobník je prázdný: nelze pokračovat jinak, než jak se dosud postupovalo (tj. zásobník je prázdný, ale cílová klauzule je neprázdná), *končíme výpočet s neúspěchem* (vypíšeme `no`).

4. Unifikujeme cílovou klauzuli a nalezenou klauzuli, uplatníme pravidlo rezoluce a rezolventu (výsledek rezoluce) použijeme jako *novou cílovou klauzuli*. Je zřejmé, že nejlevější cíl,

který jsme zpracovávali v původní cílové klauzuli, se v nové cílové klauzuli neobjeví, je „odříznut“.

Do zásobníku je uloženo číslo klauzule, která je unifikována s cílem, a údaje o použité substituci. Pokračujeme bodem 2.

Příklad 1.10

Odvození odpovědi na dotaz „Jde Pepa do restaurace?“ z uvedeného programu provedeme nejdřív v klauzulární logice a pak v Prologu.

V klauzulární logice:

1. $\rightarrow turista(pepa)$ SA_1
2. $\rightarrow cestuje(pepa, dlouho)$ SA_2
3. $nudi\_se(X), spolecensky(X) \rightarrow jde\_do(X, restaurace)$ SA_3
4. $turista(X), ma\_hlad(X) \rightarrow jde\_do(X, restaurace)$ SA_4
5. $cestuje(X, dlouho) \rightarrow ma\_hlad(X)$ SA_5
6. $jde\_do(pepa, restaurace) \rightarrow$ PM (výchozí cílová klauzule)
7. $nudi\_se(pepa), spolecensky(pepa) \rightarrow jde\_do(pepa, restaurace)$ S(3)$\{pepa/X\}$
8. $nudi\_se(pepa), spolecensky(pepa) \rightarrow$ R(6,7)
   nelze pokračovat, vyjmeme ze zásobníku $[3, pepa/X]$, pokračujeme 4. klauzulí
9. $turista(pepa), ma\_hlad(pepa) \rightarrow jde\_do(pepa, restaurace)$ S(4)$\{pepa/X\}$
10. $turista(pepa), ma\_hlad(pepa) \rightarrow$ R(6,9)
11. $ma\_hlad(pepa) \rightarrow$ R(1,10)
12. $cestuje(pepa, dlouho) \rightarrow ma\_hlad(pepa)$ S(5)$\{pepa/X\}$
13. $cestuje(pepa, dlouho) \rightarrow$ R(11,12)
14. $\rightarrow$ R(2,13)

Všimněte si, že všechny klauzule, které jsme vytvořili uplatněním rezolučního pravidla (od bodu 7), mají prázdný konsekvent.

V Prologu:

```
turista(pepa).                                          (1)
cestuje(pepa,dlouho).                                   (2)
jde_do(X,restaurace) :- nudi_se(X),spolecensky(X).      (3)
jde_do(X,restaurace) :- turista(X),ma_hlad(X).          (4)
ma_hlad(X) :- cestuje(X,dlouho).                        (5)
?- jde_do(pepa,restaurace).                             (dotaz)
```

Ve skutečnosti bychom museli přidat ještě nějaké klauzule s predikátem `nudi_se` a `spolecensky` v hlavě (není s čím unifikovat), protože jinak by při pokusu o vyhodnocení cíle `jde_do` Prolog vypsal chybové hlášení `Predicate Not Defined`.

Postup výpočtu:

1. Výchozí cílová klauzule je
   `:- jde_do(pepa,restaurace).`

   Hledáme v hlavách klauzulí predikát `jde_do`.

2. Nalezli jsme klauzuli číslo 3, hlava klauzule souhlasí s prvním (momentálně jediným) cílem cílové klauzule. Provedeme unifikaci, výsledkem unifikace jsou klauzule
   `:- jde_do(pepa,restaurace).`
   `jde_do(pepa,restaurace) :- nudi_se(pepa),spolecensky(pepa).`

   Do zásobníku uložíme `{[3,{pepa/X_3}]`, číslo 3 určuje klauzuli.

   Cílová klauzule (po unifikaci a rezoluci) je
   `:- nudi_se(pepa),spolecensky(pepa).`

   Hledáme v hlavách klauzulí predikát `nudi_se`.

3. Navracení, predikát `nudi_se` není v hlavě žádné unifikovatelné klauzule. Ze zásobníku vyjmeme `[3,{pepa/X_3}]`, tyto údaje pomohou zjistit původní cílovou klauzuli, která se opět stává cílovou klauzulí. Prohledáváme program od klauzule číslo 4 (tj. 3+1).

   Cílová klauzule je
   `:- jde_do(pepa,restaurace).`

4. Nalezli jsme klauzuli číslo 4. Provedeme unifikaci, výsledkem unifikace jsou klauzule
   `:- jde_do(pepa,restaurace).`
   `jde_do(pepa,restaurace) :- turista(pepa),ma_hlad(pepa).`

   Do zásobníku uložíme `[4,{pepa/X_4}]`.

   Cílová klauzule je
   `:- turista(pepa),ma_hlad(pepa).`

   Hledáme v hlavách klauzulí predikát `turista`.

5. Nalezli jsme klauzuli číslo 1. Unifikace je prázdná množina (není třeba unifikovat, v obou klauzulích jsou pouze konstanty).

   Do zásobníku uložíme `[1,∅]`.

   Cílová klauzule je
   `:- ma_hlad(pepa).`

   Hledáme v hlavách klauzulí predikát `ma_hlad`.

6. Nalezli jsme klauzuli číslo 5. Provedeme unifikaci, výsledkem jsou klauzule
   `:- ma_hlad(pepa).`
   `ma_hlad(pepa) :- cestuje(pepa,dlouho).`

   Do zásobníku uložíme `[5,{pepa/X_5}]`.

   Cílová klauzule je
   `:- cestuje(pepa,dlouho).`

   Hledáme v hlavách klauzulí predikát `cestuje`.

7. Nalezli jsme klauzuli číslo 2. Unifikace je prázdná množina.

   Do zásobníku uložíme `[2,∅]`.

Cílová klauzule je

```
:- .
```

Protože je to prázdná klauzule, končíme výpočet větve s výsledkem `yes`. V dotazu nejsou žádné proměnné, proto nemusíme pokračovat dál. Vypíšeme `yes` a vyprázdníme zásobník.

Příklad 1.11

Podle stejné báze provedeme odvození odpovědi na dotaz „Kdo je hladový?“ (tj. tážeme se „Existuje někdo hladový?“ a chceme zároveň vědět, kdo to je).

V klauzulární logice:

1. $\rightarrow turista(pepa)$ SA_1
2. $\rightarrow cestuje(pepa, dlouho)$ SA_2
3. $nudi\_se(X), spolecensky(X) \rightarrow jde\_do(X, restaurace)$ SA_3
4. $turista(X), ma\_hlad(X) \rightarrow jde\_do(X, restaurace)$ SA_4
5. $cestuje(X, dlouho) \rightarrow ma\_hlad(X)$ SA_5
6. $ma\_hlad(X) \rightarrow$ PM (výchozí cílová klauzule)
7. $cestuje(X, dlouho) \rightarrow ma\_hlad(X)$ S(5)$\{X/X\}$ (unifikace)
8. $cestuje(X, dlouho) \rightarrow$ R(6,7)
9. $cestuje(pepa, dlouho) \rightarrow$ S(8)$\{pepa/X\}$ (unifikace s 2)
10. $\rightarrow$ R(2,9)

V Prologu:

```
turista(pepa).
cestuje(pepa,dlouho).
jde_do(X,restaurace) :- nudi_se(X),spolecensky(X).
jde_do(X,restaurace) :- turista(X),ma_hlad(X).
ma_hlad(X) :- cestuje(X,dlouho).

?- ma_hlad(X).
```

Aby Prolog mohl pracovat korektně, ve skutečnosti budeme potřebovat také klauzule, které mají v hlavě predikáty `nudi_se` a `spolecensky`, jako u předchozího příkladu.

Postup výpočtu:

1. Výchozí cílová klauzule je
   `:- ma_hlad(X).`
   Hledáme v hlavách klauzulí predikát `ma_hlad`.
2. Nalezli jsme klauzuli číslo 5. Výsledkem unifikace jsou klauzule
   `:- ma_hlad(X).`
   `ma_hlad(X) :- cestuje(X,dlouho).` (tj. klauzule se nemění)

Do zásobníku uložíme `[5,{X/X,X_5/X}]`.

Cílová klauzule je

```
:- cestuje(X,dlouho).
```

Hledáme v hlavách klauzulí predikát `cestuje`.

3. Nalezli jsme klauzuli číslo 2. Výsledkem unifikace jsou klauzule

   ```
   :- cestuje(pepa,dlouho).
   cestuje(pepa,dlouho).
   ```

   Do zásobníku uložíme `[2,{pepa/X}]`.

   Cílová klauzule je

   ```
   :- .
   ```

4. Je to prázdná klauzule, proto vypíšeme poslední hodnotu, která byla substituována za X (tj. to co je na zásobníku nejvýše):

   ```
   X = pepa
   ```

   a čekáme na stisk klávesy.

5. Byla stisknuta klávesa ;.

6. Navracení: vyjmeme ze zásobníku poslední uložené údaje – `[2,{pepa/X}]`.

   Cílová klauzule je

   ```
   :- cestuje(X,dlouho).
   ```

   Hledáme v hlavách klauzulí predikát `cestuje`, a to až od 3. klauzule.

7. Navracení, predikát `cestuje` není v hlavě žádné klauzule od klauzule č. 3. Ze zásobníku vyjmeme `[5,{X/X,X_5/X}]`.

   Cílová klauzule je

   ```
   :- ma_hlad(X).
   ```

   Hledáme v hlavách klauzulí predikát `ma_hlad`, a to až od 6. klauzule.

8. V bázi však už šestá klauzule není, navracení nelze provést (zásobník je prázdný), proto končíme výpočet větve s neúspěchem (vypíšeme `no`) a ukončíme celý výpočet.

Během výpočtu byl vygenerován výstup

```
X = pepa ;          (v bodu 4 tohoto postupu)
no                  (v bodu 8 tohoto postupu)
```

S anonymními proměnnými zachází Prolog při unifikaci poněkud volněji, lze je unifikovat s kteroukoliv konstantou a také proměnnou (Prolog si hlídá, aby univerzum diskurzu nebylo prázdné).

Příklad 1.12

Jsou dány tyto prologovské klauzule:

```
vlk(akela).
vyje(X) :- vlk(X).
hluk :- vyje(_).
```

Pokud se dotážeme, zda je hluk (dotaz `hluk.`, cílová klauzule je `:- hluk.`), je třeba unifikovat atom `vyje(_)` s hlavou druhé klauzule, která v příslušném argumentu obsahuje proměnnou. Unifikace nemůže znamenat zobecnění, ale buď konkretizaci nebo zachování původního stupně obecnosti. Tedy výsledkem unifikace nemůže být dosazení proměnné za anonymní proměnnou, ale naopak dosazení anonymní proměnné za proměnnou.

Zjednodušeně chápeme stupnici obecnosti jako *konstanta — anonymní proměnná — proměnná*.

Po uplatnění unifikace a rezoluce získáme cílovou klauzuli `:- vyje(_).`, v tomto kroku je úkolem zjistit, zda někdo vyje, postupujeme podobně jako v předchozím kroku.

Další cílovou klauzulí je `:- vlk(_).`, nyní unifikujeme anonymní proměnnou s konstantou `akela`. Konstanta má menší stupeň obecnosti než anonymní proměnná, proto je výsledkem unifikace dosazení konstanty.

Úkol

Máme tento program v Prologu:

```
lev(hubert).
dite(zita, hubert).
vlk(azor).
selma(X, kockovita) :- lev(X).
selma(X, psovita) :- vlk(X).
lev(X) :- dite(X, Y), lev(Y).
vlk(X) :- dite(X, Y), vlk(Y).
nebezpecny(X) :- selma(X,_).
```

Zjistěte, jak jsou vyhodnocovány dotazy

- `?- lev(X).` (Vyjmenuj všechny lvy.)
- `?- selma(azor, S).` (Jaký typ šelmy je Azor?)
- `?- nebezpecny(zita).` (Je Zita nebezpečná?)

1.4 Řízení výpočtu

Prolog má mnoho vestavěných predikátů, z nichž některé slouží k řízení výpočtu. Všechny tyto možnosti jsou podrobněji probírány ve volitelném předmětu *Praktikum z logického programování*.

1.4.1 Predikáty popření, selhání a řezu

Nás budou zajímat především predikáty `fail` a `!`. Predikát `fail` je vždy vyhodnocen jako $false$ (také se nazývá „predikát selhání“), predikát `!` se nazývá *predikát řezu* a slouží k „odříznutí“ dalších možných generovaných řešení (je vyhodnocen jako $true$, tedy uspěje, ale znemožní navracení).

Predikát `fail` v kombinaci s predikátem řezu je použit například při definování predikátu `not` pro negaci atomu:

```
not(nějaký_atom) :- call(nějaký_atom),!,fail.
not(nějaký_atom).
```

Nejdřív je zavolán cíl `nějaký_atom` pomocí vestavěného predikátu `call/1` (ten jenom zavolá – „spustí“ – svůj argument). Dále se pokračuje podle toho, jakou pravdivostní hodnotu vrátí vyhodnocení `call(nějaký_atom)`:

true: díky predikátu `fail` první klauzule vrátí hodnotu $false$ a díky predikátu `!` vyhodnocování predikátu `not(X)` nepokračuje další klauzulí, tedy `not(nějaký_atom)` je vyhodnoceno jako $false$,

$false$: v první klauzuli vyhodnocování nedojde až k predikátu řezu `!` a tedy při navracení pokračujeme k druhé klauzuli, kterou lze s cílem unifikovat vždy a protože je to fakt bez předpokladů, cíl je vyhodnocen jako *true*.

Příklad 1.13

Můžeme vyzkoušet na těch nejjednodušších atomech, zadáme následující dotazy:

```
?- not(true).
false.

?- not(fail).
true.
```

Dále vytvoříme program, ve kterém se bude vyskytovat predikát `not`:

```
strom(jedle).
strom(jablon).
strom(topol).
strom(moruse).

vyska(jedle,8).
vyska(jablon,2).
vyska(topol,10).
vyska(moruse,3).

velkyStrom(X) :- strom(X),vyska(X,Y),Y>5.

malyStrom(X)  :- strom(X),not(velkyStrom(X)).
```

Vyzkoušíme pár dotazů:

```
?- velkyStrom(jedle).
true.

?- velkyStrom(jablon).
false.

?- velkyStrom(S).
S = jedle ;
S = topol ;
false.

?- malyStrom(S).
S = jablon ;
S = moruse.
```

Program funguje podle očekávání. Všimněte si však odlišného chování Prologu u posledních dvou dotazů: zatímco při výpisu velkých stromů hledal Prolog poctivě všechna řešení a po nenalezení dalšího vypsal `false`, u malých stromů výstup false na konci chybí (není to tím, že by uživatel nestiskl středník, ostatně všimněte si té tečky na konci). Je to z toho důvodu, že predikát `not`, který zde přichází ke slovu, má ve své definici predikát řezu.

Příklad 1.14

Vyjádříme v Prologu větu „Všichni kromě Honzy jsou přítomni.“

```
pritomen(X) :- X=honza,!,fail.
pritomen(X).
```

Vyhodnocování probíhá takto: když chceme zjistit, zda je určitý člověk (například Honza) přítomen, začneme nejdřív první klauzulí (za `X` je dosazen dotyčný zjišťovaný).

Jestliže atom `X=honza` je vyhodnocen jako $true$, pokračujeme dalšími atomy první formule. Atom `!` je vyhodnocen jako $true$, atom `fail` je však vyhodnocen jako $false$, a proto dojde k navracení. Protože však před atomem `fail` je predikát řezu, k navracení nedojde a je vrácena hodnota vrácená atomem `fail`, tedy $false$. Jinými slovy – pokud je argumentem atomu `pritomen(X)` Honza, zjistíme, že není přítomen.

Jestliže však je při unifikaci za `X` dosazeno něco jiného než Honza, pak atom `X=honza` je vyhodnocen jako $false$ a hned dojde k navracení (backtrackingu), které tentokrát není „zamezeno“ predikátem řezu, proto pokračujeme k druhé klauzuli. Ta je vyhodnocena jako $true$, ať už je za její argument dosazeno cokoliv (pokud je co dosazovat), takže celkově dojdeme k závěru, že dotyčný je přítomen.

Tento postup má jednu vadu: predikát `pritomen` je možné použít pro zjištění, zda někdo konkrétní (zadaný konstantou) je přítomen, ale při dosazení proměnné (ptáme se, kdo je přítomen) predikát selže (Prolog vypíše chybové hlášení).

Příklad 1.15

Vyjádříme v Prologu větu „Honza není přítomen.“

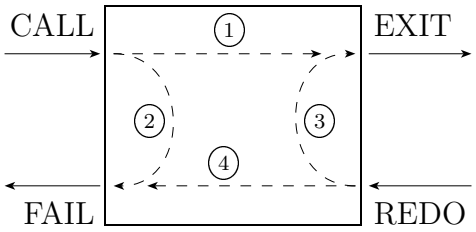
```
pritomen(X) :- X=honza,!,fail.
```

Poznámka:

V Prologu však ve skutečnosti *není třeba explicitně uvádět, že daný subjekt či objekt nemá určitou vlastnost*, protože Prolog pracuje v uzavřeném světě (o čem mu neřekneme, že je pravda, to podle něho není pravda), za určitých okolností však takto můžeme vyřešit některé problémy vyplývající ze způsobu práce Prologu. Proto predikát `not` používáme spíše v podmínkách (jestliže daný objekt či subjekt nemá danou vlastnost...).

1.4.2 Krabičkový model

Zpracování atomů a klauzulí se dá vizualizovat pomocí blokového schématu, kterému se familiárně říká „krabičnový model“ (někteří autoři zůstávají jednoduše u názvu „blokové schéma“, případně bez zdrobnění „krabicový model“).



Obrázek 1.9: Krabičkový model běžného atomu v klauzuli

Na obrázku 1.9 vidíme krabičkový model běžného atomu. Předpokládejme, že se právě vyhodnocuje klauzule s několika atomy, konkrétně jeden z těchto atomů (tedy cíl pro daný krok). Právě se nacházíme ve výpočetním stromě na určitém místě. Cíl je zavolán (vnitřně predikátem `call`) a dál lze pokračovat dvěma způsoby:

① atom je vyhodnocen jako *true*: do zásobníku vložíme informaci o vyhodnocení (klauzule a unifikátor), opouštíme krabičku pro tento atom směrem vpravo (EXIT) a přecházíme do krabičky pro následující atom (např. v cílové klauzuli za čárkou), ve výpočetním stromě pokračujeme dolů,

② atom je vyhodnocen jako *false*: opouštíme krabičku směrem vlevo (FAIL), tedy zpět, probíhá backtracking (ve větvi nelze dál pokračovat).

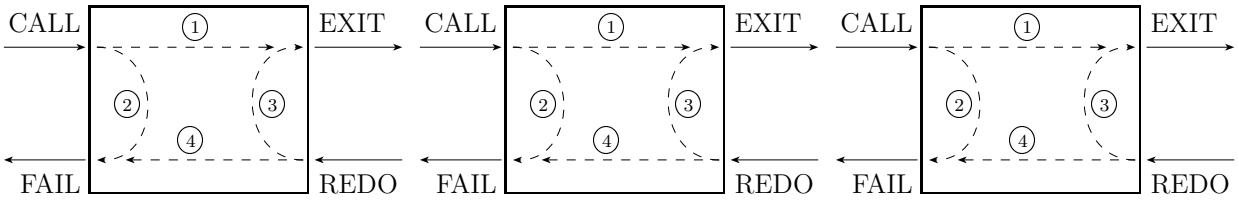
V případě ① tedy přecházíme do „krabičky“ vpravo příslušející dalšímu atomu v pořadí, kde proběhne podobné vyhodnocení jako u této, atd. Někde dál ve větvi dojde k backtrackingu, tedy navracení směrem nahoru (podobně jako by byl případ ②). Pak se do naší krabičky dostaneme znovu, a to zprava (REDO). Co potom? Vyndáme ze zásobníku prvek, který jsme v bodu ① vložili (klauzule a unifikátor) a opět máme dvě možnosti:

③ najdeme další alternativu (unifikaci nebo další klauzuli pro pravidlo rezolučního řezu): vložíme tuto informaci do zásobníku a pokračujeme do další vhodné „krabičky“ vpravo (EXIT), tedy začneme tvořit novou větev,

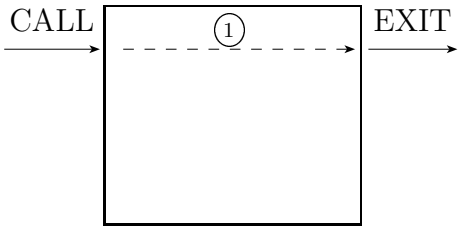
④ v této větvi nelze pokračovat, proto provedeme backtracking směrem vlevo (FAIL), tedy ve stromě nahoru, a hledáme další řešení.

Obrázek 1.10 na straně 29 ukazuje, jak na sebe navazuje zpracování několika následujících atomů (cílů), což odpovídá tomu, že ve výpočetním stromě jdeme v některé větvi směrem dolů (a při navracení pak směrem nahoru, zde do krabiček vlevo).

Jak to však bude vypadat při použití predikátu řezu? To vidíme na obrázku 1.11. Tento predikát vždy uspěje, tedy jdeme přímo ke konci EXIT, a pokud je nějaká „krabička“ (atom) dál vpravo, pokračuje se ve vyhodnocování. Pokud však někde vpravo dojde k backtrackingu, zpět do této „krabičky“ už se nedostaneme, tedy ani není cesta ve výpočetním stromě nahoru, nehledá se další řešení a výpočet skončí.



Obrázek 1.10: Krabičky pro několik po sobě následujících vyhodnocovaných atomů



Obrázek 1.11: Krabičkový model predikátu řezu

Příklad 1.16

Mějme tento jednoduchý program:

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

Zadáme tento dotaz:

```
?- a.
```

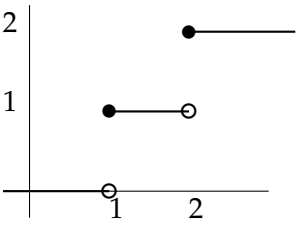
Jak proběhne vyřešení tohoto dotazu? Záleží samozřejmě na hodnotách jednotlivých atomů.

- pokud budou b, c splněny, přecházíme k predikátu řezu, který nás pustí dál vpravo, pokračuje se vyhodnocením d, e (použije se substituce, kterou jsme pro b, c, atd.), jdeme po větvi výpočetního stromu dolů, na konci větve se sice „otočíme“ a začne backtracking, ale když při navracení znovu narazíme na predikát řezu, ukončíme výpočet a nehledáme další řešení pro b, c,
- pokud b, c nebudou splněny, na řez nedojde a není nutno řešit, jestli bude nebo nebude backtracking.

Predikát řezu tedy pustí výpočet směrem vpravo, ale při návratu směrem doleva postaví bariéru (ukončí výpočet, odřízne další větve výpočtu generující další řešení).

Příklad 1.17

Je dána tato funkce:



Naším úkolem je reprezentovat tuto funkci v Prologu.

První řešení:

```
f(X,Y) :- X < 1, Y=0.
f(X,Y) :- 1=<X, X<2, Y=1.
f(X,Y) :- 2=<X, Y=2.
```

Program konzultujeme a položíme několik dotazů:

```
?- f(0,0).
true  .

?- f(0,1).
false.

?- f(0,Y),Y>1.
false.
```

Klauzule jsme sestavili prostě podle tří navzájem disjunktních možností. Program bude fungovat, jen se zbytečně hledají další řešení, když už je jedno správné nalezeno. Projevuje se to u prvního položeného dotazu. Proto použijeme predikáty řezu:

```
f(X,Y) :- X < 1, !, Y=0.
f(X,Y) :- 1=<X, X<2, !, Y=1.
f(X,Y) :- 2=<X, Y=2.
```

Důsledkem je, že po nalezení správného řešení dostaneme první nalezené řešení a další se už nehledá. Třetí klauzuli můžeme zkrátit do formy „když neplatí nic z výše uvedeného, bude platit i toto“, protože k ní se stejně dostaneme jen v případě, že předchozí dvě klauzule neuspěly:

```
f(X,Y) :- X < 1, !, Y=0.
f(X,Y) :- 1=<X, X<2, !, Y=1.
f(_,Y) :- Y=2.
```

Ještě drobná úprava, aby program fungoval optimálněji:

```
f(X,0) :- X<1,!.
f(X,1) :- X>=1,X<2,!.
f(_,2).
```

Rozlišujeme dva druhy řezu:

- zelený řez – nemění deklarativní sémantiku programu, tedy po jeho odstranění dostaneme stejné výsledky (jen odřízneme neúspěšné větve),
- červený řez – zasahuje do sémantiky programu, ovlivňuje vypisované výsledky, po jeho odstranění se vypíšou i jiné výsledky než před odstraněním.

Příklad 1.18

Podívejme se na program v předchozím příkladě. Co se stane, když odstraníme predikáty řezu z programu po první úpravě?

```
f(X,Y) :- X < 1, Y=0.
f(X,Y) :- 1=<X, X<2, Y=1.
f(X,Y) :- 2=<X, Y=2.
```

Program bude podávat stejné výsledky, jen zbytečně procházíme i ty větve, které nedávají správné řešení. Proto jde v obou případech o zelený řez. Ale podívejme se na finální program a odstraňme z něj predikáty řezu:

```
f(X,Y) :- X < 1, !, Y=0.
f(X,Y) :- 1=<X, X<2, !, Y=1.
f(_,Y) :- Y=2.
```

Zadáme několik dotazů:

```
?- f(0,0).
true ;
false.

?- f(0,Y),Y>0.
Y = 2.

?- f(0,Y).
Y = 0 ;
Y = 2.
```

Už výsledek prvního dotazu působí zvláštně. Ovšem u druhého a třetího přímo dostáváme chybné řešení. Pro X=0 má být přece řešení Y=0 a žádné jiné. To znamená, že jsme odstranili červený řez. Kromě správného řešení Prolog pokračoval našel další řešení v jiných větvích, ale ta nejsou správná, přesto výpočet nebyl zastaven.

Zelený řez má za úkol zoptimalizovat výpočet, rozhodně je v programu užitečný. Červenému řezu je lepší se vyhnout, protože za určitých okolností může mít vedlejší nečekané důsledky.

Příklad 1.19

Pokračujme v příkladu: pokud budeme chtít využít pouze zelený řez (a vyhnout se červenému), bude program vypadat takto:

```
f(X,0) :- X<1, !.
f(X,1) :- X>=1, X<2, !.
f(X,2) :- X>=2.
```

Kapitola 2

Programovací techniky v Prologu

Rychlý náhled: V této kapitole se zaměříme na různé programovací techniky v jazyce Prolog. Projdeme si syntaxi Prologu, typy používaných údajů, problematiku použití rekurze v kódu, negaci, vstupy a výstupy.

Klíčová slova: Prolog, program, databáze, anonymní proměnná, predikát rovnosti, rekurze, rezoluce, negace, přiřazování, porovnávání, unifikace.

Cíle studia: Po prostudování této kapitoly se naučíte používat některé nové predikáty pro práci s proměnnými, výrazy a atomy ve formulích.

2.1 Ovlivňování obsahu databáze

Z předchozí kapitoly, kde jsme se učili o nápovědě, víme, že predikáty `asserta/1` a `assertz/1` slouží k dodatečném přidávání klauzulí do báze. Problém je v tom, že Prolog nám nedovolí takto „obohacovat“ definici již zavedených predikátů, tedy například pokud bychom chtěli přidat do databáze další osobu, která je žena takto:

```
assert(zena(kamila)).     % chyba!!!!
```

Obdrželi bychom chybové hlášení, že k provedení operace chybí přístupové oprávnění. Ve skutečnosti to jde, ale z důvodu určité ochrany databáze je třeba v programu (ideálně na jeho začátku) označit predikáty, pro které je dovoleno dynamicky za běhu programu přidávat další klauzule. Takže na začátek souboru s programem bychom napsali:

```
:-dynamic(zena/1).
:-dynamic(muz/1).
```

Pak už bude možné v prostředí Prologu pro tyto predikáty bez problémů přidávat nové klauzule pomocí `asserta/1` a `assertz/1`. Pro nové predikáty to samozřejmě není nutné, například:

```
?- retract(zamestnani(programator)).
true.
```

```
?- assertz(zamestnani(jan,programator)).
true.

?- assertz(zamestnani(ivana,manager)).
true.

?- listing(zamestnani).
:- dynamic zamestnani/1.


:- dynamic zamestnani/2.

zamestnani(jan, programator).
zamestnani(ivana, manager).

true.
```

No, ve skutečnosti existuje cesta, jak do báze vložit klauzuli pro predikát, pro který již v bázi nějaké klauzule jsou, a přitom není označen jako dynamic, ale pozor – následující postup přepíše všechny klauzule s daným predikátem, které do té chvíle v bázi existovaly, což tedy nechceme.

Postup

Postup je takový, že použijeme příkaz `[user].`, čímž dáme na vědomí, že teď se bude konzultovat to, co zadá přímo uživatel z klávesnice. Ve skutečnosti vytváříme nový soubor v RAM s daným obsahem. Postupně zadáme nové klauzule (zde zadáme jen jednu) a pak stiskneme klávesovou zkratku Ctrl+D.

```
?- listing(zena).
zena(jana).
zena(zuzana).
zena(ivana).
zena(petra).

true.

?- [user].
|: zena(kamila).


Warning: user://2:80:
Warning:    Redefined static procedure zena/1
Warning:    Previously defined at d:/vyuka/prolog/priklady/rodina.pl:6
|:
% user://2 compiled 0.00 sec, 1 clauses
true.

?- listing(zena).
zena(kamila).

true.
```

Všimněte si, že zatímco před zadáváním byly čtyři klauzule pro predikát `zena/1`, po ukončení zadávání je v bázi jen ten jeden, který jsme nově vytvořili. Původní byly odstraněny. Ostatní klauzule (pro další predikáty) nebudou dotčeny. Pokud chceme vrátit původní stav klauzulí, nestačí použít `make`, musíme znovu konzultovat soubor.

Naopak pro odebrání klauzulí pro daný predikát z databáze slouží predikát `retract/1`. Například:

```
retract(zamestnani).
```

Úkol

Vyzkoušejte si dynamické přidání nových klauzulí pomocí `assertz/1` a `asserta/1` (obojí) – nezapomeňte na predikát `dynamic`, a pak vypište klauzule pro nový predikát pomocí `listing/1`. Dále si rozšiřte program o nové klauzule (třeba přidejte další muže a ženy, případně další vztahy).

2.2 Datové typy a čísla

Základní prvky syntaxe Prologu již známe – proměnné, anonymní proměnné, atomy, samotné klauzule. Podíváme se, co dalšího lze v Prologu používat.

2.2.1 Přiřazování, porovnávání, unifikace

V Prologu je s počítáním a porovnáváním trochu problém, funguje to trochu jinak než v procedurálních programovacích jazycích. Například rovnítko není pro přiřazování, používá se pro unifikaci, což není totéž. Pro přiřazování se používá operátor `is` (ano, to je opravdu operátor). Srovnejte následující dva dotazy:

```
?- X = 1+4.
X = 1+4.

?- X is 1+4.
X = 5.
```

Ten první je unifikace (proměnná `X` nabyla hodnoty řetězce `1+4`), ten první je opravdu přiřazení, tedy provedl se výpočet a jeho výsledek se teprve unifikoval. Pokud by za rovnítkem nebyl výraz s operátory, ale pouze číslo, pak by ovšem bylo možné použít i rovnítko, šlo by o jednoduchou unifikaci.

Unifikace pro operátor = funguje i u parametrů termů, například:

```
?- pratele(X,Y) = pratele(pat,mat).
X = pat,
Y = mat.
```

Používáme tyto aritmetické operátory (jde o výběr nejpoužívanějších):

- `+`, `-`, `*`, `/`, `**` (mocnina)
- `//` (celočíselné dělení), `rem` (zbytek po celočíselném dělení `//`)
- `div` (celočíselné dělení se zaokrouhlením dolů) , `mod` (zbytek po celočíselném dělení `div`)
- `max` (maximum), `min` (minimum)
- `is` (vyhodnocení a unifikace)

Zatímco pro běžné relace nemá Prolog problém s inverzní funkcí (dokáže dohledat to, co se dá dosadit, což jsme viděli v předchozích dotazech), u operátoru is nelze s inverzí počítat.

Například toto funguje:

```
?- X is 74+8.
X = 82.
```

Ale toto fungovat nebude, protože po Prologu chceme inverzní chování pro operátor `is`:

```
?- 82 is Y-5.
ERROR: Arguments are not sufficiently instantiated
```

Operátor `is` je určen pro výpočet a přiřazení, přičemž na levé straně musí být proměnná, tedy následující není správně (protože se nejedná o relační operátor):

```
?- 1+2 is 2+1.
false.
```

Ovšem je zajímavé, že operátor = naopak zvládá i inverzní chování (jenže zase nezvládá výpočet):

```
?- 2 = X.
X = 2.
```

 Operátor =:= provádí výpočet, ale neprovádí unifikaci, tedy nelze ho použít na volnou proměnnou:

```
?- 1+2 =:= 2+1.
true.

?- X =:= 1.
ERROR: Arguments are not sufficiently instantiated
```

 Ve skutečnosti se jedná o relační operátor, obvykle ho používáme na predikáty. Přehled používaných relačních operátorů:

- <, >, =<, <= (pozor, kde se píše rovnítko u těch složených)
- =:= (rovno), =/= (nerovno)

Můžeme také používat bitové operace:

- >> (bitový posun vpravo), << (bitový posun vlevo)
- \ (bitová negace)
- \ (bitové OR), /\ (bitové AND)
- xor (bitový XOR)

Řetězce porovnáváme takto:

- == (termy jsou totožné)
- \== (termy jsou různé)

Tabulka 2.1 shrnuje vlastnosti operátorů (no, vlastně predikátů) na porovnávání termů.

Příklad 2.1

Pozor na apostrofy vs. uvozovky. Řetězec utvoříme pomocí obojího, ale nepůjde o ekvivaletní řetězce. Navíc to, co máme v apostrofech (a neobsahuje to mezeru a začíná malým písmenem), je ekvivalentní s řetězcem po odstranění apostrofů.

Predikát	Význam	Provádí unifikaci?	Provádí výpočet?
`=`	shoduje se	ano, obě strany	ne
`\=`	neshoduje se, totéž jako not(...=...)	ne	ne
`is`	přiřazení	ano, levá strana	ano, pravá strana
`=:=`	vyhodnotí, ověří shodu	ne	ano, obě strany
`=\=`	vyhodnotí, ověří neshodu	ne	ano, obě strany
`==`	pouze porovná, ověří shodu	ne	ne
`\==`	pouze porovná, ověří neshodu	ne	ne

Tabulka 2.1: Relační operátory v Prologu

Vyzkoušíme různé možnosti:

```
?- X = 'jahoda', X == 'jahoda'.
X = jahoda.

?- X = 'jahoda', X \== 'jahoda'.
false.

?- X = 'jahoda', X == "jahoda".
false.

?- X = 'jahoda', X == jahoda.
X = jahoda.

?- X = 'jahoda', X \== "jahoda".
X = jahoda.
```

Relačními operátory jsou i čárka (ve smyslu konjunkce) a středník (ve smyslu disjunkce). Dále je to relační operátor `not/1`, který však používáme s velkou opatrností.

Úkol

Vyzkoušejte různé operátory uvedené v této sekci. Zjistěte také, jak se používají operátory pro maximum a minimum.

2.2.2 Testování typů údajů

Při práci s aritmetickými výrazy (ale i jindy) potřebujeme znalost o skutečném typu hodnoty uložené do proměnné či jiného termu. K tomuto účelu slouží vestavěné predikáty, které najdeme v tabulce 2.2.

Například pokud chceme proměnnou využít v aritmetickém výrazu, měli bychom otestovat, zda je v ní uloženo číslo.

Tyto predikáty lze používat jak v dotazech, tak i v klauzulích v programu.

Predikát	**Význam – testuje, zda je** `argument` ...
`atom(argument)`	atom, také řetězcová konstanta
`atomic(argument)`	atom, konstantní číslo, řetězec
`integer(argument)`	celé číslo, také v proměnné
`float(argument)`	racionální číslo, také v proměnné
`number(argument)`	číslo, také v proměnné
`string(argument)`	řetězec, také v proměnné
`var(argument)`	volná proměnná
`nonvar(argument)`	vázaná proměnná

Tabulka 2.2: Vestavěné predikáty pro testování termů

Příklad 2.2

Nejdřív se zaměříme na datové typy pro čísla a řetězce. Opět položíme dotazy a budeme sledovat, jak Prolog odpoví.

```
?- integer(2).
true.

?- integer(1.5).
false.

?- X=2,integer(X).
X = 2.

?- X is 2, integer(X), Y is X.
X = Y, Y = 2.

?- float(1.5).
true.

?- float(2).
false.
```

U čísel si všimněte, že celá čísla nejsou považována za racionální (float, čísla v plovoucí řádové čárce). A teď řetězce:

```
?- string(abc).
false.

?- string('abc').
false.

?- string("abc").
true.
```

Takže přímo za řetězec je považováno jen to, co je v uvozovkách.

Připomeňme si, že proměnné mohou být vázané nebo volné. Vázaná proměnná je asociována s konkrétní hodnotou, kdežto u volné proměnné není stanoveno, co obsahuje. Predikát `var/1` vrací true, pokud parametr je volná proměnná, kdežto `nonvar/1` vrátí true pro vázanou proměnnou.

Příklad 2.3

Otestujeme si, zda je proměnná volná nebo vázaná.

```
?- var(X).
true.

?- nonvar(X).
false.

?- X=2,var(X).
false.

?- X=2,nonvar(X).
X = 2.
```

Tyto predikáty jsou celkem předvídatelné.

Další dva predikáty jsou poněkud hůře rozlišitelné. Predikát `atom/1` vrací true, pokud jeho argument je vázán na atom (popřípadě je přímo atomem), kdežto `atomic/1` vrací true, pokud jde o vázaný term (ne proměnnou, ale konstantu ano) a není složený.

Příklad 2.4

Ukážeme si, jak reagují predikáty `atom/1` a `atomic/1`.

```
?- atom(2).
false.

?- atomic(2).
true.

?-  X is 2, atom(X), Y is X.
false.

?- X is 2, atomic(X), Y is X.
X = Y, Y = 2.

?- atom(abc).
true.

?- atomic(abc).
true.

?- atom(soused).
true.

?- atom(soused(a,b)).
false.

?- atom("abc").
false.

?- atomic("abc").
true.
```

Seznamy jsme zatím nebrali, takže jen stručně: seznam (jakýchkoliv prvků) zapisujeme obvykle do hranatých závorek, jde vlastně o složenou proměnnou. Je seznam atom nebo atomic? To záleží na tom, jestli je prázdný. Prázdný seznam `[]` totiž sice není atom, ale považován za složený vázaný term (neobsahuje nic neurčitého).

```
?- X=[a,b], atom(X).
false.


?- X=[a,b], atomic(X).
false.

?- X=[], atom(X).
false.

?- X=[], atomic(X).
X = [].
```

Úkol

Vyzkoušejte ukázky v příkladech této sekce, také další variace.

2.3 Výpočty a rekurze

2.3.1 Počítání a posílání hodnot v rekurzi

Pokud chceme provést jednoduchý výpočet a vypsat výsledek, tak už víme, že můžme použít operátor `is`. Výsledek také lze použít v klauzuli ve vázané proměnné. Pozor, na levé straně tohoto operátoru musí být volná proměnná (naopak ve výrazu musí být pouze vázané proměnné, tedy takové, které už byly dříve inicializovány). Proto nemůžeme mít na obou stranách tohoto operátoru tutéž proměnnou, například chybové hlášení by vyvolalo toto:

```
N is N + 1     % chyba!
```

Jde zjevně o čítač, který už byl dříve inicializován (vázán), tedy ho nelze použít na levé straně.

Je třeba si také uvědomit, že veškeré proměnné jsou v Prologu z principu lokální v rámci klauzule, proto proměnná `X` v jedné klauzuli je vlastně jiná proměnná než `X` v jiné klauzuli, i když se stejně jmenují. Při unifikaci si Prolog k názvům proměnných přidává identifikátor klauzule, ve které se proměnná nachází, aby nedocházelo ke kolizím názvů.

 Rekurze je základní výpočetní prvek Prologu. Setkávali jsme se s ní už na předchozích stránkách, když jsme do těla klauzule umístili atom s tímtéž predikátem, jaký měl atom z hlavy klauzule. Predikát definujeme pomocí téhož predikátu, přičemž se musíme postarat o včasné ukončení rekurze. Je třeba definovat ukončující podmínku, a to na vhodném místě v programu. Abychom tedy neskončili v nekonečné rekurzi, ukončující klauzuli dáváme jako první.

 Poměrně obvyklou programovací technikou pro rekurzivní výpočty je vytvoření dvou verzí predikátů, jedné „pro uživatele“ (tj. rozhraní, hovorově „košilka“), druhé pracovní, která sku-

tečně provede výpočet. Důvodem může být například využití pomocných proměnných, kterými nemusíme uživatele zatěžovat.

Příklad 2.5

Nejdřív si ukážeme výpočet faktoriálu. Jak víme, matematicky je definován takto:

```
faktorial(0) = 1
faktorial(N) = N * faktorial(N-1) pro N>0
```

První vzorec zastavuje rekurzi, samotná rekurze je vidět v druhém vzorci. Všimněte si, že v druhém vzorci je (na konci) podmínka určující, kdy vlastně má rekurze skončit.

V Prologu bude vhodné vytvořit dva predikáty. Jeden bude „finální“, ten použijeme v dotazech, druhý bude pomocný používající rekurzi.

Nejdřív si definujeme „finální“ predikát `faktorial/2`, pak teprve pomocný „pracující“ predikát `fakt_pom/3`. Ve finálním predikátu určeném pro volání uživatelem je prvním parametrem číslo, ze kterého chceme faktoriál vypočíst, druhý parametr je určen pro výsledek:

```
faktorial(N, Vysl) :-
    fakt_pom(N, 1, Vysl).
```

V předpokladu (tj. v predikátové logice před implikací) máme pomocný predikát `fakt_pom/3`. První parametr je opět číslo, ze kterého v daném kroku počítáme faktoriál, druhý parametr je mezivýsledek (akumulátor), který je inicializován na 1 (protože budeme násobit, tedy potřebujeme „neutrální prvek“). Třetí parametr je proměnná pro výsledek.

Dále přidáme klauzule pro predikát `fakt_pom/3`. První bude zastavovat rekurzi, druhá klauzule bude v rekurzi počítat výraz `N * fakt(N-1)`.

```
fakt_pom(0, Vysl, Vysl).
fakt_pom(Citac, Akumulator, Vysl) :-
    Citac > 0,
    C is Citac-1,
    A is Akumulator*Citac,
    fakt_pom(C, A, Vysl).
```

Klauzule zastavující rekurzi jednoduše říká, že pro `N=0` je výsledek stejný jako akumulátor (tj. provede se unifikace třetího argumentu s druhým – do výsledku se načte hodnota akumulátoru).

Klauzule s rekurzí nejdřív otestuje, zda je `Citac` větší než 0. To se může zdát zbytečné, protože situace pro `Citac=0` je ošetřena první klauzulí, ale není zbytečná – ostatně můžete vyzkoušet, co se stane, když ten řádek odstraníte.

Pokud je tedy `Citac` větší než nula, vytvoří se pomocná lokální proměnná `C` a inicializuje na hodnotu o jedna menší než `Citac`. Dále se provede výpočet pro tento krok: akumulátor se vynásobí hodnotou `Citac` pro tento krok, výsledek je v proměnné `A`. Následně zavoláme rekurzívně tentýž predikát, a to s upravenými hodnotami `C` a `A`. Třetí parametr je proměnná, do které chceme v rekurzí poslat zpět nahoru výsledek.

Funkci použijeme takto:

```
?- faktorial(4,V).
V = 24 .
```

Nicméně se můžeme i zeptat, zda určité číslo je faktoriálem daného čísla, například:

```
?- faktorial(4,24).
true .
```

Při položení dotazu `?- faktorial(3,F).` se v rekurzi volají klauzule:

```
fakt_pom(3, 1, Vysl).
    fakt_pom(2, 3, Vysl).
        fakt_pom(1, 6, Vysl).
            fakt_pom(0, 6, Vysl). % poslední unifikace, F = Vysl.
```

Příklad 2.6

Jiný způsob řešení faktoriálu:

```
faktorial(0,1).
faktorial(Citac, Vysl) :-
    Citac > 0,
    C is Citac-1,
    faktorial(C,V),
    Vysl is Citac * V.
```

Jaký je rozdíl oproti předchozímu řešení? Předně jsme si nedělali starosti s nějakým oddělováním predikátů, a dále jsme přesunuli násobení až do fáze navracení z rekurze (v předchozím příkladu se násobilo tehdy, když jsme v rekurzi postupovali dolů, tedy nejdřív jsme násobili a až potom vytvářeli další patro rekurze).

Úkol

Vyzkoušejte oba uvedené způsoby naprogramování faktoriálu (odlište je např. pojmenováním predikátů, ať je můžete využít v jednom zdrojovém souboru), porovnejte jejich složitost a také možnosti využití.

Zvažte, jak by se dalo přidat ověření správnosti datového typu u zadávaných hodnot.

2.3.2 Relační operátory při dělení výpočtu

Někdy je třeba vytvořit různé větve výpočtu přímo v kódu, například tehdy, když dělíme výpočet podle hodnoty proměnné (podle toho, zda je hodnota nad nebo pod daným limitem). Postup si ukážeme na Fibonacciho posloupnosti a algoritmu na výpočet největšího společného dělitele.

Příklad 2.7

Dále se podíváme na Fibonacciho posloupnost. Je to v principu podobné faktoriálu: jde o řadu, kde každý prvek (od třetího dále) je součtem předchozích dvou. Fibonacciho posloupnost tedy vypadá takto:

1, 1, 2, 3, 5, 8, 13, 21, ...

Při výpočtu N-tého prvku posloupnosti půjdeme v rekurzi od hledaného indexu směrem k začátku posloupnosti, samotné sčítání budeme provádět při návratu z rekurze (ve stromě směrem nahoru). Jako první klauzuli použijeme ukončení rekurze, v těle klauzule bude podmínka použití pro index (pořadí) 1:

```
fib(X, X) :-
X =< 1.
```

Všimněte si operátoru „menší rovno“: rovnítko je vždy u „zobáčku“.

Následuje klauzule s rekurzí v těle.

```
fib(Poradi,Vysledek) :-
    Poradi > 1,
    Poradi1 is Poradi-1,
    Poradi2 is Poradi-2,
    fib(Poradi1 ,Vysledek1),
    fib(Poradi2, Vysledek2),
    Vysledek is Vysledek1 + Vysledek2.
```

Opět nejdřív umístíme určení, pro které hodnoty prvního argumentu má být tato klauzule provedena, dále vytvoříme pomocné proměnné pro indexy předchozích dvou prvků posloupnosti, dále rekurzívně zjistíme hodnoty předchozích dvou prvků a vypočteme výsledek pro prvek tohoto kroku.

Použití je následující (zde pro čtvrtý prvek posloupnosti):

```
?- fib(4,V).
V = 3 .
```

Příklad 2.8

Pomocí rekurze definujeme predikát pro výpočet největšího společného dělitele dvou čísel Euklidovým algoritmem. První dva argumenty predikátu jsou vstupní a určují dvě zpracovávaná čísla, do třetího argumentu uložíme výsledek (bude pouze výstupní).

```
nsd(A,0,A).
nsd(A,A,A).

nsd(A,B,Vysl) :-
    A < B,
    nsd(B,A,Vysl).

nsd(A,B,Vysl) :-
    A > B,
    Pom is A mod B,
    nsd(B,Pom,Vysl).
```

První dvě klauzule zastavují rekurzi. Ošetřují nejjednodušší případy, kdy druhé číslo je nula a kdy jsou obě čísla stejná. Třetí klauzule pouze obrací pořadí argumentů v případě, že první číslo je menší než druhé. Samotný rekurzivní výpočet probíhá v poslední klauzuli.

Ještě opravíme možné nesprávné chování programu při zadání nečíselných nebo jinak nesprávných hodnot (argumenty by měly být přirozená čísla):

```
nsd(A,0,A) :- integer(A), A>0.
nsd(A,A,A) :- integer(A), A>0.

nsd(A,B,Vysl) :-
    integer(A), integer(B),
    A > 0, A < B,
    nsd(B,A,Vysl).

nsd(A,B,Vysl) :-
    integer(A), integer(B),
    B > 0, A > B,
    Pom is A mod B,
    nsd(B,Pom,Vysl).
```

Dotazy mohou být ve formě `nsd(4,10,X),` do třetího argumentu se ukládá výsledek.

Postup není zcela korektní (například když po vypsání výsledku stiskneme středník pro vypsání dalšího řešení, obdržíme chybové hlášení) ani optimální, s přihlédnutím k dosavadní probrané látce však postačí.

Postup

Ještě to trochu pozměníme a vylepšíme, a to způsobem, který bude jasnější až po probrání dalších sekcí (zejména přidání operátoru řezu, což je ten vykřičník):

```
nsd(X,_,_) :-
    X=<0,write('NSD nevypoctu.'),!,false.

nsd(_,X,_) :-
    X=<0,write('NSD nevypoctu.'),!,false.

nsd(P,P,P) :- !.
nsd(P1,P2,V) :-
    P1>P2,
    P is P1-P2,
    nsd(P,P2,V),!.

nsd(P1,P2,V) :-
    P1<P2,
    P is P2-P1,
    nsd(P1,P,V),!.
```

Úkoly

1. Vyzkoušejte programy pro výpočet n-tého člena Fibonacciho posloupnosti a NSD. Myslíte, že by bylo možné naprogramovat jinou variantu, podobně jako u faktoriálu?
2. Program pro výpočet n-tého člena Fibonacciho posloupnosti rozšiřte o predikát, který zjistí, zda zadané číslo je prvkem Fibonacciho posloupnosti. Jediným argumentem predikátu bude hledané číslo.

2.4 Jednoduchý výstup

S Prologem můžeme pracovat i po stránce vstupu a výstupu:

- `write/1` – vypíše svůj parametr, nevyhodnocuje ho,
- `display/1` – provede něco podobného, výraz chápe jako výraz, ale nevyhodnotí ho,
- `nl/0` – přechod na nový řádek (new line),
- `read/1` – načtení vstupu od uživatele.

Příklad 2.9

Ukázky použití `write/1` a `nl/0`:

```
?- write('abc').
abc
true.
?- X = 'abc',write(X), nl,nl.
abc

X = abc.

?- write(3*2+1).
3*2+1
true.

?- display(3*2+1).
+(*(3,2),1)
true.

?- X is 1+2, display(X).
3
X = 3.
```

2.5 Formát formule

2.5.1 Disjunkce

Z klauzulární logiky víme, že disjunkci atomů v antecedentu (tj. v předpokladech) obvykle řešíme rozdělením původní klauzule na více klauzulí. Toto řešení však někdy nemusí být vyhovující, například z důvodu přehlednosti programu nebo pro ztrátu vazby mezi stejně pojmenovanými proměnnými, které se tak dostanou do různých klauzulí.

Prolog nabízí možnost reprezentovat disjunkci použitím symbolu „`;`“ místo čárky mezi atomy. Zatímco čárka mezi atomy znamená konjunkci, středník určuje disjunkci. Symbol „`;`“ je ve skutečnosti predikát se dvěma argumenty (jeho arita je 2), což zapisujeme „`;/2`“.

Příklad 2.10

Větu „Student je člověk, který nosí index nebo skripta.“ lze do Prologu zapsat dvěma způsoby – buď dvěma klauzulemi, anebo v jediné klauzuli s disjunkcí:

1)
```
student(X) :- clovek(X) , nosi(X,index).
student(X) :- clovek(X) , nosi(X,skripta).
```

2)
```
student(X) :- clovek(X) , (nosi(X,index) ; nosi(X,skripta)).
```

Oba způsoby jsou z pohledu Prologu rovnocenné. Závorky kolem atomů spojených disjunkcí jsou v tomto případě nutné, protože konjunkce má v Prologu vyšší prioritu než disjunkce (podobně jako například ve výrazu $a * b + c$ násobení má vyšší prioritu a více váže).

Chybné chování by se projevilo například v programu:

```
clovek(pepa).
nosi(pepa,skripta).
slepeckyPes(azor).
nosi(azor,index).
student(X) :- clovek(X) , nosi(X,skripta) ; nosi(X,index).
```

Na dotaz `student(X).` bychom kromě správné odpovědi `X = pepa` dostali také nesprávnou odpověď `X = azor`.

Disjunkci lze použít také v zadání dotazu, kde platí stejná pravidla jako při použití v antecedentu pravidla báze (raději závorkujeme). S dalším typickým použitím jsme se už seznámili – když existuje více různých odpovědí na jeden dotaz, k dalším odpovědím se dostaneme přes opakované stisknutí klávesy se středníkem.

2.5.2 Negace

Vzhledem k tomu, že v Prologu lze využívat pouze Hornovy klauzule, jsou omezeny možnosti *řešení negace* přesunem atomu z antecedentu do konsekventu a naopak (i když je lze použít, pokud vyjde Hornova klauzule). Proto máme speciální predikát `not`.

Příklad 2.11

Vezměme si takto zapsaný predikát:

```
not(p(a,Y))
```

Jak se bude vyhodnocovat? Nejdřív se vyhodnotí predikát `p(a,Y)`, vyjde `true` nebo `false`, a tato hodnota se převrátí. Pokud je původní hodnota predikátu `p/2` true, pak to znamená, že daný fakt je odvoditelný z báze, jeho negace není odvoditelná z báze (všimněte si, že ze není napsáno, že je nepravdivý).

Jenže to není všechno. V argumentu máme (pravděodobně původně) vázanou proměnnou `Y`. Co se s ní stane? Jestliže je tedy proměnná `Y` vázaná univerzálně (obecným kvantifikátorem), pak se po znegování stane volnou proměnnou a ztratí vazbu na své předchozí výskyty.

Takže si to shrňme:

- anonymní proměnná reprezentující existenčně vázanou proměnnou z klauzulární logiky je Prologem chápána jako volná, tuto vazbu neumí Prolog při negaci korektně vyřešit,
- proměnná vázaná univerzálním kvantifikátorem (tj. začínající velkým písmenem) přestává být vázaná, stává se volnou proměnnou a přestává být totožná s ostatními (dříve či později uvedenými) proměnnými téhož jména v klauzuli.

To řešíme tak, že pokud se opravdu nemůžeme negaci vyhnout, tak v klauzuli umístíme negovaný predikát až za jiné predikáty obsahující v argumentech tutéž proměnnou, a pokud existuje více pravidel se stejnou hlavou (závěrem), pravidlo obsahující negaci umístíme z nich jako poslední.

Například místo klauzule

```
r(a,b) :-
p(X), not(q(X,Y)), q(Y), z(a,M).
```

bude klauzule

```
r(a,b) :-
p(X), q(Y), not(q(X,Y)), z(a,M).
```

Úkoly

1. Podívejme se na tento program:

   ```
   osobni(autoPepy).
   osobni(autoJany).
   nakladni(autoStandy).
   ma_vozik(autoJany).

   auto(X) :- osobni(X).
   auto(X) :- nakladni(X).

   velky_naklad(X) :- nakladni(X).
   velky_naklad(X) :- osobni(X), ma_vozik(X).

   maly_naklad(X) :- auto(X), not(velky_naklad(X)).
   ```

   Všimněte si poslední klauzule: tam právě máme negaci, a to na konci klauzule. Zkonzultujte tento program v Prologu a zadejte dotazy s následujícím významem:

   - Která auta jsou nákladní?
   - Která osobní auta nejsou od Jany? (zde potřebujete v dotazu dva predikáty a proměnnou, pomocí které je propojíte).

2. Vytvořte soubor s bází prologovských klauzulí podle těchto vět:

   - Panenka a autíčko jsou hračky, Popelka a Perníková chaloupka jsou pohádky.
   - Panenka je OK (bez poškození).
   - Hračky, které nejsou OK, jsou rozbité.
   - Rozbité věci jsou nebezpečné.
   - Pohádky a takové hračky, které nejsou nebezpečné, jsou pro děti.

Použijte následující predikáty:

`hracka(`$\langle co \rangle$`)`	`ok(`$\langle co \rangle$`)`	`nebezpecne(`$\langle co \rangle$`)`
`pohadka(`$\langle co \rangle$`)`	`rozbite(`$\langle co \rangle$`)`	`proDeti(`$\langle co \rangle$`)`

Uložte, konzultujte do Prologu a položte následující dotazy:

- Jaké hračky máme?
- Existují nějaké pohádky?
- Co je nebezpečné?
- Co je pro děti?
- Je něco pro děti?
- Které hračky nejsou rozbité? (tj. vypsat vše, co je hračka, ale není to rozbité)

Kapitola 3

Seznamy a související postupy

Rychlý náhled: V této kapitole se budeme zabývat pokročilejšími možnostmi programování v Prologu, konkrétně technikami postavenými na seznamech. Naučíme se vytvářet a používat seznamy, a také na nich stavět algoritmy.

Klíčová slova: Seznam, řadicí algoritmus, Euklidův algoritmus, pole, graf, prvočíslo, Eratosthenovo síto.

Cíle studia: Po prostudování této kapitoly se naučíte používat speciální datový typ pro seznamy a programovat úlohy postavené na tomto datovém typu.

3.1 Jak na seznamy

Seznam v Prologu je dynamická struktura skládající se z prvků jakéhokoliv typu. Pro délku seznamu není dán limit, ani tuto délku nedeklarujeme při vytvoření seznamu – může se měnit. To na druhou stranu může být i problém, protože při chybě v algoritmu může seznam růst tak dlouho, až zahltí paměť, o čemž jsme informováni chybovým hlášením.

3.1.1 Testování prvků seznamu

Seznam v Prologu je rekurzivní struktura obsahující posloupnost prvků. Skládá se z hlavy a těla. K hlavě (což je obvykle první prvek seznamu) máme jednodušší přístup, kdežto tělo je „vnořený“ seznam.

Příklad 3.1

Pokud je v hlavě seznamu prvek `a`, tělo je seznam `[b,c]`, lze tento seznam zapsat následovně:

```
[a, b, c]
[a | b, c]
.(a, [b,c])
.(a, .(b, .c,[]))
```


Obvyklou úlohou je zjišťování, zda určitý prvek je prvkem seznamu.

Příklad 3.2

Napíšeme predikát `prvek/2`, jehož první argument je hledaný prvek, druhý argument je seznam, v němž prvek hledáme.

```
prvek(X,[X|_]).
prvek(X,[_|T]) :-
    prvek(X,T).
```

Vyzkoušíme dotazy:

```
?- prvek(jablko,[tresen,jablko,hrusen]).
true .

?- prvek(X,[tresen,jablko,hrusen]).
X = tresen ;
X = jablko ;
X = hrusen ;
false.

?- prvek(X,[]).
false.
```

Účelem prvního dotazu je zjistit, zda zadaný prvek je či není v zadaném seznamu, účelem druhého a třetího je vypsat prvky seznamu.

Pozor na rozdíl:

- `[ H | T]` ... `T` je (pod)seznam
- `[ H , T]` ... `T` je prvek seznamu

Poznámka:

Pro tento účel ve skutečnosti nemusíme psát vlastní predikát, existuje předdefinovaný (ostatně jako pro další operace se seznamy), který se chová stejně:

`member/2`

Zde si píšeme vlastní predikáty pro operace se seznamem především proto, ať si uvědomíme, jak seznamy fungují, jinak není problém používat předdefinované predikáty.

Úkol

Přepište do programu výše uvedené klauzule pro predikát `prvek/2`. Dále zjistěte, jak konkrétně se má používat predikát `member/2` a zjistěte, jestli se tyto predikáty opravdu používají stejně.

3.1.2 Spojení dvou seznamů

Spojením dvou seznamů vznikne další seznam. Pro tuto operaci použijeme rekurzi.

Příklad 3.3

Každou rekurzi je třeba někdy zastavit, klauzuli pro tento účel tedy napíšeme jako první. Pak následuje rekurzivní klauzule.

```
spoj(L1,L2,L) :-
    L1=[],
    L=L2.

spoj(L1,L2,L) :-
    L1=[H|T1],
    L=[H|T],
    spoj(T1,L2,T).
```

Rekurzivní klauzule funguje takto: předej do výsledného seznamu hlavu prvního seznamu, na zbytek prvního seznamu spusť rekurzi.

Můžeme vyzkoušet tyto dotazy:

```
spoj([ab,xx],[123,5,mm],X).
spoj([a,b],[c,d],[a,b,c,d]).
spoj(X,[c,d],[a,b,c,d]).
```

Úkol

Na straně 12 je vysvětleno značení argumentů v nápovědě. Jak podle tohoto značení bude vypadat předpis pro použití predikátu `spoj/3`?

3.1.3 Odstranění prvku ze seznamu, přidání prvku do seznamu

Úkolem je prohledat seznam, a pokud tam najdeme zadaný prvek, odstranit ho. První argument je hledaný prvek, druhý argument je prohledávaný seznam. Výsledkem (v třetím argumentu) je seznam bez zadaného prvku.

Příklad 3.4

Opět použijeme rekurzivní postup. Nejdřív napíšeme klauzuli pro zastavení rekurze (pokud je nalezený prvek v hlavě prohledávaného seznamu, rekurze skončí a vrátí se tělo seznamu, tedy bez hlavy).

V rekurzivní klauzuli pak řešíme situaci, kdy v hlavě prohledávaného seznamu není hledaný prvek: pak rekurzívně voláme tentýž předikát, do argumentu pro prohledávaný seznam dáme tělo původního seznamu. Tedy vlastně prohledávaný seznam zkrátíme o jeden (ten první) prvek. Z toho vyplývá, že rekurze skončí nejpozději tehdy, když projdeme celý seznam (nebo tehdy, když prvek najdeme).

```
odstran(D,Lorig,Lvysl) :-
    Lorig=[D|T],
    Lvysl=T.

odstran(D,Lorig,Lvysl) :-
    Lorig=[H|T],
    Lvysl=[H|Tvysl],
    odstran(D,T,Tvysl).
```

Vyzkoušíme pár dotazů:

```
?- odstran(a,[b,a,c],S).
S = [b, c] ;
false.
?- odstran(m,[a,b,c],X).
false.

?- odstran(a,[b,a,c],[b,c]).
true ;
false.

?- odstran(a,[a,b,a,c],X).
X = [b, a, c] ;
X = [a, b, c] ;
false.
```

Jak vidíme, druhý a třetí dotaz nebyly úplně ideálně vyřešeny. U druhého je to jedno false navíc, u druhého došlo k odstranění pouze jednoho výskytu prvku (byla nalezena dvě řešení, v každém je odstraněn jeden výskyt).

Příklad 3.5

Zkusíme jiné řešení.

```
odstran2(_,[],[]).

odstran2(D,[D|T],Lvysl) :-
    odstran2(D,T,Lvysl).

odstran2(D,[H|T1],[H|T2]) :-
    D\=H,
    odstran2(D,T1,T2).
```

Máme tři klauzule. První z nich zastavuje rekurzi (použije se v případě, že prohledávaný seznam je prázdný). Druhá a třetí klauzule jsou rekurzivní. Druhá se použije tehdy, když prvním prvkem seznamu je právě hledaný prvek, jinak se použije třetí klauzule. Vyzkoušíme pár dotazů:

```
?- odstran2(a,[b,a,c],X).
X = [b, c] ;
false.

?- odstran2(m,[a,b,c],X).
X = [a, b, c] ;
false.
```

```
?- odstran2(a,[a,b,a,c],X).
X = [b, c] ;
false.
```

Výsledek je vždy vypsán, odstraní se všechny výskyty hledaného prvku.

Příklad 3.6

Vložit prvek na začátek seznamu je celkem jednoduché, prostě ho prohlásíme za novou hlavu seznamu (a původní seznam se stane tělem nového seznamu):

```
vloz(P,Seznam,[P|Seznam]).
```

Úkol

Pokuste se najít existující obdoby výše uvedených predikátů pro práci se seznamy.

3.1.4 Podseznam

Dalším typem úlohy je zjištění, zda určitý seznam je podseznamem jiného seznamu. Pro zjednodušení budeme předpokládat, že prázdný seznam není součástí jiného seznamu.

Příklad 3.7

Použijeme již dříve definovaný predikát spoj/3, který zjistí, zda první a druhý argument jsou po spojení stejné jako třetí argument:

```
spoj(L1,L2,L) :-
    L1=[],
    L=L2.

spoj(L1,L2,L) :-
    L1=[H|T1],
    L=[H|T],
    spoj(T1,L2,T).
```

Tento predikát jsme si definovali v jedné z předchozích sekcí, alternativně se dá použít předdefinovaný predikát `append/2`. Pokračujme dále. Jak zjistíme, že jeden seznam je podseznamem jiného? Pokud celkový seznam má tuto strukturu:

```
levýKontext,část,pravýKontext
```

Tedy musíme zjistit, jestli po připojení něčeho (pravého kontextu) k části vznikne seznam takový, že když k němu připojíme něco z levé strany (levý kontext), dostaneme celkový seznam. Aby nedošlo k přetečení zásobníku pri použití s proměnnou v prvním argumentu, převrátíme pořadí „spojovacích“ predikátů v těle klauzule:

```
podseznam(Cast,Celek) :-
    spoj(LevyKontext,Pom,Celek),
    spoj(Cast,PravyKontext,Pom),
    Cast \= [].
```

Drobná úprava – Prolog bude protestovat, že proměnné `PravyKontext` a `LevyKontext` jsou použity jen jednou a není je na co navázat, proto je nahradíme anonymními proměnnými (přirozeně – má tam být „něco“, tedy existenčně vázaný údaj):

```
podseznam(Cast,Celek) :-
    spoj(_,Pom,Celek),
    spoj(Cast,_,Pom),
    Cast \= [].
```

Úkoly

1. Vyzkoušejte, jestli kód z předchozího příkladu funguje tak jak má, například na těchto dotazech:

   ```
   podseznam([1,2],[1,2,a,b]).
   podseznam([1,2],[2,1,a,b]).
   podseznam([1,2],[a,1,b,c]).
   podseznam([1,2],[a,1,2,b]).
   ...
   podseznam(X,[a,b,c]).
   podseznam([a,b],X).
   ```

   V posledních dvou případech nezapomeňte, že středníkem žádáte o další řešení, tečkou ukončujete hledání.
2. Pokud by náš predikát byl v nápovědě, jak by v jeho specifikaci byly označeny jeho argumenty, když zjevně za první argument můžeme dosadit proměnnou?

3.1.5 Různé operace nad prvky seznamu

V seznamech máme často takové hodnoty, které je třeba porovnávat či používat ve výrazu. Proto si připomeneme, jak do proměnné přiřadit hodnotu nebo výsledek výrazu. Jak víme, je k dispozici operátor „`=`“ (rovnítko) a dále operátor „`is`“.

Příklad 3.8

Naprogramujeme si predikát pro součet prvků seznamu. Vyzkoušíme několik variant:

```
soucet1([],0).
soucet1([H|T],V) :-
    V=V1+H,
    soucet1(T,V1).

soucet2([],0).
soucet2([H|T],V) :-
    V is V1+H,
    soucet2(T,V1).

soucet3([],0).
soucet3([H|T],V) :-
    soucet3(T,V1),
    V is V1+H.
```

Vyzkoušíme postupně tyto tři predikáty na tomtéž seznamu:

```
?- soucet1([8,5,-3],X).
X = 0+ -3+5+8.
```

Aha, tak to asi nebude ono. V čem je problém? Použili jsme operátor rovnítko, který sice dokáže přiřadit do proměnné výsledek, ale výraz na pravé straně nevyhodnotí – přiřadí ho jako řetězec. To nechceme. Proto v druhé verzi máme operátor „`is`".

```
?- soucet2([8,5,-3],X).
ERROR: Arguments are not sufficiently instantiated
ERROR: In:
ERROR: [11] _8146 is _8152+8
ERROR: [10] soucet2([8,5|...],_8180) at d:/vyuka/prolog/priklady/pokus3.pl:10
ERROR:  [9] toplevel_call(user:user: ...) at d:/programy/swipl/boot/toplevel.pl:1173
```

Hm, tak to taky není ono. V čem je problém tentokrát? Operátor „`is`" opravdu umí zajistit, aby byl výraz na pravé straně vyhodnocen, ale vyžaduje, aby v čase volání vše v tom výrazu mělo přiřazenou hodnotu. Jenže proměnná `V1` ji ještě nemá, získá ji až po volání na následujícím řádku. Zbývá třetí varianta:

```
?- soucet3([8,5,-3],X).
X = 10.
```

Tak je to správně. Použili jsme operátor `is` a na pravé straně (za ním) je vše instanciováno, tedy má to předem jasnou hodnotu, proto lze výraz vyhodnotit a jeho výsledek přiřadit do proměnné na levé straně výrazu.

Poznámka:

To však neznamená, že atom s operátorem `is` vždy musí být za rekurzivním voláním (v našem případě za rekurzivním voláním predikátu `soucet3`). V tomto případě ano, ale v jiném případě může být efektivnější uvést v těle klauzule nejdřív výpočet a pak až související rekurzivní volání. Záleží na tom, s jakými proměnnými v daném výrazu pracujeme.

Zkusíme si k rekurzi přidat podmínku. Můžeme sečíst všechny prvky seznamu, pokud ovšem jde o čísla, která se dají sčítat.

Příklad 3.9

Vytvoříme predikát `soucetkladnych/2`, který vrátí součet kladných čísel v seznamu. Seznam bude prvním argumentem, proměnná pro součet je druhý argument.

```
soucetkladnych([],0) :-!.

soucetkladnych([H|T],K) :-
    H=<0,
    soucetkladnych(T,K).

soucetkladnych([H|T],K) :-
    H>0,
    soucetkladnych(T,K1),
    K is K1+H.
```

První klauzule zastavuje rekurzi. Použili jsme operátor řezu, protože nechceme, aby se zbytečně vypisoval výsledek false (ale v tomto případě to funguje jen v případě prázdného seznamu). Další dvě klauzule jsou rekurzivní. Jedna pro případ, že v hlavě je číslo menší nebo rovno 0, další pro případ, že v hlavě klauzule je číslo, které má být sečteno.

Vyzkoušíme pár dotazů:

```
?- soucetkladnych([2,14,-100,5,-20],N).
N = 21 ;
false.

?- soucetkladnych([-1,-2],N).
N = 0 ;
false.

?- soucetkladnych([],N).
N = 0.

?- soucetkladnych([2,4,-2],6).
true ;
false.

?- soucetkladnych([2,4,-2],5).
false.
```

Úkoly

1. Zamyslete se nad tím, jak by se dal kód pro sčítání kladných čísel v seznamu optimalizovat. Určitě není příjemné, že se „nadměrně“ vypisuje řetězec `false`. Dá se použít zelený řez i jinde než v predikátu pro prázdný seznam?
2. Vymyslete si další možné operace nad prvky seznamu. Můžete například vyhledávat minimum či maximum seznamu nebo zjišťovat, zda určitá hodnota je či není prvkem seznamu.

3.2 Ladění kódu

Na seznamech si ukážeme, jak se provádí trasování ve SWI-Prologu. Trasování znamená, že nechceme „jen“ vypsat výsledek, ale zajímá nás, jaké kroky Prolog postupně provedl, aby se k tomu výsledku dostal, včetně výpisu hodnot proměnných za běhu v jednotlivých krocích výpočtu, a také s informacemi o zanoření v rekurzi.

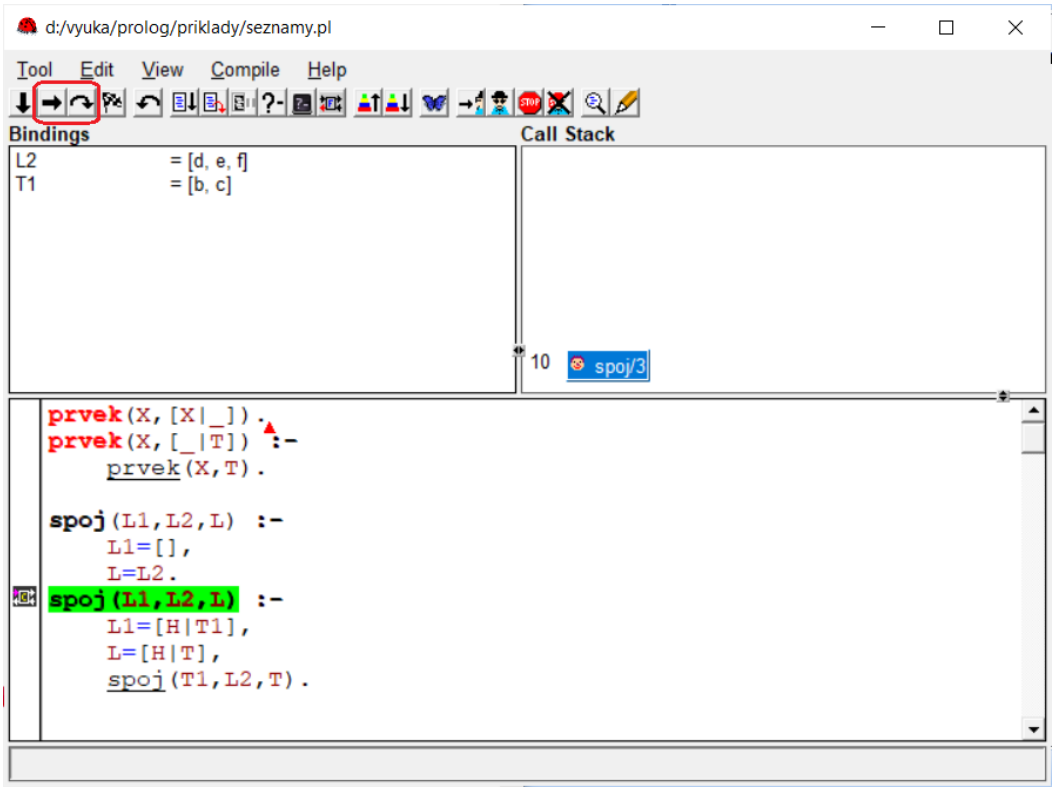
Pro nás bude kromě hodnot proměnných důležitý stav zásobníku. V něm uvidíme, která cílová klauzule se zrovna řeší, a taky k jaké se při návratu z rekurze vrátíme, tedy vazby mezi klauzulemi použitými při konstrukci dané větve ve výpočetním stromě.

Ve SWI-Prologu se dá trasovat buď přímo v konzoli, nebo v samostatném okně, každému bude vyhovovat něco trochu jiného. Pokud chceme použít trasování v grafickém okně, zvolíme v menu *Debug – Graphical Debugger*, jinak si této položky nevšímáme.

Trasovací režim zapneme přes menu *Run – Interrupt*. Prolog se zeptá, co konkrétně chceme provést. Můžeme/nemusíme stisknout „h“ (chceme nápovědu), pak Prolog vypíše:

```
?-
Action (h for help) ? Options:
a:            abort          b:            break
c:            continue       e:            exit
g:            goals          s:            C-backtrace
t:            trace          p:              Show PID
h (?):        help
```

Stiskneme „t“ (tedy chceme trasovat). Zadáme dotaz, třeba `spoj([a,b,c],[d,e,f],V).`

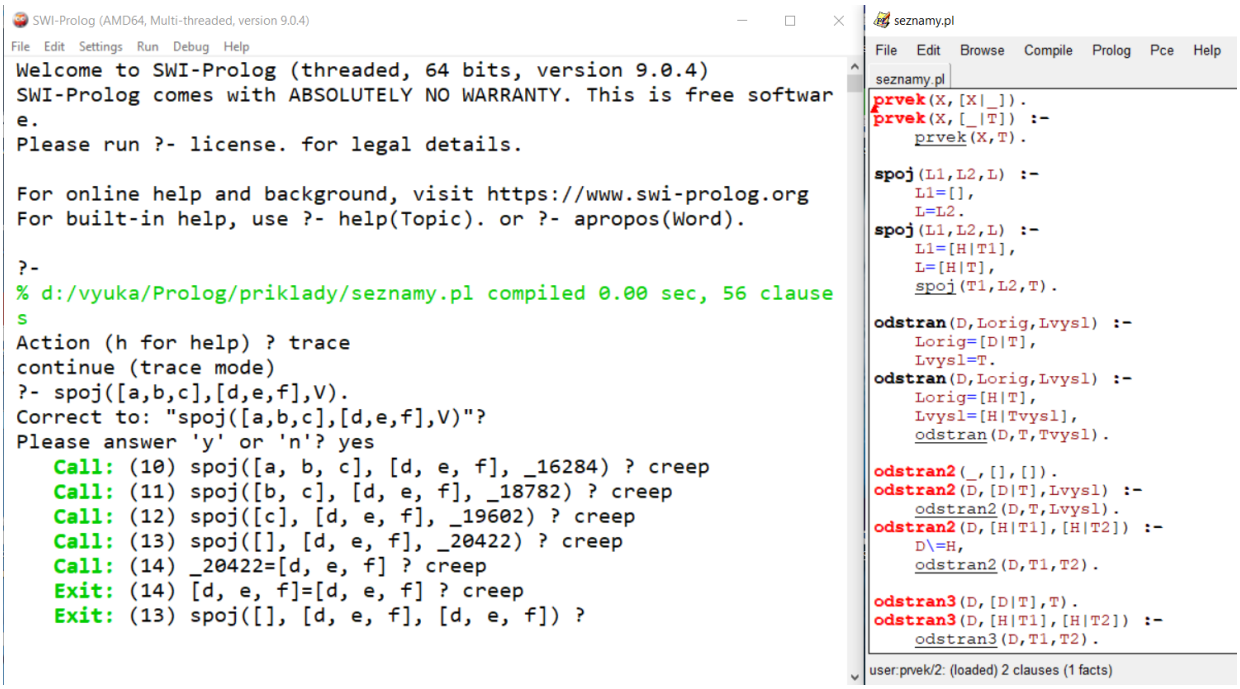


Obrázek 3.1: Trasování vyhodnocování dotazu na spojení dvou seznamů

Pokud jsme předtím zvolili debugování v grafickém módu, objeví se okno s podrobnými údaji o probíhajícím výpočtu – viz obrázek 3.1. A můžeme krokovat. V novém okně nahoře v panelu nástrojů je několik ikon, z nich nás budou zajímat ty zvýrazněné (na obrázku 3.1 nahoře): postupné krokování a „rychlejší“ krokování, při kterém nejdeme do dílčích cílů (jsou prováděny bez podrobného krokování). Místo klepání myší na šipku můžeme použít klávesu mezerník. Je tam také tlačítko „Continue without debugging“, které zruší trasovací režim.

V podokně *Bindings* vidíme momentální hodnotu proměnných (všechny proměnné v Prologu jsou lokální, vidíme jen proměnné unifikované v právě prováděné klauzuli). Vedlejší podokno *Call Stack* ukazuje obsah zásobníku s „rozpracovanými“ klauzulemi, ve spodním podokně je program.

Na obrázku 3.2 vidíme případ, kdy jsme nezvolili debugování v grafickém režimu, ale zůstáváme v konzole. Zde už jsme ve fázi navracení z rekurze, postupně se vypisují mezivýsledky (hodnoty atributů – lokálních proměnných).



Obrázek 3.2: Trasování v konzoli

V obou případech se po krocích posouváme klepáním na mezerník. Grafický režim je pro většinu uživatelů zřejmě pohodlnější a přehlednější, ladění v konzole zase nevyžaduje přecházení mezi okny, výstup svou formou odpovídá atomům.

Úkol

Vytrasujte si postupně dotazy na různé predikáty, které jsme probírali v této sekci.

3.3 Další algoritmy se seznamy

3.3.1 Permutace

Permutace známe z matematiky – je dána určitá posloupnost a naším úkolem je najít všechny možné posloupnosti skládající se ze stejných prvků, ale s jiným pořadím.

Příklad 3.10

Půjde o rekurzivní algoritmus, který bude procházet stavový prostor (tedy prostor všech možných řešení), přičemž původní seznam postupně rozloží na menší a menší podseznamy, a postupně vypisovat výsledky.

Prvním argumentem našeho predikátu je původní seznam, druhým prvkem je některá jeho permutace. Typické volání bude takové, že do prvního argumentu dosadíme seznam, který chceme „zpřeházet“, do druhého pak umístíme proměnnou, Prolog nám pak bude vypisovat různé možnosti, které se dají do této proměnné dosadit (tedy různé permutace prvního argumentu).

Nejdřív ošetříme ukončení rekurze, tedy situaci, kdy už máme prázdný seznam. Prázdný seznam nelze dál rozdělit, ve stavovém prostoru jsme se dostali na konec větve a nic nebudeme vypisovat:

```
permutace([],[]).
```

A teď k rekurzi. Budeme postupvat tak, že z prvního argumentu odstraníme první prvek druhého argumentu (v tom prvním byl pravděpodobně někde uvnitř seznamu, v druhém je na začátku). Tím seznamy zmenšíme o jeden prvek, načež rekurzívně zavoláme náš predikát, aby se totéž provedlo na seznamech o ten jeden prvek kratších. Prolog spustí postupně takových výpočtů více (pro různé formy seznamu v druhém argumentu), čímž vznikne strom různých řešení, pro každou permutaci původního seznamu jedna větev. Na konci každé větve jsou oba seznamy prázdné (to už jsme ošetřili). Pouze ty větve, kde při „likvidaci" prvků obou seznamů zůstáváme pouze u prvků vyskytujících se v tom prvním, skončí úspěchem (a tedy druhý argument se vypíše).

Použijeme již dříve definovaný predikát `odstran/3`, jehož prvním argumentem je prvek, který chceme odstranit, druhý argument je původní seznam, třetí argument je seznam po odstranění daného prvku. V našem případě

```
permutace(S,[H|T]) :-
    odstran(H,S,SKratsi),
    permutace(SKratsi,T).
```

Můžeme vyzkoušet na tomto dotazu:

```
permutace([a,b,c],X).
```

Úkol

V kódu jsme použili predikát `odstran`, ale výše jsme zkoušeli jeho další variantu `odstran2`. Myslíte, že by se zde dal použít?

3.3.2 Očísluj seznam

Máme seznam jakýchkoliv prvků, třeba řetězců, a chceme ho vypsat jako číslovaný seznam (položky pod sebou, každá opatřená pořadovým číslem). Jak na to?

Bude dobré si vytvořit pomocný predikát, který bude mít buď jiný název, nebo sice stejný název, ale jiný počet argumentů. My zvolíme druhou možnost, ale samozřejmě je použitelná i ta první.

Příklad 3.11

Není to nic složitého, jen na rozdíl od předchozích příkladů využijeme i předdefinované predikáty pro výstup.

Hlavní predikát bude `ocisluj/1`, jehož jediným argumentem bude seznam k očíslování. Pomocný predikát bude `ocisluj/2`, kde první prvek bude seznam, druhý prvek bude pořadové

číslo, od kterého se má číslovat. Přesněji – tento predikát budeme volat rekurzívně, přičemž v každém kroku vypíšeme jeden prvek seznamu s daným pořadovým číslem a rekurzívně zavoláme tentýž predikát, ale s kratším seznamem a číslem o 1 vyšším.

```
ocisluj(S) :-
    ocisluj(S,1).

ocisluj([],_) :-
    nl.
ocisluj([H|T],C) :-
    write(C),
    write('. '),
    write(H),
    nl,
    C1 is C+1,
    ocisluj(T,C1).
```

V každém kroku tedy vypíšeme pořadové číslo předávané v čítači, tečku, hlavu seznamu, zvýšíme čítač o 1 a provedeme rekurzi (předáme z původního seznamu jen tělo a zvýšený čítač).

Úkol

Vyzkoušejte predikát pro očíslování seznamu nejdřív takto:

```
ocisluj([jahody,hrusky,tresne,jablka]).
```

Dále vyzkoušejte, jestli bude správně fungovat na prázdný seznam. A co když budeme chtít, aby číslování začalo od jiného čísla než od 1?

3.3.3 Simulace pole

V Prologu nemáme datový typ pole, ale typ seznam má určité podobné vlastnosti. Vytvoříme predikát, který vypíše i-tý prvek seznamu, což už trochu připomíná práci s polem.

Příklad 3.12

Vytvoříme predikát se třemi argumenty. První argument bude seznam, druhý index prvku, který nás zajímá, a třetí bude proměnná, do které se má daný prvek uložit.

Použijeme rekurzi. Rekurze bude končit tehdy, když v hlavě seznamu je hledaný prvek a zároveň hledaný index je 0. Pokud tam není (je tam cokoliv jiného), odsekneme první prvek a rekurzívně spustíme predikát na tělo seznamu a index o 1 menší.

Nejdřív uvedeme klauzuli zastavující rekurzi:

```
ityprvek([H|_],0,H) :-!.
```

Operátor řezu není až tak nutný, jde o zelený řez, jehož účelem je zefektivnit výpočet (odříznou se ty větve výpočtu, které vedou k závěru `false`).

Jak vidíme, rekurze se zastaví tehdy, když v hlavě seznamu je náš prvek a zároveň index je 0, vypíše se tedy první prvek seznamu.

Teď rekurzívní klauzule:

```
ityprvek([_|T],N,P) :-
    N1 is N-1, ityprvek(T,N1,P).
```

Protože jsme tu nerekurzivní klauzuli uvedli nejdřív a na jejím konci je predikát řezu, druhá klauzule dostane slovo jen tehdy, když první klauzuli nelze použít (unifikovat).

Klauzule jsou sestaveny tak, aby v případě, že index neodpovídá existujícímu prvku (počítá se od 0), dotaz vrátí `false`. A také pokud hledaný prvek nechceme vypsat, ale jen zjistit, jestli existuje, použijeme anonymní proměnnou místo posledního parametru a výsledkem bude `true` nebo `false`, podle toho, zda prvek s daným indexem existuje nebo ne.

Můžeme vyzkoušet na následujících dotazech:

```
ityprvek([a,b,c],0,X).
ityprvek([a,b,c],2,X).
ityprvek([a,b,c],5,X).
ityprvek([a,b,c],-2,X).
ityprvek([],1,X).
ityprvek([a,b,c],1,_).
ityprvek([a,b,c],N,c).
```

Rekurzívně procházíme seznam od začátku, ale hledaný index naopak snižujeme o 1. Když najdeme i-tý prvek, index uložený v proměnné `N` už bude mít hodnotu 0. Vlastně odpočítáváme prvky od začátku, a není jiná možnost jak hlídat, kdy se „dopočítáme“, než se právě budeme odečítat jedničku.

Příklad 3.13

Můžeme si to vyzkoušet – provedeme drobnou úpravu:

```
ityprvek([H|_],0,H) :-!.
ityprvek([H|T],N,P) :-
    N1 is N-1,
    write(N),write(H),nl,
    ityprvek(T,N1,P).
```

Tedy v případě, že ještě nejsme na N-tém prvku, vypíšeme právě prověřované číslo N, zařádkujeme a jdeme do další úrovně rekurze. Můžeme vyzkoušet na tomto dotazu:

```
?- ityprvek([a,b,c,d,e],4,X).
4a
3b
2c
1d
X = e.
```

Prvek „`a`“ byl přeskočen, protože index není 0, ale vypsal se momentální index (4) a snížil se o 1, atd.

Další typický úkol při práci s polem je přiřazení konkrétní hodnoty do pole na danou pozici. Vytvoříme predikát `dosad/4`, který dělá právě toto.

Příklad 3.14

Predikát ttsmalldosad/4 má 4 argumenty: první je prvek, který chceme do pole vpašovat, další je index, na který chceme tento prvek dosadit. Třetí argument je pole v původním tvaru, poslední argument je výsledek.

Budeme opět postupovat rekurzívně, index postupne dekrementujeme, nahrazení provedeme ve chvíli, kdy index bude mít hodnotu 0, tedy když se dostaneme k hlavě seznamu.

Nejdřív ošetříme právě tu akční část, tedy situaci, kdy index je 0 a naším úkolem je dosadit prvek do hlavy seznamu (na začátek):

```
dosad(P,K,S1,S2) :-
    K==0,
    S1=[_|T],
    S2=[P|T],!.
```

První řádek těla klauzule ověřuje, že index `K` je 0 (pozor, dvojité rovnítko). Následuje atom, který klauzuli sděluje, jak je označeno tělo prvního seznamu, v dalším atomu říkáme, že ve výsledku má být tělo seznamu stejné, ale hlava má obsahovat náš prvek `P`.

Vykřičník na konci je řez, v našem případě jde o zelený řez, protože chceme zamezit zbytečnému hledání dalšího řešení (které neexistuje). Důsledkem je, že po úspěšném výpisu výsledku se neobjeví dodatečný řádek „`false`".

A teď rekurzivní klauzule řešící posun na index o 1 menší:

```
dosad(P,K,S1,S2) :-
    K>0,
    S1=[H|T],
    S2=[H|T1],
    K1 is K-1,
    dosad(P,K1,T,T1).
```

Všimněte si, že naše dvě klauzule se stejnou hlavou se použijí v disjunktních případech, což zajišťuje hned první atom: v tomto případě určuje, že se klauzule použije pouze v případě, že `K>0`.

Proč seznamy `S1` a `S2` mají v hlavě stejný prvek? Protože takto z původního seznamu `S1` do nového seznamu `S2` postupně transportujeme všechny prvky, přesněji – instancujeme hlavu druhého seznamu s obsahem hlavy prvního seznamu.

Úkoly

1. Vyzkoušejte výše definovaný predikát `dosad/4` na těchto dotazech:

   ```
   dosad(a,0,[x,y,z],V).
   dosad(a,1,[x,y,z],V).
   dosad(a,2,[x,y,z],V).
   dosad(a,3,[x,y,z],V).
   ```

   Proč v posledním případě skončil dotaz neúspěchem?

2. Na konci příkladu je vysvětlení, proč v druhé klauzuli mají oba seznamy v hlavě stejnou proměnnou. Ověřte, že to vysvětlení platí. Pozměňte predikát takto:

```
dosad(P,K,S1,S2) :-
    K>0,
    S1=[H|T],
    S2=[H1|T1],
    K1 is K-1,
    dosad(P,K1,T,T1).
```

Ignorujte hlášení Prologu o „singleton variables“ (takto se vás pokouší donutit k tomu, abyste místo `H` a `H1` použili anonymní proměnnou, protože se v klauzuli s ničím neunifikují) a položte dotazy z předchozího úkolu. Jak vypadá výsledek? Proč?

3.4 Řadicí algoritmy

Z jiných programovacích jazyků známe různé řadicí algoritmy, kde je cílem seřadit seznam podle zadaného kritéria (třeba podle abecedy, pokud se jedná o řetězce, nebo velikosti, když se jedná o čísla). Také v Prologu se dá řadit.

Budeme předpokládat, že v seznamu jsou čísla, jako kritérium pro řazení použijeme běžnou relaci „menší“.

Je třeba si uvědomit, že algoritmy pro deklarativní programování týkající se seznamů nebývají vždy až tak intuitivní jak by se dalo čekat. U seznamů se často dostáváme do situace, kdy máme dva seznamy, přičemž jeden je „zdroj“ a druhý je „cíl“, a algoritmus pak může pracovat tak, že za základě údajů v jednom seznamu změníme data v druhém. Nebo jeden seznam rozdělíme na dva, s oběma či jedním něco provedeme (třeba spustíme rekurzívně tentýž výpočet, který vyžaduje rozdělení) a pak je zase spojíme.


Algoritmus BubbleSort je velmi jednoduchý, prostě postupně necháváme „probublávat“ jednotlivé prvky tam, kam patří.

Příklad 3.15

Naprogramujeme si algoritmus BubbleSort. Potřebujeme pomocný predikát `swap/2`, jehož dva argumenty jsou seznamy. První je zdrojový, druhý cílový. Pokud ve zdrojovém jsou první dva prvky ve „špatném“ pořadí, v druhém je přehodíme. Pokud oba seznamy začínají stejným prvkem, pustíme algoritmus rekurzívně na další prvky.

```
swap([X,Y|Zbytek],[Y,X|Zbytek]) :-
    X>Y.

swap([X|Zbytek],[X|Zbytek1]) :-
    swap(Zbytek,Zbytek1).
```

V první klauzuli jsme nepoužili predikát řezu, tedy druhá klauzule se použije i v případě, že první klauzule skončila úspěchem (true). Důsledkem je, že v rekurzi projdeme celý „levý“ seznam a otestujeme vždy dva sousední prvky. To samozřejmě ještě není všechno, bublinkové třídění znamená, že seznam je třeba takto zpracovat opakovaně, protože výměnou sousedních

prvků se opakovaně můžeme dostávat do stavu, kdy vedle sebe dostaneme prvky ve „špatném“ pořadí.

A teď samotný třídicí algoritmus. Predikát bude mít také dva argumenty, oba seznamy. První argument chceme setřídit, do druhého má být uložen výsledek.

```
bubblesort(Seznam,Vysledek) :-
    swap(Seznam,S1),!,
    bubblesort(S1,Vysledek).

bubblesort(Seznam,Seznam).
```

Nejdřív projdeme `Seznam` a srovnáme vždy dvojice sousedících prvků. Pokud jsou ve špatném pořadí, budou přehozeny. Pak se pokračuje v rekurzi. Všimněte si použití řezu. Důsledkem je, že větev, ve které nedojde k žádné operaci (tj. v ní je už seznam celý správně seřazen) bude odříznuta a při správně seřazeném seznamu nebude vypsána hláška `false`. V tomto případě se jedná o červený řez, protože nejde jen o odříznutí nadbytečných částí výpočtu, ale výpočet je přímo ovlivněn.

Úkoly

1. Vyzkoušejte predikát `bubblesort/2` například těmito dotazy (nebo podobnými):

   ```
   bubblesort([25,-4,8,0,90],X).
   bubblesort([],X).
   bubblesort([4,-2,10],[-2,4,10]).
   bubblesort([4,-2,10],[4,5,10]).
   bubblesort([4,-2,10],[]).
   ```

2. Co se stane, když zakomentujete poslední řádek (tj. `bubblesort(Seznam,Seznam).`)?

Algoritmus QuickSort stojí na principu „rozděl a panuj“, tedy řazený blok vždy rozdělí na dvě části a přeuspořádá prvky mezi těmito dvěma částmi (seznamy) tak, aby v levém seznamu byly prvky menší než „středový prvek“ (který se nazývá pivot).

Příklad 3.16

Predikát `rozdel/4` provádí rozdělení a přeuspořádání podle pivota.

```
rozdel(_,[],[],[]).
rozdel(H,[X|T1],[X|T2],S3) :-
    H=<X,
    rozdel(H,T1,T2,S3).

rozdel(H,[X|T1],S2,[X|T3]) :-
    H>X,
    rozdel(H,T1,S2,T3).
```

Samotný řadicí predikát bude mít dva parametry, použijeme pomocný se třemi parametry.

```
quicksort(Puvodni,Tridene) :-
    qsort(Puvodni,[],Tridene).

qsort([],Acc,Acc).

qsort([H|T],A,Trideny) :-
    rozdel(H,T,S1,S2),
    qsort(S1,A,Trideny1),
    qsort(S2,[H|Trideny1],Trideny).
```

Jak vidíme, rekurze se přelévá vždy do dvou částí.

Úkoly

1. Vyzkoušejte predikát `qicksort/2` těmito dotazy (nebo podobnými):

   ```
   quicksort([8,-2,104,5],X).
   quicksort([],X).
   quicksort([8,-2,104,5],_).
   quicksort([25,4,-1,10],[-1,4,10,25]).
   quicksort([25,4,-1,10],[-1,4,10]).
   ```

2. Program zbytečně hledá i taková řešení, která nás nezajímají. Zamyslete se nad tím, jak zajistit, aby se vrátilo vždy jen jedno (správné) řešení. Nápověda: zaměřte se na predikát `rozdel/4`, přesněji na všechny jeho klauzule. Pak vyzkoušejte výše uvedené dotazy.

3.5 Hledání cesty v grafu

Graf je datová struktura zachycující prvky a propojení mezi nimi, tedy zda mezi konkrétní dvojicí prvků je či není spojení. Může být zachyceno buď pouze to, zda dané spojení je/není, nebo i další informace, jako třeba délka či směr cesty, ale to zde řešit nebudeme.

Pokud chceme zachytit graf pomocí seznamu v Prologu, musíme nejdřív definovat přímá spojení, tedy „sousedství“. Budeme řešit úlohu hledání cesty v grafu, seznam tedy bude právě výsledná cesta (seznam prvků, přes které vede cesta mezi zadanými krajními body). Seznamy zde budeme spíše používat pracovně pro generování různých řešení, výstupem bude jeden nebo více takových seznamů.

Příklad 3.17

Abychom si náš úkol přiblížili úlohám ze skutečného života, použijeme jako prvky (uzly) grafu státy, sousedství určíme jako dvojice států se společnou hranicí, výstupem pro cestu mezi dvěma státy bude seznam států, přes které cesta povede.

Nejdřív si tedy definujeme sousedy. Seznam sousedů jen pro Evropu by byl hodně dlouhý, zde se smíříme pouze s částečnou informací.

```
soused(cesko,slovensko).
soused(cesko,rakousko).
soused(cesko,nemecko).
```

```
soused(rakousko,svycarsko).
soused(slovensko,madarsko).
soused(cesko,polsko).
soused(slovensko,polsko).
soused(slovensko,ukrajina).
soused(polsko,ukrajina).
soused(madarsko,ukrajina).
soused(nemecko,svycarsko).
soused(svycarsko,italie).
soused(nemecko,nizozemsko).
soused(nizozemsko,belgie).
soused(belgie,francie).
soused(madarsko,rumunsko).
```

Protože nechceme rozlišovat směr pohybu, definujeme si predikát `pruchod/2`, který budeme používat místo předchozího predikátu, ať nemusíme rozlišovat různé směry:

```
pruchod(X,Y) :-
    soused(X,Y).
pruchod(X,Y) :-
    soused(Y,X).
```

Aby to uživatel měl jednodušší, použijeme již dříve vysvětlenou techniku dvou predikátů: hlavní (uživatelský) predikát má jen tři argumenty (první dva jsou státy, mezi kterými hledáme cestu, třetí argument je výsledný seznam).

V těle tohoto uživatelského predikátu je volán složitější „pracovní“ predikát (zde stejného jména, ale to nevadí – má jiný počet argumentů, tedy se technicky jedná o jiný predikát). V pracovním predikátu jsou čtyři argumenty: první dva jsou opět země, které chceme propojit, v třetím argumentu ukládáme mezivýsledek, čtvrtý je výsledek.

```
cesta(X,Y,Z) :-
    cesta(X,Y,[],Z).

% případ, kdy cestuji do téže země:
cesta(X,X,_,[X]) :- !.

cesta(X,Y,Projdeno,[X|Hotovo]) :-
    not(member(X,Projdeno)),
    pruchod(X,Z),
    cesta(Z,Y,[X|Projdeno],Hotovo).
```

Také zde využíváme rekurzi. Vestavěný predikát `member/2`, který vidíme v rekurzivní klauzuli, vrací true/false podle toho, zda první argument (`X`) je prvkem seznamu v druhém argumentu. Účelem je předejít zacyklení (postupně zkoumáme různé sousedy, přes které jsme ještě necestovali, proto ta negace).

Úkol

Vyzkoušejte výše napsaný program (upravte si sousedy dle uvážení), nechte si vypsat cesty mezi různými státy. Všimněte si, že se obvykle vypisuje několik řešení. Dal by se algoritmus zoptimalizovat, popřípadě vylepšit tak, aby vypisoval jen jedno (nejlepší) řešení? V našem případě by nejlepší byl ten seznam, který je nejkratší.

3.6 Eratosthenovo síto

Eratosthenes z Kyrény (Kyréna je v dnešní Libyi) byl jedním z nejvýznamnějších starověkých matematiků. Dokázal nečekaně přesně vypočítat obvod planety Země, a kromě toho (a kromě mnohého jiného) se zabýval prvočísly.

Eratosthenovo síto je algoritmus nalezení seznamu prvočísel z určitého rozsahu. Tento algoritmus postupně vyřazuje ze seznamu násobky čísla 2, 3, 4,..., a co v tomto sítu zůstane, to jsou prvočísla.

Vytvoříme program v Prologu, na kterém si procvičíme jak práci se seznamy, tak i komunikaci s uživatelem.

Příklad 3.18

Naším cílem je vytvořit predikát `prvocisla/0`, který

- se při svém provádění zeptá uživatele na horní hranici hledaných prvočísel,
- vygeneruje seznam lichých čísel (tímto si předzpracujeme seznam, ať nemusíme vyřazovat násobky čísla 2),
- postupně vyřadí násobky čísla 3, 4, atd.,
- vypíše výsledný seznam v řádcích tak, aby na každém řádku bylo 10 čísel.

Nejdřív to nejjednodušší, výstup. Sestavíme klauzule pro predikát `vypis/2`, jehož první argument je počet prvků na řádku, druhý argument je seznam, jehož prvky je třeba vypsat.

```
vypis(Rad,S) :-
    vypis(Rad,0,S).

vypis(_,_,[]) :- nl.
vypis(Rad,Rad,S) :-
    nl,
    vypis(Rad,0,S).
vypis(Rad,N,[H|T]) :-
    N<Rad,
    write(H), write(' '),
    N1 is N+1,
    vypis(Rad,N1,T).
```

Všimněte si, že jsme opět použili metodu komunikačního a pracovního predikátu. Komunikační má dva argumenty, pracovní má tři argumenty. Protože seznam k vypsání může mít různý počet prvků, použijeme rekurzi.

Rekurzivní klauzule zkontroluje, jestli má ještě zapisovat na tentýž řádek nebo přejít na další (`N<Rad`, resp. varianta kdy první dva argumenty jsou stejné – zde je nutno zařádkovat). Vypisuje se hlava seznamu, oddělovač (mezera) a pak se rekurzívně zavolá s indexem na řádku o 1 vyšším a se seznamem o hlavu kratším.

Dále se zaměříme na generování seznamu lichých čísel. Hlavním predikátem (který bude volán jinými predikáty) je `seznamlichych/2`, pracovní predikát `sezn/3` provádí v rekurzi samotné generování (opustili jsme pravidlo stejného názvu obou predikátů, je to moc dlouhé slovo...).

```
%vytvori seznam lichych cisel <= N
seznamlichych(N,S) :-
    sezn(3,N,S).
```

```
%vygeneruje seznam cisel H, H+2, H+4, ...N
sezn(N,N,[N]).
sezn(Velke,N,[]) :-
    Velke > N.
sezn(H,N,[H|T]) :-
    H<N,
    H1 is H+2,
    sezn(H1,N,T).
```

Všimněte si, že začínáme číslem 3. Proč? Násobky čísla 2 už totiž máme pryč, samotné číslo 2 přidáme „ručně“. Číslo přidané v předchozím kroku máme zapamatované v proměnné `H` (přesněji – jde o hlavu postupně tvořeného seznamu, generovaný seznam je totiž od největšího po nejmenší číslo).

Vyzkoušeli jsme funkčnost klauzulí pro predikát na generování lichých čísel, teď se pustíme do predikátu odstraňujícího ze seznamu všechny násobky konkrétního čísla:

```
%vyhod(C,P,S,V):
%vyhodi ze seznamu S cisla P+k*C pro k=0,1,2,...

vyhod(_,_,[],[]).

vyhod(C,P,[P|T],V) :-
    P1 is P+C,
    vyhod(C,P1,T,V).

vyhod(C,P,[H|T],V) :-
    P<H,
    P1 is P+C,
    vyhod(C,P1,[H|T],V).

vyhod(C,P,[H|T],[H|V]) :-
    P>H,
    vyhod(C,P,T,V).
```

Protřídění seznamu má na starosti predikát `vyhod/4`. První argument je číslo, jehož násobky chceme odstranit, druhý argument je startovací pozice pro vytřiďování, třetí argument je seznam ke zpracování, do čtvrtého argumentu se načte výsledek. Všimněte si, že rekurzívní volání je jako poslední atom v klauzulích predikátu `vyhod/4`, tedy těch, kde se rekurze vyskytuje. Klauzule ukončující rekurzi se použije až tehdy, kdy výpočet projde celým seznamem a už nezbývá nic k vytřídění (tj. máme prázdný seznam).

V klauzulích je taky ošetřeno, aby se algoritmus nepokoušel vyřazovat prvky, které v seznamu nejsou – porovnáním `P<H`, resp. `P>H`. V druhém případě pouze dokončíme rekurzi bez dalšího vyhazování (ale rekurzi dokončit musíme, aby se správně provedl backtracking – navracení).

Zastřešující predikáty pro síto jsou `sito/4` a `eratosthenes/3`. Máme zde také pomocný predikát `obratit/3`, který spojí seznamy v prvních dvou argumentech, přičemž v prvním zároveň obrátí pořadí, výsledek je v třetím argumentu.

```
% vezme seznam S a vsechna prvocisla z nej zaradi do E, klesajici
eratosthenes(N,S,E) :-
    sito(N,S,[],E).
```

```
% pomocny predikat sito(max,seznam1,seznam2,seznam3) pro predchozi
sito(_,[],S,S).
sito(N,[H|T],S,P) :-
    H*H=<N,
    H1 is H+H,
    vyhod(H,H1,T,T1), !,
    sito(N,T1,[H|S],P).
sito(N,[H|T],S,P) :-
    H*H>N,
    obratit(T,[H|S],P).

% obratit(R,S,V): V bude zretezeni obraceneho seznamu R se seznamem S
obratit([],S,S).
obratit([H|T],S,V) :-
    obratit(T,[H|S],V).
```

Predikát `sito/4` postupně volá predikát `vyhod/4` a postupně ze seznamu odstraňuje násobky jiných prvků, po dokončení rekurze by v seznamu měla zůstat jen prvočísla.

Predikát `eratosthenes/3` zastřešuje protřiďovací část. Jako vstupní seznam přijímá seznam lichých čísel, výstupem je protříděný seznam (ale v opačném pořadí). Můžeme vyzkoušet třeba na

```
eratosthenes(20,[3,5,7,9,11,13,15,17,19],V).
```

Zbývá hlavní komunikační predikát, který nepotřebuje žádné argumenty, protože s uživatelem komunikuje pomocí vstupních a výstupních predikátů:

```
prvocislo :-
    write('Zadej cislo: '),
    read(N), N>1,
    seznamlichych(N,S),
    eratosthenes(N,S,Obraceny),
    obratit(Obraceny,[],Vysl),
    vypis(10,[2|Vysl]).
```

Od uživatele si vyžádáme číslo (pozor, uživatel musí svůj vstup ukončit tečkou, i když jde o zadání čísla), pak vygenerujeme seznam lichých čísel, následující predikáty zajistí protřídní a výstup. Všimněte si, že hodnota zadaná uživatelem se předává v argumentech několika následujících predikátů. Číslo 10 v posledním atomu (predikát `vypis/2` určuje, že se výstup má rozdělit na řádky po 10 číslech.

Úkoly

1. Vyzkoušejte predikát `prvocisla/0`, zadávejte různé horní hranice pro síto.
2. Na jednotlivých predikátech si procvičte trasování:

   ```
   seznamlichych(20,S).
   vyhod(3,5,[3,5,7,9,11,13],V).
   ```

   a další podle možností.

Literatura

[1] BEN-ARI, Mordechai. *Mathematical Logic for Computer Science* (second edition). Springer-Verlag (2001). Dostupné také na: https://link.springer.com/book/10.1007/978-1-4471-4129-7

[2] BLACKBURN, Patrick, Johan BOS, Kristina STRIEGNITZ. Learn Prolog Now! [online]. *SWI-Prolog.org* [cit. 2023-03-02]. Dostupné na: https://www.let.rug.nl/bos/lpn/

[3] Expert Systems in Prolog [online]. *Amzi!* [cit. 2023-12-27]. Dostupné na: http://www.amzi.com/ExpertSystemsInProlog/xsiptop.php

[4] GAHÉR, František. *Logické hádanky, hlavolamy a paradoxy.* Iris Bratislava (1997).

[5] KRYL, Rudolf. *Úvod do programovacího jazyka Prolog*. KSVI MFF UK Praha. Online. cuni.cz [cit. 2024-03-26]. Dostupné na: https://ksvi.mff.cuni.cz/~kryl/prolog.pdf

[6] PAVLÍČEK, Jiří, Peter MIKULECKÝ, Josef HYNEK. *Logické programování a Prolog*. Gaudeamus Hradec Králové (1995). ISBN 80-7041-253-4.

[7] SUBER, Peter. *Logical Systems*. Skripta Earlham College Richmond, Indiana. Související materiály jsou dostupné na: http://legacy.earlham.edu/~peters/courses/logsys/lshome.htm [cit. 2023-12-27].

[8] SWI Prolog Reference manual [online]. *SWI-Prolog.org* [cit. 2023-03-02]. Dostupné na: https://www.swi-prolog.org/pldoc/doc_for?object=manual